How to Use a 64-Bit Virtual Address Space

Jeffrey S. Chase, Henry M. Levy, Miche Baker-Harvey, and Edward D. Lazowska
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195

Technical Report 92-03-02

Abstract

Most operating systems execute programs in private address spaces communicating through messages or files. The traditional private address space model was developed for 16- and 32-bit architectures, on which virtual addresses are a scarce resource.

The recent appearance of architectures with flat 64-bit virtual addressing opens an opportunity to reconsider our use of virtual address spaces. In this paper we argue for an alternative addressing model, in which all programs and data reside in a single *global virtual address space* shared by multiple protection domains. Hardware-based memory protection exists within the single address space, providing firewalls as strong as in conventional systems.

We explore the tradeoffs in the use of a global virtual address space relative to the private address space model. We contend that a shared address space can eliminate obstacles to efficient sharing and exchange of data structures that are inherent in private address space systems. The shared address space offers advantages similar to some capability-based computer systems, but can be implemented on standard page-based hardware, without the performance costs and restricted data model typical of capability-based systems.

We introduce Opal, an operating system to support this model on paged 64-bit architectures. The key feature of Opal is a flat virtual address space that includes data on long-term storage and across the network.

1 Introduction

The rapidly increasing demands of software (both applications and operating systems), together with increasing VLSI densities, have brought us to the verge of a major architectural change: the move from a 32-bit to a 64-bit virtual address space. This trend can be seen, for example, in the architectures of the HP PA-RISC [Lee 89], and the IBM RS/6000 [Groves & Oehler 90]. The MIPS R4000 [MIP 91] and Digital's recently announced Alpha family [Dobberpuhl et al. 92] are the first architectures with unsegmented 64-bit address spaces.

Unlike the move from 16- to 32-bit addressing, a 64-bit address space could be revolutionary instead of evolutionary with respect to the way operating systems and applications use virtual memory. Consider that 40 bits can address a terabyte, two orders of magnitude beyond the primary and secondary storage capacity of all but the largest systems today. More significantly, a full 64-bit address space, consumed at the rate of 100 megabytes per second, will last for nearly 5000 years. Such large address spaces eliminate the need to reuse addresses in order to supply each executing program with sufficient name space, invalidating a basic assumption that has driven operating system design since the 1960s.

The purpose of this paper is not to argue in favor of the architectural change to 64-bit addressing; we simply assume that wide-address architectures will appear, independently of how operating system designers decide to use them. Instead, our purpose is to argue that operating systems should take advantage of these wider addresses by supporting a single *global virtual address space* that maps all data in the system. The

This work is supported in part by the National Science Foundation under Grants No. CCR-8619663, CCR-8907666, CCR-9200832, and CDA-9123308; by the Washington Technology Center; by the Boeing Corporation; and by Digital Equipment Corporation through the Systems Research Center, DECwest Engineering, the External Research Program, and the Graduate Engineering Education Program.

"system" could be a distributed memory computer or even a large network, though we do not explicitly discuss distribution issues in this paper.

In this paper, we explore the costs and benefits of this model (the *uniform-address model*) by comparing and contrasting it with the memory addressing model used in most operating systems today (the *process-based model*). Both models define the concepts of *address space*, the virtual addresses used by an executing processor to name operands in memory, and *protection domain*, an execution context that restricts the processor's access to memory.[1] They differ only in their view of the relationship between these concepts:

- *Process-based* systems, or *private address space* systems, equate the notions of protection domain and address space, combining them into a single concept known as a *process* in Unix or Multics, a *task* in Mach, and an *actor* in Chorus. In these systems every protection domain has its own address space.
- *Uniform-address* systems, or *single address space systems*, execute all protection domains in a common paged virtual address space containing all data. The notions of address space and protection domain are fully independent.

As an example of the uniform-address model, we introduce Opal, a distributed operating system that we are building. An Opal protection domain is in many ways the analog of a Unix process or Mach task: for example, there is typically one protection domain per executing application, containing some private data, threads, and RPC connections to other domains. The difference is that an Opal domain has a private set of access privileges for globally addressable pages rather than a private virtual naming environment. Domains may grow, shrink, or overlap arbitrarily by sharing or tranferring page permissions among themselves. Opal's flat virtual address space includes data on long-term storage and across the network. In general, a given piece of data appears at a fixed virtual address independent of where it is stored or which domains access it.

No special hardware support for Opal is assumed or required. Our architectural assumptions are: (1) an unsegmented paged virtual address space, (2) 64-bit virtual addresses with 40 or more meaningful bits, (3) efficient sparse use of the address space, (4) support for multiple protection domains that control access to data on a page granularity, and (5) no constraints on virtual address usage within a protection domain. These conditions could be met by simply extending the addressing widths on a conventional paged architecture with a software-loaded translation buffer, such as the MIPS R3000. We believe that Opal could run efficiently on DEC Alpha and MIPS R4000 processors.

1.1 Why a Single Address Space?

The key property of a uniform-address system is that virtual addresses are *context-independent* – the meaning of a virtual address is independent of the entity issuing that address. We have noted elsewhere [Koldinger et al. 91] that context-independent addressing frees processor hardware from the need to support multiple sets of address translations, allowing optimizations in the memory system, notably the trouble-free use of a virtually addressed data cache. In the software, context-independent addressing simplifies the sharing, storage, and retrieval of data, especially *structured* data containing embedded pointers. It is this improved support for sharing that we are concerned with in this paper. This section previews the main points of our argument.

Our notion of *sharing* is quite general: sharing occurs *whenever a domain uses data or code that is created by some other domain*. By this definition sharing is common. Sharing can be sequential as well as concurrent, so it encompasses data transfer from one domain to another through messages or files. Distributed communication and long-term storage are special cases of sharing between domains, in which the domains are physically separated by a network or their lifetimes are disjoint.

