Sharing and Protection in a Single-Address-Space Operating System

JEFFREY S. CHASE, HENRY M. LEVY, MICHAEL J. FEELEY,
and EDWARD D. LAZOWSKA
University of Washington

This article explores memory sharing and protection support in Opal, a single-address-space operating system designed for wide-address (64-bit) architectures. Opal threads execute within protection domains in a single shared virtual address space. Sharing is simplified, because addresses are context independent. There is no loss of protection, because addressability and access are independent; the right to access a segment is determined by the protection domain in which a thread executes. This model enables beneficial code- and data-sharing patterns that are currently prohibitive, due in part to the inherent restrictions of multiple address spaces, and in part to Unix programming style.

We have designed and implemented an Opal prototype using the Mach 3.0 microkernel as a base. Our implementation demonstrates how a single-address-space structure can be supported alongside of other environments on a modern microkernel operating system, using modern wide-address architectures. This article justifies the Opal model and its goals for sharing and protection, presents the system and its abstractions, describes the prototype implementation, and reports experience with integrated applications.

Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Other Architecture Styles—*capability architectures*; D.1 [**Software**]: Programming Techniques; D.3.3 [**Programming Languages**]: Language Constructs and Features—*modules*, *packages*; D.4.2 [**Operating Systems**]: Storage Management; D.4.4 [**Operating Systems**]: Communications Management; D.4.6 [**Operating Systems**]: Security and Protection—*access controls*; *information flow controls*; D.4.7 [**Operating Systems**]: Organization and Design; D.4.8 [**Operating Systems**]: Performance—*measurements*; E.1 [**Data**]: Data Structures; E.2 [**Data**]: Data Storage Representations

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: 64-bit architectures, capability-based systems, microkernel operating systems, object-oriented database systems, persistent storage, protection, single-address-space operating systems, wide-address architectures

This work is supported in part by the National Science Foundation (grants no. CCR-8907666, CDA-9123308, and CCR-9200832), the Washington Technology Center, Digital Equipment Corporation, Boeing Computer Services, Intel Corporation, Hewlett-Packard Corporation, and Apple Computer. H. Levy is supported in part by a Fulbright research award and by INRIA. J. Chase is supported by an Intel Foundation Graduate Fellowship.

Authors' address: Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195.

1. INTRODUCTION

The appearance of 64-bit address space architectures, such as the DEC Alpha [Digital Equipment 1992], HP PA-RISC [Lee 1989], and MIPS R4000 [MIPS Computer Systems 1991], signals a radical increase in the amount of address space available to operating systems and applications. This shift provides the opportunity to reexamine fundamental operating system structure—specifically, to change the way that operating systems use address space. Our goal is to restructure operating systems in order to improve the organization of both the system and its applications. In particular, we wish to enhance sharing, to simplify integration, and to improve the reliability and performance of complex, cooperating applications manipulating large persistent data structures. For example, our target application domain includes integrated software environments, such as engineering design (CAD or CASE), composed of groups of data-centered tools that are inherently interdependent and have rich interactions.

This article describes Opal, a *single-address-space* operating system intended to support these complex applications on wide-address architectures. Opal provides a single global virtual address space that is shared by all procedures and all data. Crucial to the design is the full separation of *addressing* and *protection*, which are intimately bound in the "process" concept of systems such as Unix and Multics [Daley and Dennis 1968].

The fundamental principle of the Opal system is that addresses have a unique interpretation, for all applications, for potentially all time. Virtual addresses are *context independent*: they resolve to the same data, independently of who uses them. While a thread can *name* all data in the system, it will generally not have the right to *access* all of that data; the *protection domain* in which that thread executes defines its access rights, limiting its access to a specific set of pages at a specific instant.

Wide-address architectures facilitate the single-address-space approach by eliminating the need to reuse addresses, which is required on 32-bit architectures. We do not wish to debate whether a particular address size is enough for what we propose here. A full 64-bit address space will last for 500 years if allocated at the rate of one gigabyte per second. We believe that 64 bits is enough "for all time" on a single computer, enough for a long time on a small network, and not enough for very long at all on the global network. But in any case, it is clear that address sizes have leaped past the 32-bit boundary and will continue to grow. In the past, this growth has averaged one additional address bit—a doubling of address space—every year [Siewiorek et al. 1982].

The purpose of the Opal experiment is to explore the strengths and weaknesses of the single-address-space approach, which is a significant departure from the traditional model of private, virtual address spaces for each executing program (e.g., Unix). The concepts in Opal are related to those of many previous hardware and software systems spanning over 25 years, for example, Multics [Daley and Dennis 1968], Hydra [Wulf et al. 1975], Pilot [Redell et al. 1980], Monads [Rosenberg and Abramson 1985], Intel 432

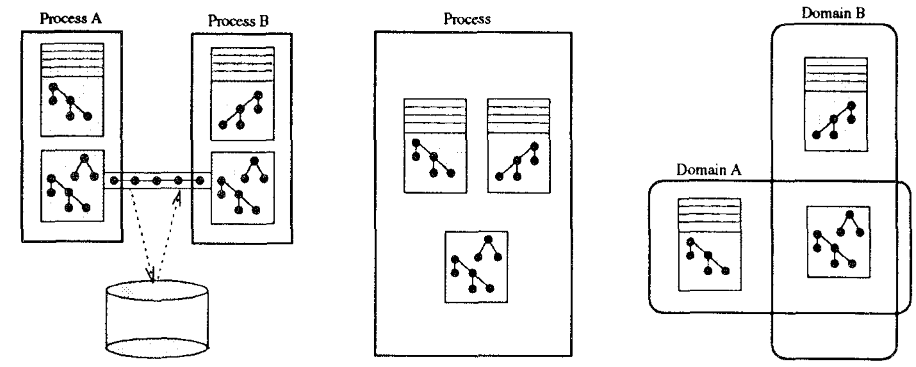
[Organick 1983], IBM System 38 [Houdek et al. 1981], Psyche [Scott et al. 1990], and the early work on capability systems [Dennis and Van Horn 1966]. A detailed discussion and comparison of these systems is provided in Section 7. Fundamentally, we hope to demonstrate that recent advances in hardware, operating systems, and language technology enable us to achieve the goals of many previous systems, but without the need for special-purpose hardware, without loss of protection or performance, and without requiring the single type-safe language that many of those systems demanded.

The following section introduces the basic premises of Opal's single-address-space structure and contrasts with the traditional private-address-space approach. We concentrate in particular on Opal's support for sharing and on the use of protection structures. Section 3 gives an overview of the basic Opal abstractions. Section 4 then describes our Mach-based prototype, which demonstrates that the single-address-space structure can be supported on existing microkernel platforms in a way that permits interaction with traditional Unix environments. Section 5 discusses the use of Opal by applications in our target domain, describes a mechanism we have implemented on Opal to manage sharing and protection, and provides some performance measurements of our prototype. Section 6 explores some objections to our approach, and Section 7 describes the relationship to previous systems. We summarize our experience and conclude in Section 8.

2. THE SINGLE-ADDRESS-SPACE APPROACH

Before examining the concepts of a single-address-space operating system, it is useful to review the multiple-address-space approach that we now take for granted. The major advantages of private address spaces are: (1) they increase the amount of address space available to all programs, (2) they provide hard memory protection boundaries, and (3) they permit easy cleanup when a program exits. The disadvantage of this approach, however, is that the mechanism for memory protection—isolating programs within private virtual address spaces—presents obstacles to efficient cooperation between protected application components. In particular, pointers have no meaning beyond the boundary or lifetime of the process that creates them; therefore pointer-based information is not easily shared, stored, or transmitted. The primary cooperation mechanisms rely on copying data between private virtual memories, typically converting it to and from a neutral intermediate representation. This is inconvenient and expensive for large or sparsely used data structures.

Private-address-space systems force poor tradeoffs between protection, performance, and integration. An application designer has two basic choices: place application components in independent processes that exchange data through pipes, files, or messages (Figure 1(a)), thereby sacrificing performance, or place all components in one process, sacrificing protection (Figure 1(b)). Neither choice is adequate for a growing and important class of applications that are composed of groups of programs cooperating through a shared pointer-rich database. This results in software systems that are slow, unreliable, or poorly integrated. These applications need better control of protection



(a) Copying linearized data through a pipe or file incurs costs to copy and store redundant data. Incremental updates are difficult.

(b) Placing multiple components in the same process avoids copying but compromises protection.

(c) In Opal, components can be placed in overlapping protection domains with a shared segment of memory.

Fig. 1. Three choices for structuring cooperation between application components.

and sharing than current systems can provide: protection is crucial because the programs are independently developed and evolving, and yet there are natural sharing relationships, e.g., the output of one program is often used as the input of another.

We believe that this dilemma can be resolved with judicious sharing of virtual memory between protected components. Sharing need not compromise modularity, and can increase performance substantially without sacrificing protection (e.g., through read-only sharing). While most modern systems support shared memory in some form, there are pitfalls and limitations to its use within the private-address-space model. Use of pointers typically requires a priori coordination of address space for shared regions, thus sharing patterns must be known statically. In a single-address-space model, the *system* rather than the *applications* coordinates the address bindings, to accommodate dynamic sharing patterns in a uniform way. Recent systems have taken steps in this direction; examples include *based sections* In Microsoft Windows/NT [Custer 1993] and the *mmap* facilities in recent Unix systems. Even in these systems, however, the mix of shared and private regions introduces several problems. Pointers may still be ambiguous: private data pointers in shared regions are difficult to detect or handle, and private code pointers (important for object-oriented languages) cannot be passed or shared. Early decisions must be made about what can be shared and what is private. Sharing is ad hoc, since any process that will *ever* use a shared region must have the same virtual addresses available.

Single-address-space systems avoid these problems by treating *all* virtual address space as a global resource controlled by the operating system, like disk space or physical memory. This can be done without sacrificing the major advantages of private address spaces. Specifically, (1) with wide-address architectures, we can now provide sufficient addressing without multiple address spaces, (2) there need be no loss of protection in the single address space, and (3) cleanup is no harder in a single address space for "conventional" programs that do not share data. (Alternatively, cleanup is no easier in private address spaces for programs that *do* share data.) In fact, many problems with the use of sharing and protection, including reclamation, can be simplified or eliminated by decoupling protection domains from other concepts that are bundled in the notion of a "process" on conventional systems. In addition to separating protection from addressing, Opal also separates program execution, resource ownership, and resource naming from protection domains. The purpose of this decoupling is to make memory protection cheaper, easier to use, and easier to change.

The independence of protection and addressing in Opal substantially increases flexibility. Programs can directly share procedures and complex pointer-based data structures, without requiring a priori negotiation of address space usage, as depicted in Figure 1(c). The common address space allows *dynamic* imports and binding to data structures and code. There is no need to decide in advance what is shared and what is private: any memory segment can be shared at any time without address conflicts, and memory access rights are easily passed on-the-fly from domain to domain. Opal has no conventional "programs": all code exists as procedures residing in the shared address space. Any procedure can be an entry point for a protection domain: subprograms can execute with or without a private protection domain, depending on the trust relationship between the caller and callee.

Furthermore, the single-address-space structure has properties that can be exploited by the memory system implementation. Specifically, address overloading is eliminated. This removes ambiguity in virtual tags in processor caches, as well as the need to maintain separate tables of virtual-physical translations for each process [Koldinger et al. 1992]. On processors equipped with software-loaded TLBs, it permits the use of alternative translation table structures (e.g., Huck and Hays [1993]) that better accommodate large, sparse virtual memories. (On the negative side, this property prohibits programs from mapping different data at the same virtual addresses. The effect of this restriction is discussed in Section 6.4.)

2.1 Sharing and Trust

Today's dominant protection model promotes protection domains (e.g., processes) encapsulating fully isolated software components and their data, interacting only through messages. This strict model of fully disjoint protection accommodates distribution, is tempting in its simplicity, and is central to both "server-structured" [Young et al. 1987; Rozier et al. 1988; Mullender and Tanenbaum 1986; Custer 1993] and "object-oriented" [Allchin and McKendry 1983; Almes et al. 1985] systems. However, we believe it is too simplistic and confining for several reasons:

— Asymmetric trust relationships are common and can be exploited: *A* might accept inputs (or memory segments) from *B* even when *B* does not trust *A*. For example, a name server could provide read-only access to it database, requiring protected messages only for updates.

— Direct sharing is useful even between mutually suspicious threads. Memory can be shared read-only, used in restricted ways, or passed sequentially from domain to domain.

— Protection domains can be used to coordinate data access among mutually trusting threads. For example, multiple instances of a database program may prefer to execute in separate domains to enforce different access privileges, or to use protection faults to drive implicit locking [Chang and Mergen 1988].

— Tradeoffs between protection and performance are unavoidable. Complete isolation can never be achieved: even if protection is fully disjoint, granularity tradeoffs still must be made. Also, programs may have naturally overlapping access to stored data, and this may not be known in advance.

Fundamentally, we believe (as do others [Druschel et al. 1992b]) that operating system protection structures are not the right level to impose modularity. In fact, protection structures do not impose modularity; they only *enforce* selected module boundaries. In any case, shared data should be accessed through procedural interfaces, and protection structures should be flexible enough to permit application-specific choice of how modularity is enforced.

2.2 Persistence and Distribution

The single-address-space structure can accommodate address spaces of different temporal or geographical scopes; that is, a single-address-space system on one node can be extended to include network-wide data and persistent data not in active use. This is attractive for applications manipulating pointer-rich data structures, which often need to store structures or share them across the network. In a distributed, persistent, single-address-space system, network nodes can exchange addresses directly through messages, and data structures can be directly saved on long-term storage and later accessed by other programs without the need to translate internal pointers. One of our goals is to extend the addressing domain across a small workstation cluster; this is discussed in more detail in Section 4.6.

We note that the purpose of including inactive and distributed data in the virtual address space is *not* to eliminate entirely the mechanisms for data conversion (i.e., swizzling, marshaling, and translation), but rather to reduce the *frequency* with which those mechanisms must be applied. A distributed persistent address space can be used to eliminate translation for the most common cases of transfers: between programs on a single node, between memory and long-term storage, and between nodes within a small LAN cluster. This amounts to *caching* a precomputed machine-dependent representation of the data, while retaining the ability to convert back to a machine-independent representation in exceptional cases, e.g., transmission outside of the local addressing domain. Programming-in-the-large of pointer-based applications demands some language-level knowledge (e.g., compiler-generated templates) of the structure of the data, in order to support garbage collection and integrity checks. Such structures have been used (e.g., Wilson [1991])—and can continue to be used—transparently to convert the format of data as needed when crossing boundaries outside of the single address space, whatever its scope.

This article focuses on the implications of the single-address-space structure on sharing and protection on a single node, and thus we will not discuss persistence and distribution in much detail. We simply note that the choice of a single address space for a node does not strictly require that persistent and distributed data be included within the local address space; however, the ability to accommodate these extensions naturally and uniformly is one of the model's strengths.

2.3 Summary

We believe that the familiar model of programs as independent short-lived processes that transform a stream of input to a stream of output is needlessly restrictive and forces poor structuring and performance tradeoffs for a broad and increasingly important class of applications. We believe that these applications are better served by the single-address-space structure, which is enabled by the appearance of 64-bit addressing on modern RISC processors. The objective of the single-address-space operating system is to expand the choices in the structuring of computations, the use of protection, and the sharing, storage, and communication of data.

3. OPAL ABSTRACTIONS AND MECHANISMS

This section defines the fundamental Opal mechanisms used for the management of the single address space and provides insight into our design choices. We limit our focus to aspects of the system that are essential to the single-address-space structure, or to our goal of supporting modular sharing and protection. The basic concepts should seem familiar to many and have appeared previously in various contexts. The key is the simple application of these concepts once you accept the separation of protection and addressing on wide-address architectures.

The system facilities described in this section need not be exposed directly to applications; they are intended as a substrate for building language and runtime environments, such as the *mediators* framework described in Section 5.2. Our Opal prototype includes a standard runtime package that defines a simple C^{++} programming interface. This package is used throughout our prototype, but alternative languages and application environments could be implemented within our framework. The runtime package and other aspects of the prototype are described in more detail in Section 4.

3.1 Storage and Protection

The Opal units of storage allocation and protection are *segments*, which are contiguous extents of virtual pages. The virtual address of a segment is a permanent attribute, fixed by the system at allocation time. The smallest possible segment is one page, but in general we expect segments to be large to allow growth of the structures they contain. Segments can be marked as *persistent* and managed through explicit reference counting as described in Section 3.6.

The Opal units of execution are *threads.* A *protection domain* is an execution context for threads, restricting their access to a specific set of segments at a particular instant in time. Each thread executes in exactly one protection domain, but many threads may execute in the same domain. Domains are the *subjects* of memory access control. An Opal domain is the analogue of a Unix process, except that domains are not private virtual address spaces, but rather are passive protection contexts within a global virtual address space. Alternatively, the "protection domain" could be viewed as the collection of segments accessible to such a protection context.

Our philosophy in Opal is that storage allocation, protection, and reclamation should be *coarse grained* at the operating system level. Fine-grained control is best provided at the language level by compilers and runtime systems. For example, our standard runtime package allocates large segments in which it provides heap storage for dynamic memory allocation. Programs can allocate objects from multiple heaps to control how data structures are partitioned across segments. We believe that burdening the operating system (or worse, the hardware) with fine-grained protection is an error, particularly given the safety that can be guaranteed by strongly typed languages. However, even with safe languages, the operating system must still support hard protection boundaries in order to separate nontrusting parties and different safe or unsafe language environments. Such hard boundaries, provided in Opal as protection domains, can easily be supported through standard page-based protection mechanisms on modern processors.

3.2 Access Control

All Opal kernel resources, such as protection domains and segments, are named by *capabilities*; a capability is a 256-bit reference that confers permission to operate on the named object in specific ways. A name service supports symbolic names for capabilities, with access control lists (ACLs) for protection. Opal uses *password capabilities* [Anderson et al. 1986], similar to those in Amoeba [Mullender and Tanenbaum 1986] and Chorus [Rozier et al. 1988], rather than Mach-style capabilities (also called *port rights*) that are maintained by the kernel [Young et al. 1987]. The advantage is that password capabilities can be passed directly in shared memory and used to name global resources; capabilities in Mach are meaningful only within a Mach protection context (*task*), preventing this sharing.

Given a segment capability, an executing thread can explicitly *attach* that segment to its protection domain, permitting threads executing within that domain to access the segment directly. Conversely, a thread can explicitly *detach* a segment to deny access. An *Attach* request can specify particular access rights to the segment (e.g., read-only or read-write), but cannot specify more rights than are permitted by the capability. *Attach* is the Opal analogue of the Unix *mmap* primitive for mapping files into a process, except that in Opal the system, rather than the application, always chooses the mapped address. Also, *all* data in Opal resides in segments that are potentially attachable, given the proper capability; there is no data that is inherently private to a particular executing program.

Segments can also be attached transparently on address faults, if the application chooses. The holder of a segment capability may *publish* that capability, along with an ACL.[1] When an address fault occurs, a runtime fault handler in the domain requests the published capability, specifying the faulting address. If access is granted (based on the ACL) then the handler attaches the segment before returning from the address exception.

3.3 Interdomain Communication

In Opal, shared memory is the primary form of sharing and communication between threads in different protection domains. Additionally, the system must permit control transfers from one domain to another. To support this, an Opal domain can create one or more *portals* that permit other domains to call it in a protected and controlled manner. Portals can be used to implement servers or protected objects.

A *portal* is an entry point to a domain, uniquely identified by a 64-bit value (the *portalID*). Any thread that knows the value of a portalID can make a system call that transfers control into the domain associated with the portal. Threads entering through a portal begin executing at a global virtual address that is a fixed attribute of the portal, specified by its creator. In this way the creator of a protection domain can control what code is executed within it.[2]

The global name space for portals in Opal allows the exchange of cross-domain call bindings through shared memory; password capabilities can then be implemented as a simple bind-time extension to a general-purpose RPC facility. An Opal capability is simply a 256-bit value containing a portalID, an object address, and a randomized check field that verifies authority to operate on the named object. The check field permits revocation, and a server may deny access even when called through a valid capability. The portal name allows a client to make RPC calls to a server given a capability for any of the server's resources. Servers can multiplex management of multiple objects through the single portal. For example, a segment capability contains the portalID of the segment server and identifies a segment managed by that server; a domain capability contains the portalID of the domain server and identifies a domain managed by that server.