Process-based systems intertwine naming and protection in a way that limits their ability to support sharing efficiently. The problem is that each domain chooses addresses for shared data independently of other domains; in general, this means that the producer and consumer of data use names inconsistently.

[1] We also use the term *protection domain* informally to denote the collection of data that is accessible from such an execution context, or to refer to a processor executing in the context.

Section 2 will show that these naming problems cause extra overheads when data is shared, such as excessive data copying and software pointer translation. Programs may be structured to make unnecessary cross-domain (RPC) calls because sharing is difficult to implement.

A single address space system separates naming and protection, simplifying sharing in two ways. First, virtual addresses can be passed between domains; any byte of data can be named with the same virtual address by any protection domain that has permission to access it. Second, domains can import data containing embedded virtual addresses, without risk of a name conflict with other data already in use by the domain. These properties make it easier to implement sharing as *direct sharing* of a single copy of the data.

1.2 Sharing and Trust

Improved support for direct sharing is beneficial even if there is no explicit interaction between protection domains. For example, an Opal domain can dynamically attach to shared code modules in any combination, without runtime linking (Section 3.3). More importantly, direct sharing can benefit domains that are cooperating in some way. Why do protection domains need to cooperate or share data? We generally think of each domain as executing a self-contained application, but in fact there are several reasons why interacting software components might run in separate protection domains:

- *Providing a service to multiple users.* Servers are protected from their clients, to prevent a client from interfering with service to other clients.
- *Accessing a database shared by multiple users.* Tools for directly accessing a common database execute with separate protection identities that reflect the access privileges of their users.
- *Program continuity.* By definition, a protection domain cannot survive a system restart. However a new domain may be created after a restart that picks up where an earlier one left off.
- *Distribution.* By definition, a protection domain cannot span nodes in a distributed system. However, domains on different nodes (perhaps executing different parts of the same application) may interact through message-passing or distributed shared memory.
- *Composing programs.* In most systems, code that is packaged as an executable program can be executed only by creating a new protection domain for it to run in. If we want to join the output of one program to the input of another, we connect two domains with a mechanism such as a Unix pipe.
- *Enforcing modularity.* A large application may be decomposed into multiple protection domains for software engineering reasons, e.g., to aid debugging by limiting the propagation of errors.

All of these arrangements are familiar in today's computing environments, and all of them involve different privilege and trust relationships between the cooperating domains. Composed programs could be fully trusting; if so, they might just as well be executed in the same protection domain, which is difficult to achieve in process-based systems but easy to achieve in Opal. Partial trust relationships are also common. For example, a server never trusts its clients, but each client must trust the server to some degree. A child domain always trusts its parent, though the parent never fully trusts the child. Database tools are mutually trusting to the extent that their user's access privileges overlap.

Opal provides several mechanisms to allow even untrusting domains to share a physical copy of data safely: (1) readonly access, (2) sequential sharing using transfers of access permissions, and (3) system-enforced page-level locking. Domains may use concurrently shared pages in restricted ways, validating the data before use or relying on language-level protection. Variations of these mechanisms are useful when there is partial trust. Section 3.4 outlines some of the possibilities.

Current software and architectural trends motivate more creative use of direct sharing. Data copying and protection domain switches have increased in cost relative to integer performance on recent processors [Ousterhout 90, Anderson et al. 91b]. At the same time, software systems are growing in complexity, encouraging decomposition into multiple protection domains. The recent move toward server-structured microkernel operating systems is one important example of this trend. Also, CAD systems and other data modeling applications are frequently structured as groups of cooperating database tools.

1.3 Related Work

Like much research in operating systems, the Opal project is driven by improvements in hardware technology. Uniform-address systems have been built before on the claim of improved support for sharing, but these systems suffered from the limitations of their hardware platforms. Only now is hardware advancing to the point that it is practical to organize a general-purpose operating system in this way.

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 were viewed as single-user systems because there was no hardware-based memory protection; both systems relied solely on defensive protection from the name scoping and type rules of a special-purpose 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. Psyche restricts its address space to data in active use on a single computer, partly because the system runs in a 32-bit address space. However, most of the benefits claimed for context-independent addressing in Opal apply equally to non-persistent storage in Psyche.

Systems with segmented addressing (e.g., Multics [Daley & Dennis 68]) have some of the properties of uniform-address systems. 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. Segmented architectures support uniform sharing in some cases, but with some or all of the following restrictions: (1) cross-segment pointers are not supported, (2) multiple pointer forms exist that must be treated differently by applications, and (3) application 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. Thus the Precision could support a uniform-address operating system. However, long-form pointer dereference requires a sequence of four instructions, so most software for the Precision 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) restricted data model, (2) poor performance, and (3) lack of support for distribution.

1.4 Organization of the Paper

The remaining sections develop the argument more fully in the following way. Section 2 is a detailed treatment of the obstacles to sharing in process-based systems, and the costs of some popular software workarounds. Section 3 outlines the basic abstractions of Opal, explains how Opal's uniform-address model eliminates the sharing problems of process-based systems, and suggests some examples of how direct sharing can be used effectively by software. In Section 4 we answer some common objections to uniform addressing. That section focuses on the apparent benefits of context-dependent addressing, which are sacrificed in Opal and other uniform-address systems. Section 5 summarizes our conclusions.

2 Sharing in Process-Based Systems

Most operating systems in use today are based on the traditional model of *processes* executing in *private virtual address spaces*. The process model evolved at a time when address bits were in short supply and programs were assumed to be independent and competing for resources. The purpose of the operating system was seen as arranging for equitable sharing of *physical* resources, while isolating each executing program from its competitors. In this environment private address spaces serve three purposes:

- *Contiguity.* Each process sees an abstraction of a dedicated machine with a contiguous linear address space.
- *Isolation.* An executing program cannot even *name* data that does not belong to it, since all names are interpreted in the context of its private address map.
- *Reclamation.* Storage reclamation is trivial from the perspective of the operating system; when a process terminates, its storage is reclaimed. No pointers to the data can exist outside of the terminating process.