Opal capabilities are implemented in the runtime package, hidden from users behind a C^{++} interface based on *proxies* [Shapiro 1986]. On the client side, the capability is hidden in an ordinary C^{++} object called a proxy; on the

[1] We assume the existence of an authentication service that supports some protected notion of identity as a basis for these ACLs. ACLs are not implemented in our current prototype; access is always granted to any published or symbolically named capability.

[2] Portals were designed as a kernel primitive for implementing protected procedure call (e.g., LRPC [Bershad et al. 1990]), particularly in a system with user-level threads based on scheduler activations [Anderson et al. 1992].

server side, the check field is stored in a corresponding object called a *guard* that holds a pointer to the actual object named by the capability. The methods of the proxies and guards are messaging stubs with embedded validation checks. Procedure calls to the proxy result in an RPC call to the guard, passing the capability as an argument. The protection boundary and the use of capabilities are transparent to the application. The proxy package is mentioned here for completeness and as an example of the support that can be built above raw portals. The details are beyond the scope of this article.

3.4 Using Protection Domains

A thread running in one protection domain (the *parent* domain) can create a new (more restricted) domain (a *child*), typically to protect the parent's data from an untrusted subprogram. Parents can attach arbitrary segments to their children and cause arbitrary code to execute in their children. Thus, the child fully trusts the parent, but not vice versa. (A child domain could be used also to amplify the parent's rights in a protected way, but in this case a privileged server must create the child on behalf of the nominal parent.)

Protected procedure call is the only means of causing code to execute in a child domain; in particular, there is no notion of "executing a program." To execute code in the child, the parent registers a portal for the child domain, specifies a procedure as the entry point, and then calls through that portal. Ordinarily, a portal is created for a collection of procedures (e.g., the methods of a guard type), and the code executed on entrance to the portal is a server-side dispatcher stub for those procedures. This is the policy supported by our standard runtime package, which allows the parent to create C^{++} objects in segments shared with the child, generate capabilities for those objects, and to invoke them through the portal, so the calls execute in the child domain. Also, parents may pass those capabilities to other children, effectively setting up RPC connections between siblings.

These facilities make it easy to structure user software as a group of cooperating domains, with arbitrary sharing patterns and cross-domain RPC bindings between them. An application can simply create one or more domains, attach code and data to them, and make one or more protected calls, possibly causing the domains to call each other or the parent.

3.5 Linking and Executing Code

The handling of executable code modules in systems such as Opal differs in several respects from that of conventional systems. (This topic is the focus of related work by Garrett et al. [1993].) We wish to make three points in this section: (1) the essentials of linking and execution are the same in both classes of systems, (2) sharing and dynamic use of code modules is easy in the shared address space, and (3) support of private data in shared modules is trickier in some respects.

A *module* is a group of compiled procedures together with (1) a table of symbols for code and static data items (e.g., global variables) defined in the module, (2) initial values for these static data items, and (3) a table of external symbols imported from other modules. The module's code contains memory reference instructions (loads, stores, and branches) using addressing techniques selected by the compiler (e.g., PC-relative, absolute, or register-relative modes). A *linker* utility processes the module to resolve these references; once linked, executing threads can call the module directly at the virtual addresses assigned to it.

Opal differs from conventional systems in the way that read-only symbols (code and constant data) are bound to virtual addresses, and in the way that references to those symbols are resolved. On conventional systems, each module is typically a complete self-contained program, linked to execute in a private virtual address space. The linker statically binds virtual addresses to symbols in the module, ordinarily assigning the same starting address to every program. In contrast, different Opal modules run at *different* addresses in the shared address space, although a given module continues to reside at the *same* address every time it runs.

Opal modules are statically linked into persistent segments of the global address space. The runtime addresses of a module are globally fixed when the module's segment is allocated. There are three distinct benefits from global linking of code modules:

—Threads executing in a module can easily share a single physical copy, even if the threads are running in different protection domains. Shared libraries make the most efficient use of physical storage and can reduce program startup time.

—Procedure pointers can be freely passed and shared at the system level. Procedure pointers are used by some programming languages to implement polymorphic abstract types.

—Any thread can dynamically attach and call any accessible procedure, simply by knowing its address, with no possibility of address conflicts with other modules attached to the thread's domain. This contrasts with conventional systems, in which "programs" cannot generally be called directly because they are linked to execute in a private address space; early protection-structuring choices are "locked in" at static link time in these systems.

Opal also differs from conventional systems in the way that *private static data* is addressed. Multiple instances of a module may exist concurrently, each sharing the module's code, but using a private version of its writable static data. Private data references in the shared code must compute the correct target address for each instance. On conventional systems, each instance of a given module runs in a separate virtual address space; the private copies of its static data typically reside at the *same* virtual addresses within those separate address spaces, although they map to different underlying physical pages. In Opal, different instances of the private static data must exist at *different* virtual addresses. The linker cannot statically determine those addresses or even their PC offsets; therefore, code must use register-relative addressing for private static data, with base register values assigned *dynamically*. (This issue is discussed in more detail in Section 4.5.) Note that this leads to a flexible protection relationship between instances of an Opal module, since the base register value of an executing thread completely determines the instance it addresses. Threads in multiple domains can share an instance, or multiple instances can execute in the same domain, as determined by the language environment.

3.6 Resource Management and Reclamation

Opal provides coarse-grained reclamation, similar to that used in conventional systems. The basic storage management mechanism is explicit reference counting, which applications and support facilities (runtime libraries, language implementations, and garbage collectors) use to allocate and release untyped storage in coarse units. Reference counting of segments is automatic in simple cases: Opal implicitly updates reference counts on *Attach* and *Detach*, deleting segments by default after the last detach. User software can register a persistent reference to a segment, causing it to persist even after the last detach.

Our general philosophy is *not* to dictate data management to languages. In particular, Opal does not attempt to track or control the placement of capabilities and addresses, which can be freely copied and shared in memory. Opal reference counts need not reflect the number of capabilities for a resource. Instead, a reference count indicates the number of entities that have registered an interest in a resource; later, those entities or others must make calls to decrement the reference count. Because erroneous or malicious software can prematurely (or never) release resources, the system must isolate nontrusting entities from reclamation errors. To this end *resource groups* support accounting and bulk deletes of unreclaimed resources, while *reference objects* prevent untrusting entities from releasing each other's resource references. These mechanisms are described next.

3.6.1 *Resource Groups.* Every call to create an Opal resource or to increment a reference count must pass a capability for a *resource group* as an argument. Opal tracks the resource references created on behalf of each resource group and releases those references if the resource group is destroyed. The Opal runtime package retains a *current resource group* for each thread and passes a capability for that group as a hidden argument on server calls; a thread can change its current resource group with a call to the runtime system.

Resource groups are intended as the basis for a resource control policy, e.g., quotas or billing, to encourage or require users and their applications to limit resource consumption. Additionally, they support bulk deletion of a group of related resources. This is useful as a backup should other reclamation mechanisms fail or be untrustworthy. For example, resource groups can be used to free up resources and reference counts held by a particular software entity that no longer exists.

Resource groups are nested to allow a finer grain of control over accounting and resource management. Any holder of a resource group capability can create and delete *subordinate* resource groups, or *subgroups*. Thus the set of resource groups is a collection of trees. Accounting charges flow up the tree: resources allocated by a subgroup are charged to the parent and ultimately to users represented by the root. Deletion privileges extend down the tree: the holder of a resource group capability can delete any descendent resource group, thus releasing all resources held by that resource group, and so on. For example, a new subgroup can be created to run an untrusted procedure and be deleted when the procedure returns. In this way the traditional policy of releasing resources held by a terminating process is easily emulated, yet resource ownership is decoupled from protection domains.

3.6.2 *Reference Objects.* Reference objects separate the counts to a shared resource emanating from different entities. This prevents an untrusted thread from releasing more counts than it requested, triggering early deletion of a shared resource. Reference objects are internal to the implementation of a resource, but their use is reflected in the interface to the resource.

For example, Opal uses reference objects to provide protected reference counts for shared segments. A segment capability names segments indirectly through a reference object (called a *SegRef*). There may be many *SegRefs* for a given segment, each with a private reference count, and each named by a separate capability. Any thread with a capability for a *SegRef* may access the underlying segment, but it can manipulate the reference counts for only the *SegRef* named by its capability. *SegRefs* serve other purposes as well: they support restricted accesses (e.g., a *SegRef* can be cloned conferring read-only permission), and they support selective revocation (i.e., one *SegRef* can be invalidated independently of others for the same segment).

3.6.3 *Dangling References.* Despite the use of reference counts, software can still delete objects prematurely, causing dangling references of various forms. Dangling capabilities are detectable because they contain a randomized 64-bit check field. At the system level, dangling virtual addresses are viewed as an access control problem. Programs granted access to a shared segment are trusted to use it correctly; erroneous pointer references always result from incorrect use either by the thread that stored the pointer, the thread that followed the pointer, or the thread that deleted the pointer's target. User code is responsible for using the available mechanisms (including protection domains and segment access control) to protect itself from damage caused by failures of untrusted threads.

3.7 Summary

This section described the basic Opal abstractions: protection domains, segments, portals, and resource groups. These mechanisms support applications structured as groups of threads in overlapping protection domains, communicating through shared virtual storage and protected procedure call. Virtual address pointers and resource capabilities are freely shared. Resource reclamation is handled with explicit reference counts, backed by bulk delete using resource groups, and enforced by accounting: this 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.

A key point of this section is that memory protection in Opal (using protection domains) has been separated from other issues that are combined with protection (based on processes) in conventional systems. Protection is decoupled from: (1) program execution, through use of RPC as the basic mechanism for animating passive protection domains, (2) resource naming, through the use of context-independent capabilities based on portals, (3) resource ownership, through the use of resource groups, and (4) virtual storage, through the use of named segments in a global address space. Additionally proxies can be used to make protection boundary crossings (RPC) syntactically transparent to applications. The intent of these choices is to make memory protection cheaper, easier to use, and easier to change.

4. IMPLEMENTING AN OPAL PROTOTYPE

We have implemented an Opal prototype on top of the Mach 3.0 microkernel operating system. The prototype has three major components:

—The Opal kernel supports the basic system abstractions (segments, protection domains, portals, and resource groups) and coordinates the usage of address space.

—The standard runtime package supports an application interface tailored to the C^{++} language, including user-mode threads, capability-based RPC, proxies, and heap management.

—A set of custom linking utilities statically links code modules to execute at their permanently assigned addresses in the persistent virtual address space.

We have chosen to implement the "kernel" for our prototype as a server (the *Opal server*) built above Mach, as shown in Figure 2. Although programs in the Opal environment share a single address space, other address spaces can exist on the same host. We chose this path for expediency and to allow Opal to coexist with the Unix implementation also hosted by Mach. We can thus use Unix utilities to bootstrap, monitor, and debug the Opal environment. Unix utilities linked with our runtime package can bind to the Opal server and use a subset of its facilities, e.g., create Opal protection domains and make RPC calls to them, or create and/or attach segments that do not conflict with addresses already used by the Unix process. The prototype was initially implemented on a 32-bit system, the DECstation 5000 (MIPS R3000), and has now been ported to a 64-bit DEC Alpha-based system.

4.1 Use of the Mach Microkernel

Building an Opal prototype was a straightforward process, given existing Mach primitives. Within the single address space, an Opal protection domain is implemented as a *task*, the Mach execution context for threads. An Opal segment is implemented as a *memory object*, the Mach abstraction for a

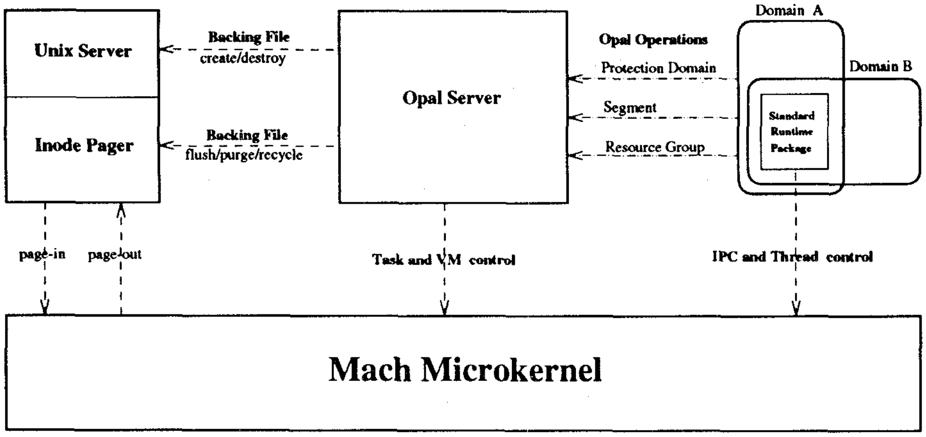


Fig. 2. Organization of the Mach-based Opal prototype.

virtual memory region whose backing storage is managed by a user-mode *paging server*.

Opal applications do not use Mach primitives directly; while they cannot cause damage if they do, this may inhibit their ability to use Opal resources at a later time. Instead, Opal programs make calls to the runtime package and to the Opal server. The Opal server maintains the assignment of virtual address ranges to segments and manages all tasks and memory objects and their relationships.

Opal segments are backed by a modified version of the standard Mach paging server, the *inode pager*. The Mach kernel handles paging operations in Opal segments by making RPC calls to this server. The paging server represents backing storage for each segment as an *inode*, the same structure used to implement files in Unix. We modified the inode pager to allow the Opal server to use backing files directly as Mach memory objects, bypassing Unix *mmap*, and to exploit the fact that they are never accessed through ordinary Unix systems calls. Also, we have added enhancements for Opal backing-file support: e.g., backing files grow physically through zero-fill page faults, and they are recycled with a combined purge/truncate operation when the associated segments are reclaimed.

We have not modified the Mach kernel to support native portals. Naming and resolving of portals is handled by the Opal server, but the runtime system simulates control transfers through portals using the Mach message-passing facility. The Opal server creates a Mach message *port* for every domain; all portals for the domain are multiplexed by sending messages to this single port. This is transparent to Opal applications, which see only a standard interface for RPC through uniformly named portals.

4.2 Standard Runtime Package

The standard Opal runtime package includes support for user-mode threads, with synchronization using locks and condition variables. We use user-mode threads for the traditional reasons—lightweight concurrency and synchronization—and also because their state (when idle) can be passed or shared between domains. Our thread package has several Opal-specific features. First, synchronization objects (e.g., locks) can be shared: the same lock object can be uniformly used for both local and cross-domain synchronization, without sacrificing the performance of the local case. Second, the thread scheduler adjusts processor usage within each domain as processors move between domains for RPC calls and returns. The thread package is discussed in more detail in Feeley et al. [1993].

Thread descriptors and thread stacks are allocated from ordinary heap segments. The thread descriptors hold pointers to some additional thread-specific state. Each thread has a *current resource group*, which is passed as a hidden argument to the server on all calls, and a *current memory pool*, which is used for all heap allocations by that thread. Both items can be retrieved or changed with a runtime system call. Applications use this feature to partition their data structures across multiple segments and to separate their allocated resources into groups, according to their needs for access control and resource control.

The runtime system also caches mappings from portalIDs to Mach ports. If a thread attempts to transfer control to domain S through portal x in S, the runtime system makes a *ResolvePortal* call to the Opal server, passing portalID x, which returns a Mach send right for the port in domain S. Runtime code then caches this association, so it ordinarily resolves each portal at most once. The send right for the domain port for a child protection domain is returned as a hidden result from the domain create operation, so there is no need to look up the mapping for freshly created portals in a child domain.

4.3 The Opal Server

The Opal server creates every protection domain with an *initial data segment*, which the server attaches to itself and preloads with runtime system data structures, including a user-mode thread scheduler and structures to manage RPC connections to other domains. All fresh domains returned by the server have the same initial state: the standard runtime package attached and loaded, one initial data segment, one initial portal, one Mach port to receive messages, and one Mach thread listening for incoming calls on that port. This allows protection domains to be preallocated and cached to reduce the latency of new domain creation calls.

The Opal server keeps a record containing status information for every segment and domain, e.g., indicating the address range allocated to each segment, and the Mach references (port send rights) for the underlying Mach resources. Each domain record has a doubly-linked list of pointers to records for its attached segments. *Attach* and *detach* operations simply modify the list appropriately and map or unmap the segment's memory object from the domain's task at its assigned address. Segment records are indexed by a global structure, used to retrieve a segment record given any virtual address;

this is used for resolving faults on "published" segments. In our prototype this structure is a linear list, but we intend to use splay trees in later implementations.

The server maintains a record for each resource group, organized into a tree that reflects the resource group hierarchy. Each resource group record holds a list of resources charged to it (at present, just domains and *SegRefs*). Deleting a resource group causes the subtree rooted in that resource group to be released.

4.4 Persistence and Recoverability

Segments managed by the Opal server may persist across system restarts. For example, linked code modules are stored in persistent segments. Backing files for persistent segments are stored under synthesized names in a protected directory of the Unix file system, so they can be recovered. Opal capabilities for segments and other resources continue to be valid across restarts. However, our current prototype does not support crash recovery. The server retains address bindings and backing files for persistent address ranges, but it does not propagate updates to backing files to guarantee that they will be in a consistent state after a crash. This support could be provided on a segment basis in a runtime package or paging server.

To support persistent memory and persistent capabilities, the server's address space management structures are themselves retained in persistent segments, rooted in a special segment at a statically determined virtual address. These structures are reattached and recovered if the server is restarted. One problem is that these structures may contain Mach port names for transient resources (tasks, threads, memory objects). These Mach resources and ports are invalidated by a shutdown, so the server must rebuild Mach kernel state and rebind the ports on recovery. To support this, each instantiation of the server is a distinct *epoch*, represented by a value supplied at boot time. As one example of how epochs are used, the current epoch is stored in all segment records and is checked on *Attach*; segment records from a previous epoch have their memory object ports refreshed from the segment's symbolic name.

4.5 Linking

As described in Section 3.5, Opal code modules are statically linked into the global address space and can be shared and called dynamically. A collection of Unix utilities, including standard compilers and build tools, prepares code for execution on the Opal prototype. A shell script calls the Opal server to allocate a segment for the module being linked, then invokes the standard Unix linker, telling it to link the compiled code to run at the virtual addresses assigned to the newly created segment. Custom linking utilities called **Refine** and **Resolve** then postprocess the linker's output. The result is a Unix file that is executable in the Opal environment; this file is then installed directly as the backing file for the module's segment.

The **Refine** utility examines the instructions generated by the compiler, possibly modifying some instructions or addressing modes. Compilers on the MIPS and Alpha generally represent all private static references as offsets from a designated base register, the *global pointer* (GP) register. Any private references based on different addressing styles are modified to use GP-relative addressing; this may cause GP offset space (limited to 64K on these architectures) to be consumed faster. On the Alpha, **Refine** also removes or modifies any instructions that change the value of the GP register. **Resolve** then patches cross-module references (e.g., calls to the standard runtime package), resolving symbols imported from a list of modules passed as an argument. At present, all cross-module text and read-only data references are represented using statically determined absolute addresses. This is simple and efficient, but the resolved references do not automatically bind to new versions of imported modules as they are produced. Language environments desiring automatic rebinding could provide it in the traditional manner, using indirect addressing through dynamic jump tables.

The refined and resolved module can now be attached to protection domains and called directly by threads in those domains. Our prototype treats each attachment as a separate instance of the module, with a private copy of any writable static data. A runtime loader allocates a block from the domain's initial data segment, copies initial values for the module's private static data into the block, relocates any value initialized to the address of another static data element, and initializes the module's *global pointer* to point to the newly allocated static data block. It is worth noting that a private address space system can defer the copy using copy-on-write and avoid the relocation step altogether. We revisit this issue in Section 6.4.

Our prototype is limited in its ability to address private static data of multiple modules attached simultaneously. Since each module has its own global pointer value, these values must be swapped into the GP register on cross-module calls and returns. The Alpha compilers generate instructions to maintain GP values, but the code computes those values as static linker-determined offsets from the PC. In Opal these values are determined dynamically and must be saved and restored from memory (i.e., the private static data of the calling module). We have not modified **Refine** to do this. Instead, our prototype uses one GP value for each domain; all threads entering the domain through portals have their GP register initialized to this value. **Refine** coordinates GP offsets between application modules and "standard" modules, e.g., the standard runtime package. Linked application modules that define private static data are assigned conflicting GP offsets, so domains are restricted to attach only one such module at a time.