Despite these apparent benefits, the multiple name spaces of process-based systems present inherent obstacles to data sharing and cooperation between programs. The problem is that a stored pointer has no meaning beyond the boundaries of the process that stored it; information containing pointers is not easily shared because the pointers may be interpreted differently by each process that uses the data. Data structures are easily shared by threads in the same process, but not across processes.

In practice, sharing of data structures can be achieved in process-based systems with some additional software overhead. A process can address shared data if it first *imports* that data into its private naming context. To do this, it allocates a range from its virtual address space and attaches the data to the range in some way. A translation procedure then converts pointer references in the shared data to virtual addresses that are meaningful in the process naming context. Broadly, this importing and translation of data can be done in two ways. One is to make a private copy of the data and translate the pointers in the copy, leaving the original undisturbed. The other is to map a single copy of the data into any process that uses it and interpret pointers in software each time they are dereferenced. This section highlights the weaknesses of several common variations of these approaches. For the moment we assume that the private address spaces of processes are in fact managed independently. In the presence of sharing this is frequently not the case, as discussed in Section 2.4

2.1 Copy-Time Translation

The standard technique for sharing data structures in process-based systems is to recursively linearize them for exchange through a file, pipe, or message [Herlihy & Liskov 82]. This technique was originally developed for gathering data into a message buffer (*marshaling*), but it has also been used to represent data structures on secondary storage (*pickling*). Pickles have been used effectively for self-contained structures that are used in their entirety, e.g., a program reading a small private data base on startup [Birrell et al. 87].

Linearizing has serious limitations as a general-purpose mechanism for translating data structures from one address space to another. Linearized data must be packed by the sender and unpacked by the receiver; both must traverse and copy the entire structure. Transmitted data is copied from as few as two times (for optimized local RPC implementations [Bershad et al. 90]) to as many as four times or more (for transmission through Unix pipes, files, messages, or sockets). Also, there is no way to represent a pointer to data outside of the package, thus it cannot be used to store or transmit a fragment of a larger structure, and a process cannot fetch pieces of a linearized data structure on demand.

2.2 Reference-Time Translation

One alternative to linearizing is to translate pointers in shared or copied data at reference time. The "pointers" in this case are not virtual addresses, but are encoded as pointer *surrogates* defined in some external naming context and interpreted by software, either the application itself or code emitted by a special-purpose compiler. The simplest scheme is to represent pointers as offsets into a file or memory segment. A variety of complex schemes for surrogates have been developed, using one or more levels of indirection through mapping tables. This approach is used by persistent object stores [Moss 89, Hornick & Zdonik 87] and commercial object-oriented database systems. The pointer indirection in these systems allows objects to be easily moved to a different part of the address space, to compact storage or to improve clustering properties.

Reference-time translation has a heavy runtime cost. Offset-based pointers can increase the cost of dereference by as much as 50%, not including the cost of saving and restoring segment base pointers.[2] For some object-based schemes the penalty is much higher. Depending on the locality and frequency of surrogate pointer dereferences, this can significantly impact execution time.

Another problem with reference-time translation of surrogates is that there are generally two forms for stored pointers, forcing the application or language processor to keep track of which form to expect on each dereference. A third pointer form may be needed to represent pointers to data outside of whatever naming context the surrogates are defined in. Examples include cross-segment pointers in a segmented architecture, or cross-database pointers in some persistent object stores [Cockshot et al. 84]. If there is no globally uniform name space for pointers, the reference may be qualified by a symbolic name (e.g., a file name). Symbolic names are designed for human consumption, and are inappropriate for use as pointers. They need special interpretation that is (1) slow, (2) nonuniform, and (3) possibly incorrect, since a human may change the binding, being unaware that a stored reference exists.

2.3 Swizzling

Some systems that use surrogate pointers attempt to reduce the cost of interpreting pointers by caching the result of a translation, overwriting the surrogate in place with the virtual address. This technique, called *swizzling*, is based on the assumption that the cost of the translation can be amortized across multiple dereferences of the same pointer. Swizzling is usually managed by a runtime package. It must be integrated with the programming language implementation to locate stored pointers in the data.

One problem with swizzling is that it forces each process using the data to make a private copy, since the swizzled pointers are specific to a particular address space. In most systems, this means that each process will allocate virtual backing store even for data that is never modified. Also, if multiple programs have tight interaction through the shared data (say, a design editor and a design checker in a CAD environment) then swizzling will incur runtime overhead to keep the copies consistent.

Swizzling has other problems inherited from its ancestor, overlays. First, swizzled pointers must be "unswizzled" back into surrogates before data is passed outside of the process boundaries (e.g., for coherency traffic, or committing updates to a database). This further increases runtime overhead, and the implementation must know how to translate pointers in both directions. The unswizzling is typically done in yet another copy of the data to avoid unswizzling the pointers in place and losing any additional benefit from the effort expended to swizzle them. Second, virtual pages cannot be reassigned to new data without some means of ferreting out stale swizzled pointers to the data that previously occupied those virtual pages; when a virtual page is reassigned, every swizzled pointer must be revalidated before it can be used again.

It has recently been suggested that a swizzling paging system could be used to support programs that demand more than 32 bits of address space, as a software alternative to wide-address hardware. Wilson [Wilson 91] proposes a system that uses wide pointers in a disk representation of the data, swizzled to

[2] The dereference sequence needs an **add** instruction as well as the usual two loads from memory. Note that offset-based pointers are different from the **(register + displacement)** addressing universally used in RISC architectures. The difference is that with offset-based pointers the displacements are not known statically.

32-bit pointers at page fault time so that only the narrow pointers appear in memory. One problem with this approach, sometimes called *eager swizzling*, is that it translates *all* imported pointers, even those that are never used. Eager swizzling is also complicated by the need to swizzle pointers whose targets have not been imported and therefore do not yet have virtual addresses assigned. For this reason, Wilson's approach allocates virtual pages not just for imported *data* but also for the *targets* of all imported pointers. This increases the risk that virtual memory will be filled, forcing an expensive reassignment.