4.6 Summary

We have described our Mach-based Opal prototype, which demonstrates that a single-address-space environment can be implemented in a straightforward way on a microkernel operating system. This implementation strategy supports the coexistence of Opal with other operating system environments, such as Unix, thus permitting bidirectional interaction between Opal and existing applications.

We are currently extending our prototype in various ways, for example, to support distribution across a small cluster of workstations. Our basic approach is to use a trusted server that preallocates coarse address ranges to cluster nodes on request (as in Amber [Chase et al. 1989]); the server also provides stable backing storage for the global virtual memory. Cluster workstation and server memories are viewed as a global cache of the server's backing storage, with the goal of satisfying most paging operations with memory-to-memory network transfers, rather than from disk.

Distributing the single address space raises the issue of the coherency of shared segments. We believe that solutions to problems such as coherency and recoverability should be applied at the application level, through a mix of language support, runtime support, and customized external paging servers. That is, many solutions exist (e.g., Carter et al. [1991] and Bershad et al. [1993]), and the operating system should not dictate a single model for coherency and recoverability to all applications. For example, collaborative work applications need different models than do parallel programs. One alternative that we are exploring uses coarse synchronization between nodes and fine-grained synchronization among threads within a node, with updates made by propagating local transaction logs to stable storage and to other caching sites across the network.

5. APPLICATIONS AND PERFORMANCE

This section discusses the use of Opal by applications and presents some performance results. Opal's features—flexible protection, simple shared memory, and mapped persistent storage—are useful in a range of contexts, but we believe they are particularly well suited to the needs of the important and growing class of *integrated* software environments. An "integrated environment" is a collection of software tools that work together to support users in complex tasks: CAD, CASE, image processing, physical modeling, etc. Integrated application systems may be large and evolving; protection is crucial because the "tools" are separately authored programs, possibly running on behalf of different users, and yet these tools must work together and share information efficiently.

We are using the Opal prototype to experiment with integrated environments developed by industry (the Boeing Company) and by software engineering researchers. Our hypothesis is that private-address-space systems encourage poor structuring tradeoffs for these integrated applications, causing loss of protection and/or performance. These applications can be restructured under Opal to improve safety (using additional protection domains), performance (using shared memory), or both. This can enhance their ability to scale, both in the range of functions they provide and in the volume of data they manipulate.

We begin by describing Boeing's aircraft CAD system in order to illustrate the target application domain and its importance. We then present the results of an experiment using an Opal-based implementation of *mediators*, a structuring paradigm for integrated environments [Sullivan and Notkin 1992]. This experiment confirms that shared memory can significantly improve performance and scalability of integrated applications and demonstrates that its use is compatible with sound software engineering principles. Finally, we discuss the performance of the underlying Opal primitives and describe our experience with layering Opal on the Mach microkernel.

5.1 The Boeing CAD System

Our Opal prototype is being studied by Boeing's High Performance Design Systems group, which is experimenting with new database and operating system technologies for next-generation CAD systems. It is useful to consider why the Opal model matches Boeing's needs.

Boeing's current CAD environment uses a centralized relational database; the database describes aircraft parts and indexes tens of thousands of geometry files. Boeing faces three major problems with their CAD system. First, its performance problems add to precious design time and cost; verifying part fits and spatial relationships can take overnight, because it demands a scan of the parts database. Second, the system cannot support anticipated order-of-magnitude growth in the size of the data and in the number of engineers for future projects. Third, it is difficult and expensive to extend functionally. These problems are all caused by fundamental weaknesses in the *structure* of the current CAD system.

The Boeing group, whose charter is to prototype future CAD software, is taking a three-pronged approach to meeting Boeing's scale and performance needs. Their objectives are:

(1) Directly store design information as pointer-based structures in the database and permit CAD tools to read these structures in bulk, cache them in local memory, and navigate them directly. The current relational system represents links as key fields traversed by associative lookups, requiring repeated queries to the database server. Opal represents links as ordinary virtual addresses that can be processed directly by programs and interpreted directly by the hardware.

(2) Explicitly represent commonly needed parts relationships (e.g., spatial and functional) in the database. CAD tools added to the current system must rebuild their internal structures from parts records each time they run, because these structures do not match the predefined database schema. Opal permits storage and sharing of arbitrary program-generated data structures, making it easier to cache and reuse intermediate results computed by these tools.

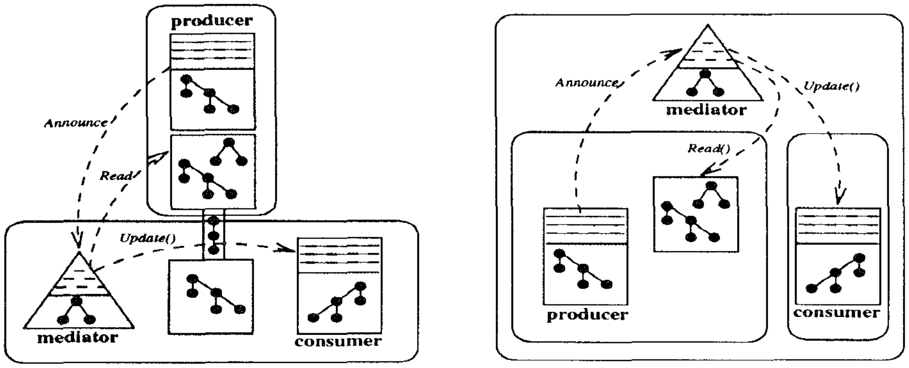
(3) Share cached information among CAD tools on each workstation, to reduce copying and communication overhead. Current tools communicate through reads and writes to the shared database, unnecessarily loading the network and server. Sharing is difficult because tools convert data into process-private pointer structures. Opal allows these structures to be shared, while preserving the meaning of embedded pointers.

Boeing's CAD system is an example of a large integrated application with pointer-rich data structures. We believe that the support afforded by existing operating systems is inadequate for these kinds of database applications. Efforts to support such applications have focused on extensible and object-oriented database systems (OODBs), with a number of commercial products now on the market. In general, OODBs must convert between object IDs and virtual addresses as data moves to and from the database. Under Opal, they can use virtual addresses instead; database format object IDs are not necessary, although they are of course not prohibited. In either case, the key point is that whether or not virtual addresses are directly stored in the database, Opal facilitates the sharing of a single copy of *active* data on the same node, resulting in a more efficient use of memory and database bandwidth than is possible in current OODBs built above private-address-space systems. This is similar in many respects to the Cricket database system [Shekita and Zwilling 1990], which allows CAD tools to directly map a shared database at fixed virtual addresses. In essence, Opal generalizes the Cricket model throughout the entire operating system; there is no distinction between the database and transient virtual memory, and protection domains can be used to prevent tools from viewing or modifying specific parts of the database. This isolation of nontrusting tools is essential if mapped databases are to be used safely.

5.2 Structuring Integrated Tools

Boeing's CAD system shares a key problem with other integrated design environments: given a shared database containing source objects being modified, how do we propagate source changes efficiently to other tools that maintain local structures derived from those sources? In the Boeing environment, the source objects are aircraft part records, and the derived objects belong to tools doing simulation, spatial analysis, and so on. Today, tools must repeatedly copy database records for source objects through messages or pipes, scanning those records and reconstructing local objects derived from those records. With Opal, we wish to access source parts in shared memory, updating the derived objects *incrementally* rather than rebuilding, due to their large size.

This approach can significantly improve performance and scalability. However, integrating independent tools in this way can be quite complex. Tools must make assumptions about the format of their input and adhere to standards for the format of their output. These standards are necessary regardless of how the tools communicate, but the richness and fragility of shared pointer structures would seem to force compromises in the independence of the tools. In contrast, stream-oriented Unix programs (e.g., **awk**, **grep**, etc.) can be combined in unanticipated ways, because they exchange only unstructured streams of bytes. Independence of these tools is achieved at a cost: data is copied by each tool; format conversion is necessary; and incremental change is not supported. We wish to support similar interactions



(a) Decomposed configuration: the mediator and consumer share a protection domain. *Announce* and *Read* are protected calls; data returned by *Read* is linearized in the reply message.

(b) Overlapping configuration: the mediator's domain has read access to the producer and read-write access to the consumer.

Fig. 3. Two configurations of a tree-indexing program built using mediators.

for tools with larger and richer information structures, without this cost of converting and copying "lowest-common-denominator" interchange formats.

5.2.1 *Mediators.* We have implemented a framework to facilitate these kinds of tool relationships based on a variant of *mediators* [Sullivan and Notkin 1992]. Mediators do not solve the integration problem, but they do reduce its complexity. The idea behind mediators is that knowledge of inter-tool dependencies is factored out of the tools themselves and into separate components—mediators—that coordinate the behaviors of the tools. This integrates the tools while allowing them to evolve independently. Tools communicate implicitly by announcing *events* when updates occur. Events announced by data-producing tools are received by mediators acting on behalf of data-consuming tools; these mediators then make a series of calls to both tools to propagate the updates. Tools and mediators should be thought of as modules with private data, executing in a variety of different protection configurations. The calls to announce and propagate updates may be either ordinary procedure calls or protected procedure calls, depending on the protection relationship between the components involved. This structure is depicted in Figure 3.

Our Opal mediator package includes runtime support and a protected *event manager*. Tools register with the event manager, specifying the events they announce, along with capabilities for segments containing procedures and data related to those events. Mediators register with the event manager, specifying events they wish to receive. The event manager sets up the appropriate connections and attachments among tools and mediators. Our mediator implementation is a test system built for demonstration purposes; however, the mediator paradigm has been used to implement interesting systems for program restructuring [Griswold and Notkin 1990], computer-aided geometric design [McCabe 1991], and radiotherapy planning [Kalet et al. 1991], among others.

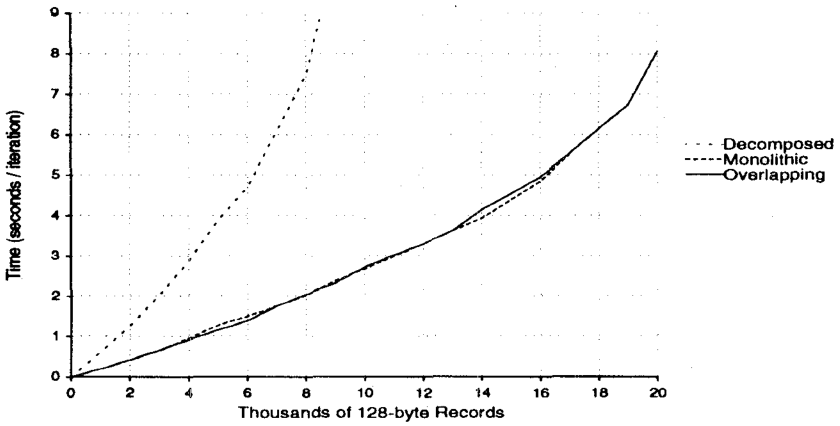


Fig. 4. Execution time for three configurations of a tree-indexing program built using mediators.

5.2.2 *An Example of Structuring Using Opal Mediators.* Opal applications built using mediators can be configured *transparently* to use different protection arrangements, including overlapping protection domains and shared memory. To illustrate the structuring options and their performance, we implemented a simple mediator-based application. A source tool (the *producer*) repeatedly creates and deletes fixed-size records, maintaining a tree index on those records and announcing events after inserts or deletes. A derived tool (the *consumer*) maintains and uses its own index structure on the same data, while a mediator keeps the derived index up to date in response to changes by the producer. An asymmetric trust relationship exists between the tools; the producer can be isolated from both the mediator and the consumer, but both of them trust the producer to supply well-formed input data. Figure 4 shows the performance of this applications, as a function of the number of records processed, for three protection configurations.

(1) *Monolithic.* All tools and the mediator share a protection domain, simulating a single Unix process (sacrificing safety).
(2) *Decomposed.* There are two nonoverlapping domains: the producer in one, the consumer and mediator in the other, using RPC between domains to simulate separate Unix processes (sacrificing performance).
(3) *Overlapping.* There are three Opal domains. The mediator domain has the producer data and code segments attached read-only, and the consumer data segments attached read-write. On an event, the source domain calls the mediator domain with an RPC, passing the address of the changed elements in the shared segment.

Figure 4 confirms the performance benefits of sharing for this environment. The overlapping organization, while as safe as the fully decomposed configuration, is nearly as fast as the monolithic configuration. Furthermore, the performance advantage relative to the decomposed configuration increases as the amount of shared data increases.

The key points of this example are that: (1) read-only shared memory is a natural and efficient alternative to pipes, files, or RPC, (2) read-only shared memory preserves the one-way isolation provided by pipes and scratch files, and (3) programs structured using mediators are easily adapted to run in overlapping protection domains. In fact, all three configurations of our tree-indexing program are built from the same source code; the shared memory is completely invisible to the program. Additionally there is a complete separation of modularity and protection; all interactions are through procedure calls and clean modular interfaces. Overlapping protection domains exist "over top of" these interfaces, representing different trust relationships. For example, an event signal (from producer to mediator) crosses both module and protection boundaries; an update call (from mediator to consumer) crosses module and *possibly* protection boundaries; a source object read call (from mediator or consumer to producer) crosses module but not protection boundaries.

5.3 Prototype Performance

The previous section used measurements of an Opal application to demonstrate the performance advantages that arise from the sharing and protection supported by Opal. Here, we take a look at the performance of some specific Opal primitives, based on our Opal prototype running above Mach (MK83/UX42) on a Digital Alpha processor (DEC 3000/400 AXP, 133.3 Mhz, 74 SPECints). The purpose is to demonstrate that we can build a single-address-space operating system on top of a conventional microkernel (Mach in this case) with performance competitive with other environments supported by that kernel, such as Unix. We have not put much effort into tuning at this point; our objective in building the prototype was to implement an Opal environment rapidly, so that we could investigate experimentally the match between Opal's abstractions and design applications such as those just described.

The cost of certain Opal primitives varies widely because the Opal server preallocates and caches segments and protection domains to meet future needs. Creating a new segment from scratch takes 3.6 ms on the current prototype. Much of that time goes to creating a segment backing file (inode) for the segment. To reduce this cost, Opal recycles inodes; assuming that the recycle list is nonempty, segment creation time is 315 μs, which is the time we expect applications to typically see. Performing an attach/detach of an Opal segment takes 478 μs in the best case. About one fifth of this is the cost of the underlying Mach *vm_map/vu_deallocate* operations; the remainder is the cost of two RPC calls to the Opal server, with a small amount of internal processing to allocate address space and record the attachment.

We now present the performance of Opal operations. As previously stated, Opal domains are preinitialized and cached by the server; a cached domain can be allocated, called, and destroyed in 3.0 ms. The majority of this time is spent destroying the domain; that is, a domain create/call takes about 650 μs and the destroy a little over 2.3 ms. The high destroy penalty results from purging and recycling the domain's data segment, which again requires Unix file system activity.

If no cached domains are available, creating a domain from scratch, calling it, and destroying it is substantially more work, requiring 12.1 ms. This compares favorably with the time for the fork/exec/exit/wait of a null Unix/CThreads program of Mach/Unix (12 ms). The underlying Mach/Unix fork/exec/exit/wait (without CThreads) is 8.75 ms, yet the same operation on DEC's OSF/1, a monolithic Unix kernel, takes only 1.9 ms on the same hardware. This gives some indication of the cost of building any system environment above Mach, but it is not a good measure of overall performance. More importantly, the cost for an Opal cross-domain call is 133 μs, which compares favorably with cross-address space calls on Mach (88 μs) and other systems [Bershad et al. 1990]. This includes a small cost incurred by our runtime support for password capabilities built above Mach RPC.

5.4 Mach's Support for Opal

Implementing the Opal kernel as a Mach server clearly entails some performance loss compared to a kernel-level implementation (as it does for Unix as well), and we can identify some of the major costs. First, the Mach virtual memory system was designed for private address spaces, with a separate translation map for each domain; a global hashed page directory (as in the HP PA-RISC [Huck and Hays 1993] system and others) would be better suited to Opal's single address space. Second, our prototype does not support native portals, so it must start a Mach kernel thread in each new domain and block it on a port, which represents roughly 25% of the cost of preparing a domain. Third, all Opal memory segments are potentially sharable and persistent; this requires that they be backed by an external paging server, which makes zero-fill page faults (336 μs each) and segment deletes (900 μs), somewhat more expensive than the "temporary" virtual memory used by Unix on Mach. Finally, the Mach thread interface is used as the basis for our user-mode thread package, rather than scheduler activations [Anderson et al. 1992].

Overall, the two points to be gleaned from our Mach experience and performance data are: (1) Opal primitives have performance that is reasonable in an absolute sense and (2) they do not add significantly to the cost of the underlying Mach abstractions on which they are implemented. There are no significant hidden costs to the single-address-space model. Of course, we could achieve substantial performance improvements over our Mach-based prototype by building a native Opal kernel implementation; or, we could reduce existing overheads in the current implementation using straightforward and well-understood optimization techniques at various system levels. In any case, we expect applications to ultimately perform better in our environment due to the ease of direct memory sharing and the simplified use of persistent storage.

6. ISSUES FOR THE SINGLE-ADDRESS-SPACE APPROACH

The previous sections have shown how Opal's structure expands the choices for use of protection and sharing, and how applications can exploit this flexibility. These benefits arise, in part, because virtual addresses have an unambiguous global interpretation in the single address space. In this section we discuss several tradeoffs that are inherent in the pure single-address-space approach. In general, these tradeoffs are related to the inability to use context-dependent addressing in single-address-space systems. That is, private-address-space systems often benefit from the assignment of different meanings to the same address.

6.1 Virtual Contiguity

In a single address space, programs are not free to select addresses for the segments they create. In particular, segments cannot grow, and segments allocated at different times will not be contiguous in the address space. This loss of contiguity is not generally a problem for programs written in high-level languages, but it could limit the use of data structures that assume contiguity. Programs with indexed memory structures (e.g., arrays) must request segments that are large enough to store these structures contiguously—yet the system must impose limits on the amount of address space that can be preallocated to allow these structures to grow.

While this scheme indeed limits the maximum segment size, the truth is that the maximum segment size is limited on current 32-bit systems as well; in fact most current operating systems limit the maximum virtual space to a fraction of their four-gigabyte architectural limit, which itself is less than one billionth of a full 64-bit address space. A single-address-space system can thus easily provide larger segments than today's private address spaces.

6.2 Conserving Address Space

Opal manages virtual address space as a global system resource. As with other physical resources (e.g., disk space), the system must ensure that address space is used fairly and that the needs of all users are met. This requires accounting and quotas (based on resource groups), and the system must deny any request that cannot be satisfied from the available address space. These restrictions are not visible in practice if the hardware address space is large enough to meet legitimate needs. We have not answered the question of how much address space is "enough"; our goal is to define and evaluate the single-address-space structure on the assumption that wide-address processors will soon provide sufficient virtual address space for this approach, without constraining applications.

If address space is limited, however, then the system must conserve it by recycling address ranges that have been used and released. Address recycling can occur at several levels. For example, heap managers recycle heap slots within a segment to conserve physical memory as well as virtual address space. In this case, dangling references cannot affect other entities that are not using that heap segment. Should the operating system reclaim the segment and later reassign it, however, an entity with retained references into the original segment might then erroneously attempt to use a pointer to the reassigned segment. This class of dangling references—like all erroneous address references—must be handled by the access control mechanism. That is, neither dangling nor erroneously generated references will permit access to memory, unless that memory lies within a segment attached to the current protection domain.

Recycling is necessary at present given the terabyte-range virtual spaces implemented on current wide-address processors. Opal avoids recycling as long as it can; if the address space is "large enough" then addresses will never be recycled. Kotz and Crow [1994] compare several address space allocation and recycling policies for single-address-space systems.

6.3 Unix-Style **Fork**

The Unix **fork** operation copies a parent process' context, including its private address space, into a child process. The address space cloning semantics of fork cannot be emulated in a single address space: while a Unix implementation can coexist with Opal above a general-purpose microkernel, Unix could not be built above a native Opal kernel.