Swizzling was designed to allow a single process to access stored data structures with restricted usage patterns. It works well if the process can establish a working set in memory and operate on it independently. It is less effective in cases involving concurrent sharing or significant I/O activity.

2.4 Summary

In process-based systems, sharing of data structures between processes requires pointer translation from one name space to another, at import time, at reference time, or on first use. There are three problems common to these translation schemes: (1) they are complex, and require support from the language interpreter and/or the application itself, (2) they restrict the scope of pointers in shared data, and (3) translation and copying have a significant runtime cost. The translation cost may be reduced by copying the data and translating the pointers in place, but this wastes storage and incurs copying and coherency overhead. In the next section we argue that these problems can be avoided in a uniform-address system by using direct virtual addresses as pointers in shared data.

We recognize that a direct virtual address may not be the best way to represent a reference, particularly if late binding of the reference is needed for some reason. The intent of this section is to show that translation schemes have a cost that is too high to pay when late binding of references is not needed. Furthermore, we claim that late binding can always be achieved with one or more levels of pointer indirection, that the underlying representation of a reference is always in terms of virtual addresses, and that these more complex schemes can also benefit from uniform addressing.

We have assumed until now that the private address spaces of processes are managed independently, as is generally the case. Processes sharing data can avoid the cost and complexity of translation by coordinating their address space usage so that the sharing partners attach each shared page to the same virtual page, a reserved area of overlap in their otherwise independent name spaces. Pointers within shared pages can then be represented as unencoded virtual addresses. For example, virtual addresses can be saved directly in a mapped file *if* that file is always mapped at the same address by any program that uses it; this is the approach used by Camelot [Eppinger 89] to support databases with pointers in them.

The problem with this solution is that the overlapping address ranges are negotiated *ad hoc*, so naming conflicts will occur if sharing patterns are not known statically. Furthermore, the mix of shared and private regions introduces dangerous ambiguity. If process **A** stores a pointer to private data into an area shared with process **B**, not only is it impossible for **B** to arrange access to the target, but it cannot even detect the error – it simply interprets the pointer incorrectly. We believe that this approach is basically correct, but that the *system* rather than the *applications* should coordinate the address bindings, in order to accommodate dynamic sharing patterns in a uniform way. This is the essence of the uniform-address model.

3 Sharing in a Uniform-Address System

The fundamental problem with the process-based model is that it defines a fixed relationship between naming and protection. The model provides no means to create a protection domain without creating a new name space, so interacting software components cannot be placed in separate protection domains without introducing nonuniform naming of shared data. Furthermore, executable code typically is linked to assume that it will execute in a private name space; since there is also no way to create a new name space without creating a new protection domain, it is impossible to execute a program image without creating a new domain for it to run in.

Uniform-address systems avoid these problems because they view naming and protection as independent. Sharing is simpler and more efficient than in process-based systems, because separate protection domains can address the same physical storage with the same names.

In this section we illustrate these points by discussing some features of Opal, a uniform-address system that we are designing. The discussion focuses on Opal's memory addressing model, particularly the notion of *virtual segments*, the relationship between segments and protection domains, uniform addressing of persistent storage, and the treatment of code modules. We avoid other aspects of the system, such as execution structures, cross-domain control transfers (RPC), and storage management.

3.1 Protection Domains and Virtual Segments

An Opal protection domain is an execution context that controls access to pages in a global virtual address space. A domain identifier is part of the register state of every executing processor, and a processor may switch from one domain to another with a privileged instruction that loads a new domain identifier from memory.[3] Each protection domain has some (possibly empty) subset of {*read, write, execute*} permission for each virtual page. Multiple processors may be executing concurrently in the same domain. Any protection domain may create a new protection domain, attach code and data to it, and start threads in it. Threads in the same domain have the same protection identity.

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. Opal segments are similar to segments in Multics and other segmented systems. The key difference is that each Opal segment is named by a virtual address range that is assigned when the segment is created and fixed until it is destroyed. An Opal segment cannot grow beyond the address range allotted to it when it is created. The address ranges assigned to different segments are always disjoint, and all memory references use a fully qualified flat virtual address. This means that stored pointers in Opal have a globally unique interpretation, 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.

Before a domain can complete a memory reference to a segment, it must establish access to it with an explicit *attach* operation. 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. One way to resolve a segment fault is to dynamically attach to the segment. This might be useful if the application is navigating a pointer graph spanning a large number of segments. In simpler cases, when an application can anticipate what storage resources it needs, segments can be attached eagerly. In this case, segment faults are used to trap and report references to data outside of the application's established set of storage resources; they serve as a first line of defense against unexpected behavior or costly resource searches caused by a stray memory reference.

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. The policies for managing persistence and recovery vary according to a segment type, and are beyond the scope of this paper. Reclamation is discussed in Section 4.2.

Persistent virtual memory can be viewed as a replacement for a traditional filesystem. Recoverable segments share many of the characteristics of mapped files. The operating system faults pages into memory transparently on reference, eliminating the programming cost and runtime overhead of copying read and

[3] If you think of a protection domain as corresponding to a value for the "address space identifier" (ASID) register in the MIPS architecture, you won't go wrong. Translation buffer entries are tagged with the contents of this register when they are created; a **load** or **store** cannot hit a TB entry unless the entry's tag matches the current ASID in the processor. Despite its name, the ASID has little to do with address spaces.

write calls. Also, the integration of file cache management with the virtual memory system avoids two-level caching problems, such as double paging and unnecessary consumption of swap space [Stonebraker 81].

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.[4] 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, to be implemented by one or more name servers that associate symbolic names with arbitrary pointers.

3.3 Code Modules

Opal stores executable code in persistent segments called *modules* that each contain a group of related procedures. Modules are independent of protection domains; a module can be attached by multiple domains, and the domain of a thread is in no way determined by the module that the thread is executing. Modules are also decoupled from the data they operate on; a given module might be called by different domains to operate on data in many different segments. These properties distinguish Opal modules from the notion of a *realm* in Psyche.

A *pure module* is a module that: (1) contains no self-modifying code, and (2) makes no static assumptions about the load address of any data that it operates on, unless that data is to be shared by all users of the module. In general, we expect that modules will be pure (see Section 4.3). Pure modules in Opal are statically linked into the global address space. Globally linked modules can be attached to any protection domain without risk of name conflicts in the code references, because each module resides at a fixed address, the same address in any domain.

There are two distinct benefits from global linking of code modules. First, domains can dynamically load modules 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 is driven by ordinary page faults. Opal can avoid dynamic linking for pure modules because its address space includes persistent memory; in uniform-address systems without a single-level store all modules must be dynamically linked.

The second benefit of global linking is that pure modules may be freely shared between domains. Within a node, only one copy of each pure module is needed, on disk and in memory, even if many programs or domains are executing it. Module-grain code sharing can substantially reduce storage demands and runtime load latency, and it is increasingly important as: (1) runtime packages implementing high-level programming abstractions (e.g., for graphics and user-interface support) continue to grow in size and popularity, and (2) traditional operating system kernel functions (e.g., thread management [Anderson et al. 91a]) migrate into runtime packages.

Of course, globally linked modules must be relinked if they contain references to some other module that has been modified. However, modules that have large numbers of references to them tend to be standard runtime libraries that are modified relatively rarely. A level of indirection (jump tables) can be used for references to frequently modified modules (e.g., during debugging).

Pure modules by definition make no assumptions about what other modules are loaded with them in a domain. There is no distinction between a program and a module. Any accessible code module can be invoked with or without creating a fresh domain for it to run in. Either the caller or callee may insist on a new domain to protect itself from the other party; if neither cares, there is no need to create a protection domain.

[4] Access to persistent memory in Opal is fully transparent, though explicit operations may be used to commit or flush modified data in segments with strong failure recovery semantics.

3.4 Using Shared Memory

Uniform addressing introduces new possibilities for organizing software systems, using variations on the general theme of shared memory. Shared memory can improve performance by reducing the need for explicit cross-domain communication. 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. Components can limit their interaction by restricting the scope of the shared region and the way it is used. Here are some interesting variations, assuming different degrees of isolation and trust:

- *Page Passing.* A segment is shared sequentially, by detaching it from the sender and attaching it to the receiver. The receiver can validate the data before use.
- *Enforced Locking.* Access to a concurrently shared segment is "sequentialized" through locking. It is most efficient to rely on voluntary compliance with locking protocols implemented in the runtime system, but the system can also enforce locking at a coarse granularity by arbitrating page access permissions within the segment. A similar scheme is described in [Chang & Mergen 88].
- *Producer/Consumer.* Data is passed through a segment that is writable by the producer but readonly to the consumer. The producer is fully isolated, but the consumer trusts the producer to comply with usage and synchronization conventions.
- *Argument/Result.* A parent domain shares a segment with its child. The parent leaves arbitrary data structures in the shared segment as input to the child, and collects the child's output from the shared segment after the child domain is destroyed.
- *Open Server.* Client domains have readonly access to a shared segment that is infrequently modified. Clients update the shared segment with protected calls to a privileged server domain with exclusive write access to the segment. The clients trust the server to maintain the integrity of the structure.
- *Deities.* A privileged domain has direct access to segments attached to client domains that rely on it for some service, e.g., supporting some aspect of the programming environment. The privileged component might be a debugger, an external paging server [Young et al. 87], or a garbage collector or transaction manager for data structures that span client domains.
- *Soft Firewalls.* A single application is structured as a group of partially trusting domains that concurrently share some but not all of their segments, to strike some useful balance between isolation and performance. This assumes that there are some clean component boundaries within the program.

These uses of sharing could be largely transparent to properly written application code. Shared data can be hidden behind a procedural abstraction just as if it were private to a separate server; cooperating domains would generally use the same module to operate on a given piece of shared data. The module must be reentrant, but it can be independent of whether or not the calling threads are executing from separate protection domains.

4 The Costs of Uniform Addressing

Up to this point we have argued the case for context-independent virtual addressing of data with no mention of the costs, if any, other than the need to pay for wide-address hardware. Our argument has been that a global addressing context leads to an unambiguous and consistent interpretation of addresses, facilitating sharing. Also, by enforcing a single mapping between virtual and physical addresses on each processor, *aliasing* and *homonyms* are eliminated, allowing trouble-free use of a virtually addressed data cache.

In this section we explore the benefits of *private* address spaces that are sacrificed to uniform sharing in Opal and other single address space systems. Section 2 listed some of these apparent benefits, all of which stem from context-dependent addressing in process-based systems. The question we wish to answer is: what will we lose if we build our operating systems with fully context-independent addressing?

Broadly, context-dependent naming has two useful properties [Saltzer 78].

- *Free choice of names.* Within a context, names can be assigned freely without danger of conflict with names used by other contexts (assuming no sharing occurs).
- *Retargetable name bindings.* It is sometimes useful for users to deliberately assign different interpretations to the same name.

This section discusses examples of how private address space systems take advantage of these properties, focusing on the lowest level of naming, the virtual addressing level. The goal is to defuse potential objections to uniform addressing by identifying specific claimed benefits of *context-dependent* addressing, and arguing one of the following for each one: (1) the context-dependent addressing is not useful, (2) context-dependent addressing *might* be useful, but an acceptable or better alternative exists, or (3) context-dependent *naming* is useful, but virtual addresses are the *wrong level* to apply the context-dependence. Our conclusion will be that all context-dependence should appear at higher levels of naming interpreted by software (e.g., relative to the registers of a particular thread) and mapped into a single context-independent virtual address space interpreted by the machine.