In the past, there were two reasons for **fork** to clone the parent's address space: (1) to create multiple concurrent processes executing the same program and (2) to allow code from the parent to initialize state (e.g., file descriptors) in the child. In the first case, lightweight threads are a better primitive for concurrency; in the second case, the system can provide a means to initialize a child without executing code in the child's context. In fact, **fork** is a source of complexity and inefficiency in Unix systems: the majority of the state copied by a **fork** is not typically needed by the child, inspiring efforts to improve performance by deferring the copy (e.g., with copy-on-write) or avoid it entirely (e.g., the **vfork** primitive in 4.2 BSD). The copying semantics of **fork** also interfere with support for threads [McJones and Swart 1987].

Opal replaces **fork** with primitives to create protection domains and initialize them by attaching segments and installing RPC endpoints. Threads in the parent domain can then make cross-domain calls to enter the child domain. This model is suited to an environment of software components that cooperate over a period of time; in contrast, **fork** was designed for independently executing programs that compute a result and terminate. Finally, we note that most Unix programs do not use **fork** directly and are thus capable of running in the single-address-space environment.

6.4 Data Copying and Copy-on-Write

An objective of the single-address-space approach is to replace data copying with in-memory sharing wherever possible. Obviously, not all instances of copying can be eliminated. In particular, data may be copied to create a new version that can be modified independently of the original, either for protection reasons or to preserve the original.

When data is copied within an address space, internal pointers within the copied data must be translated. Conventional systems can sometimes avoid this translation if the data is copied from one address space to another, by placing 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 versions.) This is not possible in single-address-space systems, because there is only one naming context; the copy must occupy different virtual addresses than the original, and pointers must be handled specially. Furthermore, a conventional system can lazily evaluate such a copy using copy-on-write, while a single-address-space system cannot. In the single-address-space system, pointers in the data must be relocated before the receiver can even *read* the data (otherwise, the receiver might point to the wrong version when the copy is made); copy-on-reference can be used, but not copy-on-write.

As mentioned in Section 4.5, this absence of remapped copying causes several problems for the handling of code modules in our Opal prototype. Opal must copy initial values of writable static data when a module is loaded (instantiated), and internal pointers must be relocated at load time. The protection domain create time reported in Section 5 reflects the higher cost of initializing code segments that contain writable static data.

It might appear that copy-on-write is completely incompatible with single-address-space systems, because assigning a new address range to the copy introduces *aliasing* in the form of multiple virtual mappings to the same physical page. Copy-on-write introduces only a benign form of aliasing called *read-only aliasing* [Chao et al. 1990]. Read-only aliasing is a hidden optimization that neither violates the consistent interpretation of pointers nor interferes with the use of a virtually addressed cache. General aliasing can cause a synonym problem, because the aliases will have separate entries in a virtually addressed cache, and these entries may be modified independently. This problem does not occur with read-only aliasing: on a write to either virtual address, 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. Because read-only aliasing is harmless, a single-address-space system can use copy-on-write in every instance that a conventional system can use it, except when pointers in the copied data must be translated.

6.5 Summary

We conclude that the tradeoffs of the single-address-space model are reasonable, given sufficient hardware address space. While context-dependent addressing is sometimes useful, an acceptable or better alternative often exists. Furthermore while context-dependent *naming* may be useful in some situations, virtual addresses may be the wrong level to apply the context dependence; context dependence can 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.

7. RELATION TO PREVIOUS WORK

Opal is closely related in style and objectives to a number of other systems, both commercial and experimental, dating back over the last 25 years. An early description of a software system with protection domains, capabilities, and dynamic sharing appeared in the late 1960s [Dennis and Van Horn 1966], and many systems tried to implement that approach.

We believe that Opal is significant because it exploits modern 64-bit processors to meet the goals of previous systems in a way that is simple, general, and efficient. In particular, Opal requires no special hardware, supports uniform context-independent naming at the memory address level, includes persistent storage, and can be implemented efficiently in a compatible way alongside current operating systems on current wide-address processors. Opal also embodies a modern division of responsibilities between the hardware, operating system kernel, system services, and language environment. It accommodates alternative language and data models and relies on language-based facilities for fine-grained protection and storage management.

7.1 Multics and Other Segmented Systems

Opal is similar to Multics [Daley and Dennis 1968] and other segmented systems [Chang and Mergen 1988; Groves and Oehler 1990] in (1) its use of medium-grained virtual memory segments as the units of memory protection and access control and (2) its emphasis on dynamic sharing and memory-mapped persistent storage.

Opal differs from Multics and other segmented systems in one critical respect: program addresses in segmented systems do not have a uniform meaning, which complicates sharing. That is, while a Multics segment has a "global virtual address," programs do not use those global addresses directly. Each process sharing a segment maps the segment into a private address space, perhaps at a different virtual location (e.g., using a different segment register). The result is that processes cannot share or exchange addresses easily, and procedures must include indirect addressing structures to manage the multiple-address assignments of shared data segments on which they operate. In general, all traditional mapped file systems operate in this way.

In contrast, all Opal segments, including persistent segments, are simultaneously and directly visible (given proper protection) in virtual memory by all applications. Linked data structures in Opal can easily span segments, perhaps with different access controls. In this respect Opal is closer to the HP PA-RISC [Lee 1989], which supports traditional segmented addressing, but also allows applications to use global virtual addresses directly. However, most software on the PA-RISC uses short-form addresses, because they are more compact and efficient, and because they permit backward compatibility with private-address-space operating systems.

7.2 Cedar and Pilot

The Xerox Pilot [Redell et al. 1980] and Cedar [Swinehart et al. 1986] systems support a single virtual address space in which all applications execute. Pilot includes a mapped file system as well. Protection is based on the use of a single safe programming language, therefore no protection is provided or necessary at the hardware level.

We agree that safe languages permit a relaxation of hardware protection, and we wish to exploit that fact in Opal. However, we also want to support multiple safe and unsafe languages, as well as stronger isolation for those who want or need it.

7.3 Capability-Based Architectures

Opal's goals are similar in some ways to capability-based hardware systems [Fabry 1974; Levy 1984]. For example, the Intel 432 [Organick 1983] emphasized uniform addressing as a basis for supporting sharing and cooperation. In fact, the addressing on the 432 was not fully uniform, due in part to the way the processor address space was managed: capabilities contained only 24 address bits, which were translated to and from 80-bit UIDs [Pollack et al. 1981] as objects moved between memory and persistent storage. Second, the 432 used very complex hardware-based protection structures and mechanisms to restrict access to fine-grained objects. Opal does not require special hardware support for object-based addressing and protection, and Opal's protection is on a coarser granularity than the 432 and similar architectures.

We believe that most capability systems suffered from (among other things) an insufficient underlying hardware base. Because the underlying virtual address space on the physical hardware was too small, the result has always been an *emulation*, at one level or another, of a large address space system on a small-address-space machine.

The IBM System/38 [Houdek et al. 1981; Soltis 1981] and IBM AS/400 [International Business Machines 1988] are coarser-grained and very similar to Opal in their protection model and granularity. Access to uniformly addressed medium-grained segments is shared by multiple protection domains. (This is different from the 432 in that access to shared user objects generally occurs with unprotected procedure calls, rather than cross-domain calls.) The System/38 provides memory-level hardware support (e.g., tag bits for each word) for capabilities. The key difference again is that Opal uses stock 64-bit processors to gain most of the benefits of these capability-based systems without their costs. Our performance and generality are comparable to standard page-based systems, but with improved support for sharing.

7.4 Object-Based Operating Systems and Languages

Early object-based operating systems, such as Hydra [Wulf et al. 1975], Eden [Almes et al. 1985], and Clouds [Allchin and McKendry 1983], support operating system objects addressed via capabilities. Objects in Eden and Clouds are *coarse grained*, meaning that they are implemented as separate virtual address spaces; object encapsulation is enforced by hard protection boundaries. While these systems conceptually support a single global address space of objects (based on capabilities), this address space exists at a higher level than virtual addresses, prohibiting fine-grained low-cost data sharing and communication between objects. (In fact, this prohibition is a goal of these systems.)

We believe that coarse-grained object-oriented systems confuse modularity and protection in a way that excludes useful structuring choices. Specifically, objects are seen as equivalent to protection domains in these systems. In Opal, objects are lightweight data items that are independent of protection domains; an Opal domain may contain many objects, and objects can be passed or shared between domains. Furthermore, programmers can move the protection boundaries between objects to achieve the desired balance between protection and performance, without affecting the modular structure of the program.

Object-based distributed languages, such as Emerald [Jul et al. 1988] and Guide-1 [Krakowiak et al. 1990], provide lightweight support for fine-grained objects in a distributed global address space. Again, the global address space exists at the object ID level, and translation of addresses is required for communication and storage. Furthermore, these systems require that all applications be written in a single safe language. These languages could run above Opal—and benefit from its flexible protection—but Opal can support sharing of pointer-based data structures without them.

7.5 Monads

Monads [Rosenberg 1992; Rosenberg and Abramson 1985] uses a large (60-bit) shared virtual address space that includes persistent data and spans a local network. A common goal of Monads and Opal is to remove the distinction between persistent data and transient data. Monads partitions the global address space among the network nodes and can locate distributed pages and keep them coherent. Monads demonstrates that a large, sparse, distributed address space is manageable.

Above its paged address space, Monads provides an object-based addressing and protection model supported by custom hardware, including capability registers. In Monads, the purpose of the single address space is to streamline the implementation of capabilities and to extend the capability space across a network. Monads differs from Opal in several key respects: (1) Opal programs use flat virtual addresses directly, with access determined by the protection domain, (2) memory protection in Opal is segment grained rather than object grained, (3) Opal uses stock 64-bit processors, and (4) the data model in Opal is defined by languages and applications rather than by the system.

7.6 Psyche and Hemlock

The Psyche system [Scott et al. 1990] uses related concepts to explore parallel programming within a protected global address space on shared-memory multiprocessors. The focus of Psyche is to build applications out of cooperating components based on different parallel programming models; these components are partially isolated in separate protection domains. Psyche's *realms* enforce object-style access to data shared between models, in part to ensure that model-dependent operations (e.g., acquiring a lock) are handled correctly regardless of which component invokes them. In contrast, Opal focuses on using domains to protect shared information in a multiuser operating system. We generally assume that cooperating components use the same programming model.