4.1 Free Choice of Addresses

In a uniform address space, programs are not fully free to select addresses for the data they create, nor can they statically determine what addresses will be chosen by the system. From the perspective of ordinary user-level code, the effect is that the address space appears to be sparse and noncontiguous, not the neat package familiar from process-based systems.

The first concern is that indexed memory structures (e.g. objects or **structs**, arrays, and linear code sequences) *must* be stored so that they are virtually contiguous. Our premise is that the system can always find a virtual address range that is large enough to store an indexed structure contiguously, assuming that the structure does not grow beyond some reasonable maximum size known in advance. Consider that a contiguous ten terabyte chunk consumes less than one millionth of a 64-bit address space. We anticipate that there will be no problem with utilization or fragmentation even if the address space spans a network of up to 10,000 nodes. A 10,000 node 64-bit address space would provide 1600 terabytes of name space per node, plenty of room to allow for huge storage volumes or aggressive preallocation of the address space.

Although indexed structures must be allocated contiguously, pointers and branches between those structures have no dependency on their relative positions in the address space. Thus the loss of complete contiguity is generally invisible to programs written in high-level languages. However, heap management packages must tolerate noncontiguous expansion of the heap.

4.2 Reclamation

A related question is the issue of reclaiming address space. In process-based systems each protection domain has a private naming context, so all names in the context are trivially reclaimed when the domain is destroyed (i.e., the program exits). In a global addressing context, data may be referenced by any domain, so its lifetime is no longer tied to the lifetime of any particular domain.

In a single address space, reclamation is only as complicated as the sharing patterns. If no sharing takes place, as in process-based systems, then reclamation is no more difficult than it was before: the system reclaims address space simply by remembering which address ranges were allocated to which domains. Thus the reclamation issue should *not* be viewed as a weakness of the single address space approach, but rather as a complication that stems from its *strength* – support for arbitrary sharing patterns.

In Opal, the operating system does not track cross-segment references. It merely provides hooks to allow servers and runtime packages that manage shared segments to plug in their own reclamation policies. Our prototype implements one possible segment server that 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 responsibility of code in the clients (the runtime package and/or the language implementation) to use those 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.

4.3 Private Static Data References in Shared Code

Uniform addressing leads us to restrict the use of certain addressing modes in shared code. In Section 3.3 we introduced the notion of a *pure* code module. A pure module is restricted in the modes that it can use to address its *private* data. Data is private if it has separate instantiations for each caller of the module. We are mostly concerned about references to private *static* data that is not allocated from a heap or thread stack, specifically: (1) global variables declared in the module, and (2) linkage tables used for late binding of external symbols referenced by the module. As an example of the second case, consider that different users of a module may want to use different implementations of an interface imported by the module, e.g., one user may want a special profiling version of the math library.

In process-based systems, private static data is typically addressed relative to the program counter, or with absolute virtual addresses statically linked into the module. These **pc-relative** and **absolute** references are efficient and easy for language processors to implement. They resolve correctly even if the code is shared, because each instantiation runs in a separate domain with a private address space. But in a uniform-address system, code that names private data using these modes is *impure*; impure code is not prohibited in Opal, but it has certain performance costs. First, impure code cannot be directly shared, because private data references using those modes in shared code will always resolve to the same piece of data, which is not what was intended. Second, code using **absolute** addresses for private data must also be dynamically relocated, because each copy must be loaded at a different virtual address.

To make modules pure, all private data references must be expressed as an offset from a base register whose value is passed in by the caller of the module. Indexing from a base register is the norm on some architectures (e.g., MIPS), and generally has no additional cost, assuming the offsets are statically known. However, the difficulty of managing base pointers depends on the sharing granularity for code modules. Many process-based systems, such as BSD Unix, allow only a single shared module (a complete executable program) to be loaded into each process. If we were to make a similar restriction in Opal, there would be only one base pointer per protection domain, and the implementation would be trivial. But if we allow more than one shared module to be loaded into each protection domain, then each module must have its own base pointer, and each module must retain (in its private data) the base pointers for the modules that it calls. Base pointers must be saved and restored whenever control crosses module boundaries.

Our conclusion is that in order to allow fully general sharing of code modules in a uniform-address system, we must limit the use of **absolute** and **pc-relative** addressing, and support per-module base pointers. This is trivial in some languages (e.g., object-oriented languages) and more difficult in others (e.g., C). However, once we have paid this cost, fully general sharing and dynamic loading of code modules are achieved very easily, without dynamic linking. In process-based systems, comparable support for shared libraries involves the same restrictions on **absolute** addressing that we have made, and even with those restrictions, requires dynamic linking and/or indirect addressing through linkage tables (e.g., for cross-module branches) [Sabatella 90].

4.4 The Trouble With Fork

The Unix **fork** operation is defined to clone the parent process execution context, including its private address space, in the child process. The copy can be modified independently of the original, hence the cloning semantics of **fork** assume a private address space for each protection domain, and so cannot be emulated in uniform-address systems.

We view **fork** as a historical artifact with little value even in process-based systems. Previously there were two reasons to treat a child process as a cloned replica of the parent: (1) to create multiple asynchronous processes executing the *same program* (e.g., to exploit parallelism or to overlap I/O), and (2) to allow code from the parent to run in the context of the child, so that it can initialize that context (e.g. file descriptors) as a prelude to executing a new program in it. The use of **fork** for asynchrony has been superseded by the thread abstraction, which is better suited to the purpose. Also, microkernel systems supply primitives to initialize the execution context of a child domain without actually executing code in the child; this ability is a prerequisite to implementing **fork** in a user-level Unix server. Thus, the only reason to implement **fork** at all is to support existing Unix programs that use it. This is not one of our research goals, nor should it be a dominant concern for operating systems in the future (note that most Unix programs do not make direct use of **fork**).

We also note that **fork** is a source of complexity and inefficiency in Unix systems. The majority of the state cloned by a fork is not needed by the child, inspiring efforts to improve performance by cloning data lazily (e.g., copy-on-write) or avoiding the copy altogether (e.g., the **vfork** primitive in 4.2 BSD). The cloning semantics of **fork** also interfere with support for threads [McJones & Swart 87].

Opal replaces **fork** with primitives to create and destroy protection domains, and initialize them by attaching segments and installing RPC endpoints. The parent domain starts a thread in the child by making a cross-domain call to it. We believe that these mechanisms are sufficiently general.

4.5 Copying and Grafting

Earlier we argued that many instances of data copying can be eliminated by defining a uniform address space. In particular, data can be shared without the naming problems that may force unnecessary copying. Obviously, not all instances of copying can be eliminated. Two closely related uses of copying remain:

- *Messages.* Some instances of message-passing could be reformulated as unmapping the message from the sender and remapping it in the receiver. But if both domains need write access to the pages, yet must be isolated from each other's changes, then the data must be copied. For example, data to be passed may reside on pages containing other data that must be kept.

- *Versions.* A piece of data may be copied to create a new piece of data that starts as a clone of the original but diverges from it in arbitrary ways.

When data is copied, pointers into the old copy must be translated if they are to point into the new copy. In particular, internal pointers within the copied data must be translated. Pointers are translated using one of the schemes described in Section 2.

In a process-based system, context-dependent addressing can sometimes be used to avoid translating pointers in data copied from one process to another. The trick is to place the copied data into the same virtual address range in the receiver process as it occupied in the sender. (Of course, it is then impossible for either process to name both the copy and the original.) For lack of a better term, let us call this *grafting*. Grafting is not possible in uniform-address systems; since there is only one naming context, the copy must occupy different virtual addresses from the original, and pointers must be handled specially.

Examples of grafting are rare in today's systems, although they do exist. It is sometimes used to make copies of statically linked self-modifying code modules, or copies of writable private data segments (private to shared code modules) that have been initialized with pointers. In practice, most copies of this type result from **fork**, which we have already dispensed with.

4.6 Copy-On-Write Optimizations

Uniform-address systems have one additional difficulty stemming from their inability to use the grafting optimization described above. A process-based system can lazily evaluate a grafting copy using copy-on-write, but a uniform-address system copying the same data could not use copy-on-write. This is because the pointers in the data must be translated before the receiver can even *read* the data; copy-on-reference could be used, but not copy-on-write.

We emphasize that grafting copies are the *only* case where a process-based system could use copy-on-write but a uniform-address system could not. It might appear that copy-on-write is incompatible with uniform addressing, because it introduces *aliasing* in the form of multiple virtual mappings to the same physical page. But in fact, copy-on-write introduces only a restricted and benign form of aliasing called *readonly aliasing* [Chao et al. 90]. Readonly aliasing is a hidden optimization that neither violates the consistent interpretation of pointers nor interferes with the use of a virtually addressed data cache, leaving intact the benefits we claim for uniform addressing. The problem introduced by more general forms of aliasing is that the aliases will have separate entries in a virtual cache, and the cache hardware must keep these duplicate entries consistent since they map to the same physical address. This problem does not occur with readonly aliasing: if a write to either virtual address occurs, a new physical address is assigned to one of them, destroying the alias and restoring a unique mapping. Thus the value of a cache line is never modified in a way that affects the value of another cache line. A more detailed discussion of the relationship between uniform addressing and virtual data caches can be found in [Koldinger et al. 91].

Because readonly aliasing is harmless, a uniform-address system can use copy-on-write in every instance that a process-based system can, except when pointers in the copied data must be translated. Most uses of copy-on-write (apart from **fork**) in existing systems are to optimize message-passing, particularly to pass large messages between I/O servers (e.g., file or memory servers) and their clients. In these cases the data either contains no pointers, or it is opaque to the server. (Of course, in this case, the server may not need to retain access to the data after it is passed, so it might be better to unmap it; this is discouraged by the process-based model because it creates a hole in the server's linear address space.)

4.7 Summary

We conclude that the loss of context-dependent addressing is generally painless. The only troubling use of context-dependent addressing is to simplify *copying*, as opposed to physical sharing, of data containing pointers. But this requires tricky coordination of address space usage ("grafting"), and we see no compelling examples of it.

It is possible to construct a hybrid of the uniform-address and process-based models that allows some uses of context-dependent addressing. In fact, our original proposal for global addressing suggested that a shared address space on 64-bit architectures should be implemented as a *policy* in microkernel operating systems, which make no assumptions about virtual address space usage by protection domains [Chase & Levy 91]. We are now pursuing the pure single address space model for two reasons. First, we believe that context-dependent addressing introduces a dangerous non-uniformity to the system. Once it is present, programs can no longer assume that a virtual address has the same meaning to another domain. Second, the policy approach complicates the memory system hardware as well as the operating system software. For example, cache lines for virtual addresses with multiple interpretations must be tagged.

5 Conclusions

We have presented a model for a *uniform-address operating system* called Opal. The key property of Opal is a single flat virtual address space encompassing all data in the system, across the network and on long-term storage. Multiple protection domains (similar to *processes* in today's systems) coexist in this common address space, providing a basis for virtual memory protection. Opal requires no special hardware support, other than a sufficient number of virtual address bits. We have evaluated the costs and benefits of this uniform-address model relative to process-based memory models.

The primary advantage of the uniform-address model over *process-based* systems is improved support for sharing of linked data structures between protection domains. We described obstacles to sharing inherent in the process-based model, and the runtime overheads of the data copying and pointer translation schemes used to circumvent those obstacles. In contrast to the process-based model, Opal allows programs running in separate protection domains to share or exchange data structures through shared memory regions, files (persistent memory), and messages, without additional software overheads. Opal code libraries can be dynamically loaded and physically shared in a fully general way, without runtime relinking. The notion of "executing a program" dissolves, since any procedure call can in principle be made to execute in a private protection domain at the caller's discretion; the input and output of the procedure/program can be arbitrary linked data structures.

The disadvantages of Opal relative to the process-based model are: (1) loss of the contiguous linear addressing environment for executing programs, (2) inability to support the Unix **fork** abstraction, and (3) inability to "graft" data structures from one address space to another. Also, by choosing 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 uniform-address system. However, we believe that existing approaches to reclamation (e.g., based on language-supported garbage collection) are applicable, and that the system can always prevent untrusting programs from harming each other with reclamation errors.

Opal's goals are similar to those of earlier capability-based systems, but there are crucial differences in the way those goals are achieved. Opal trades away the fine-grained object protection of capability systems in favor of a coarser but more efficient page-based protection model. This model can be implemented in software on "standard" 64-bit hardware, and can be extended to accommodate distribution.

To summarize, here are three ways of viewing the Opal project:

- *As another step in the trend of decoupling concepts that are traditionally combined in the notion of a "process".* The original conception of a process (in Unix or Multics) combined addressing, protection, execution, and communication – everything having to do with the relationship between an executing program and the rest of the system. Unifying these ideas in a single abstraction was intuitively appealing given the assumptions in practice at the time. Since that time, physical storage has abstracted to virtual memory, execution has abstracted to threads, and communication has abstracted to ports and RPC endpoints. We now propose that address spaces be separated from processes as well. An Opal domain is exactly a process stripped of everything but its protection identity.

- *As a means of combining the uniformity of capability-based systems with the performance and generality of process-based systems.* Capability-based computers were not widely accepted because of their complexity, performance, and restrictive object-oriented data model. Opal strives for the same uniform addressing of storage, but like process-based systems its addressing model is based on efficient untyped storage abstractions rather than objects.

- *As one approach to reconciling current software and architectural trends.* Operating systems and applications are increasingly decomposed into multiple cooperating protection domains. At the same time, the trend in high-performance computer architectures is to penalize this decomposition by increasing the relative cost of cooperation between domains. This is because cooperation in today's systems relies on data copying and cross-domain calls, which are increasing in cost relative to integer performance. Our approach encourages new mechanisms for cooperation based on shared memory, which will scale with integer performance on next-generation architectures.

We are currently prototyping the ideas described in this paper. In addition to serving as a short-term proof-of-concept, the prototype effort has two important long-term goals. First, we wish to explore the effect of a shared address space on existing operating system abstractions and mechanisms. For example, the presence of sharing affects strategies for thread management, processor allocation, and synchronization;

recent work in these areas has assumed that domains are self-contained, independent, and competing. A broader research goal is to explore ways in which the shared address space can be used to improve the structure and performance of applications and the operating system, by identifying uses of sharing that are safe and can be handled efficiently.

6 Acknowledgements

We would like to thank the many people in the operating systems community who have discussed these ideas with us, singling out the following people for special thanks: Mike Burrows, Carla Ellis, Norm Hutchinson, David Kotz, Richard LaRowe, Brian Marsh, Marc Shapiro, and John Wilkes. Many people at the University of Washington have made valuable contributions, especially David Keppel, Eric Koldinger, Kevin Sullivan, Chandu Thekkath, and John Zahorjan. Thanks also to Richard Korry and Robert Henry of Tera Computer.

References

[Anderson et al. 91a] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support of the user-level management of parallelism. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 95–109, October 1991.

[Anderson et al. 91b] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. 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.

[Bershad et al. 90] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1), February 1990.

[Birrell et al. 87] A. Birrell, M. Jones, and E. Wobber. A simple and efficient implementation for small databases. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 149–154, November 1987.

[Chang & Mergen 88] A. Chang and M. F. Mergen. 801 Storage: Architecture and Programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.

[Chao et al. 90] C. Chao, M. Mackey, and B. Sears. Mach on a virtually addressed cache architecture. In *Usenix Mach Workshop Proceedings*, pages 31–51, October 1990.

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

[Cockshot et al. 84] W. P. Cockshot, M. P. Atkinson, and K. J. Chisholm. Persistent object management system. *Software – Practice and Experience*, 14(1), January 1984.

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

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

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

[Groves & Oehler 90] R. D. Groves and R. Oehler. RISC system/6000 processor architecture. In M. Misra, editor, *IBM RISC System/6000 Technology*, pages 16–23. International Business Machines, 1990.

[Herlihy & Liskov 82] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[Hornick & Zdonik 87] M. F. Hornick and S. B. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Transactions on Office Information Systems*, 5(1), January 1987.

[Koldinger et al. 91] E. J. Koldinger, H. M. Levy, J. S. Chase, and S. J. Eggers. The protection lookaside buffer: Efficient protection for single-address space computers. Technical Report 91-11-05, University of Washington, Department of Computer Science and Engineering, November 1991.

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

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

[McJones & Swart 87] P. R. McJones and G. F. Swart. Evolving the Unix system interface to support multithreaded programs. Technical Report 21, DEC Systems Research Center, September 1987.

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

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

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

[Ousterhout 90] J. K. Ousterhout. 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] D. Redell, Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray, and S. Purcell. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[Sabatella 90] M. Sabatella. Issues in shared library design. In *Proceedings of the Usenix Conference*, pages 11–22, June 1990.

[Saltzer 78] J. H. Saltzer. Naming and binding of objects. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course (Lecture Notes in Computer Science 60)*, pages 99–208. Springer-Verlag, 1978.

[Scott et al. 90] M. L. Scott, T. J. LeBlanc, and B. D. Marsh. 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.

[Stonebraker 81] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

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

[Wilson 91] P. R. Wilson. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. *ACM SIGARCH Computer Architecture News*, 19(4), June 1991. University of Illinois at Chicago Technical Report UIC-EECS-90-6, December 1990.

[Young et al. 87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a mulitprocessor operating system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 63–76, November 1987.