Hemlock [Garrett et al. 1993] extends the Psyche work to a Unix context, adding memory-mapped persistent storage and dynamic linking to shared code modules that encapsulate shared data. Hemlock is a hybrid global/private-address-space system: addresses at the low end of the processor address space have separate mappings in each process, with a large globally addressed region at the upper end. The goal is to preserve backward compatibility with Unix (i.e., **fork**) while supporting uniform addressing of shared memory.

Opal differs from Psyche and Hemlock in that it views segments rather than modules (or realms) as the basic unit of sharing. In part this reflects our emphasis on heap-based programming languages, in which a given code module could be used to operate on data in many different segments. For example, in Hemlock, shared code modules are shared abstractions; all data used by a shared module is also shared, and that data is generally referenced with language symbols (e.g., as shared global static data), rather than with pointers. The Hemlock approach simplifies many aspects of linking, but it complicates the use of a given code module to operate on both private and shared data.

7.7 Swizzling

Swizzling [Cockshot et al. 1984] is a method of simulating a large address space on smaller-address hardware by translating pointers (from short "inform" to long "outform," or vice versa) when they are moved in and out of memory. Swizzling has recently gained new popularity as a means of supporting a persistent store [Wilson 1991; Lamb et al. 1991].

As described in Section 2.2, we believe that swizzling should be reserved for exceptional cases, such as relocating groups of objects from one network address space to another. The crucial point, however, is that whether or not data is swizzled on its way into memory, a shared virtual address space is required ultimately to allow applications to share a single copy of that data in memory. The single address space greatly simplifies that sharing.

7.8 Alternative Usage of Large Address Spaces

Several other researchers have proposed using a single wide virtual address space to support uniform addressing and/or reduce the cost of protection [Murray et al. 1993; Vochtello et al. 1993]. Okamoto et al. [1992] suggests MMU hardware that uses the current value of the PC to determine memory access permissions. Druschel and Peterson [1992a] point out that shared segments can be protected from accidental error and even malicious use by "hiding" them in the large address space. This idea is generalized in Yarvin et al. [1993] to allow an untrusted thread to operate on protected data with intradomain "anonymous" protected calls (ARPC) at lower cost than RPC calls across a hardware-enforced protection boundary. ARPC and the Opal model are complementary, *if* Opal's implementation assigns segment addresses randomly (our current prototype does not). That is, ARPC could be used by Opal applications to gain protection of a different strength and cost than hardware protection domains.

8. CONCLUSIONS

We have described the Opal system, a single-address-space operating system targeted for 64-bit address space architectures. The key notions of Opal are the uniform interpretation of addresses and the orthogonality of addressing and protection. Context-independent addressing removes impediments to sharing. Procedures and data can be shared at any time without requiring a priori address space coordination. Protection domains, which are independent of addressing, determine the context in which a thread executes and its access rights to memory segments. Segments are dynamically created and dynamically shared through capabilities, which can be passed directly (along with virtual addresses) through memory or message channels. Once created, a segment can persist over system restarts and will be available simply by accessing its data through virtual addresses.

The objective of Opal is to support a growing and important class of applications that consist of highly interacting tools manipulating a shared persistent database. We have studied the needs of one industrial application (with the Boeing Company), as well as other environments that require dynamic sharing between tools, and have prototyped a mechanism to manage sharing and protection relationships. For such applications, the Opal system simplifies sharing, communication, and persistence, and broadens the choices available for structuring computations and data.

Opal's concepts have much in common with previous systems, in particular, capability hardware systems and object-oriented operating systems. Those systems tended to be overly complex and slow. We believe that the problem was caused by applying (heavyweight) hardware or operating system protection mechanisms to fine-grained user and language objects. Opal provides segment-grained protection and leaves fine-grained storage management to languages and runtime systems. Protection domains can be used in flexible ways to provide added protection, depending on the trust relationships and needs of cooperating parties.

A second problem faced by previous systems was the insufficient addressing capacity of the underlying hardware base, which resulted in an additional level of software to translate from long-term global addresses to short-term virtual addresses. In the final analysis, these systems were all *emulating* a large address space on small-address hardware. With the appearance of RISC microprocessors with 64-bit addressing, we believe that this level of translation is no longer necessary. Moreover, by continuing to use traditional operating system structures, modern operating systems are adding unnecessary complexity; in a sense, they are emulating small-address-space structures on top of large-address architectures, causing extra work at several system layers.

We have demonstrated that a single-address-space system such as Opal can be implemented alongside of other environments on a microkernel operating system, using modern wide-address architectures. This permits experimentation with the single-address-space environment while we continue to use and communicate with existing tools. Our initial experiments with Opal demonstrate how the system can be used to provide added performance, added safety, or both relative to current implementation choices on conventional operating systems.

ACKNOWLEDGMENTS

Thanks to Kevin Sullivan for helping us adapt his work with mediators to the Opal environment, and to Bob Abarbanel, Virgil Bourassa, Joe Koszarek, and Ashutosh Tiwary at Boeing. Jan Sanislo built our Mach system base for the Alpha, while Brian Bershad, Rich Draves, and Peter Stout provided much-needed assistance with Mach. Ted Romer implemented the MIPS linking utilities, and Simon Auyeung modified them for the Alpha. Robert Bedichek, Valerie Issarny, Povl Koch, Julien Maisonneuve, Dylan McNamee, Stefan Savage, and Marc Shapiro commented on early versions of this article. Brian Marsh, Michael Scott, William Garrett, David Keppel, David Kotz, David Redell, Chandu Thekkath, and John Wilkes also gave helpful advice. Nonna Skumanich and Alicen Smith helped with typing for an author impaired by carpal tunnel syndrome. We would also like to thank the Project SOR group at INRIA, and Sacha Krakowiak and his group at Bull/IMAG Labs.

REFERENCES

ALLCHIN, J. AND MCKENDRY, M. 1983. Synchronization and recovery of actions. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*. ACM, New York, 31–44.

ALMES, G. T., BLACK, A. P., LAZOWSKA, E. D., AND NOE, J. D. 1985. The Eden system: A technical review. *IEEE Trans. Softw. Eng. SE-11*, 1 (Jan.), 43–59.

ANDERSON, M., POSE, R. D., AND WALLACE, C. S. 1986. The password-capability system. *Comput. J. 29*, 1 (Feb.), 1–8.

ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. 1992. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst. 10*, 1 (Feb.), 53–79.

BERSHAD, B., ANDERSON, T., LAZOWSKA, E., AND LEVY, H. 1990. Lightweight remote procedure call. *ACM Trans. Comput. Syst. 8*, 1 (Feb.).

BERSHAD, B., ZEKAUSKAS, M., AND SAWDON, W. 1993. The Midway distributed shared memory system. In *Proceedings of the 1993 IEEE Computer Conference*. IEEE, New York.

CARTER, J. B., BENNETT, J. K., AND ZWAENEPOEL, W. 1991. Implementation and performance of Munin. In *Proceedings of the 13th Symposium on Operating Systems Principles*. ACM, New York, 152–164.

CHANG, A. AND MERGEN, M. F. 1988. 801 storage: Architecture and programming. *ACM Trans. Comput. Syst.*, *6*, 1 (Feb.), 28–50.

CHAO, C., MACKEY, M., AND SEARS, B. 1990. Mach on a virtually addressed cache architecture. In *Usenix Mach Workshop Proceedings*. USENIX Association, Berkeley, Calif., 31–51.

CHASE, J. S., AMADOR, F. G., LAZOWSKA, E. D., LEVY, H. M., AND LITTLEFIELD, R. J. 1989. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles*. ACM, New York, 147–158.

COCKSHOT, W. P., ATKINSON, M. P., AND CHISHOLM, K. J. 1984. Persistent object management system. *Softw. Pract. Exp. 14*, 1 (Jan.).

CUSTER, H. 1993. *Inside Windows/NT*. Microsoft Press, Redmond, Wash.

DALEY, R. C. AND DENNIS, J. B. 1968. Virtual memory, processes, and sharing in Multics. *Commun. ACM 11*, 5 (May), 306–312.

DENNIS, J. B. AND VAN HORN, E. C. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM 9*, 3 (Mar.), 143–155.

DIGITAL EQUIPMENT CORPORATION. 1992. *Alpha Architecture Handbook*. Digital Equipment Corp., Maynard, Mass.

DRUSCHEL, P. AND PETERSON, L. 1992a. High performance cross-domain data transfer. Tech. Rep. 92-11, Dept. of Computer Science, Univ. of Arizona, Tucson, Ariz.

DRUSCHEL, P., PETERSON, L. L., AND HUTCHINSON, N. C. 1992b. Decoupling modularity and protection in Lipto. In *Proceedings of the 12th International Conference on Distributed Computing Systems*. IEEE, New York.

FABRY, R. S. 1974. Capability-based addressing. *Commun. ACM 17*, 7 (July), 403–412.

FEELEY, M. J., CHASE, J. S., AND LAZOWSKA, E. D. 1993. User-level threads and interprocess communication. Tech. Rep. 93-02-03, Dept. of Computer Science and Engineering, Univ. of Washington, Seattle, Wash.

GARRETT, W., SCOTT, M., BIANCHINI, R., KONTOTHANASSIS, L., MCCALLUM, R., THOMAS, J., WISNIEWSKI, R., AND LUK, S. 1993. Linking shared segments. In *Proceedings of the Winter 1993 Usenix*. USENIX Association, Berkeley, Calif.

GRISWOLD, W. AND NOTKIN, D. 1990. Program restructuring to aid software maintenance. Tech. Rep. 90-08-05, Dept. of Computer Science and Engineering, Univ. of Washington, Seattle, Wash.

GROVES, R. D. AND OEHLER, R. 1990. RISC system/6000 processor architecture. *IBM RISC System/6000 Technology*, M. Misra, Ed. International Business Machines, Armonk, N.Y.

HOUDEK, M., SOLTIS, F., AND HOFFMAN, R. L. 1981. IBM system/38 support for capability-based addressing. In *Proceedings of the 8th Symposium on Computer Architecture*. ACM/IEEE, New York.

HUCK, J. AND HAYS, J. 1993. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. ACM/IEEE, New York.

INTERNATIONAL BUSINESS MACHINES. 1988. *Application System/400 Technology*. International Business Machines, Armonk, N.Y.

JUL, E., LEVY, H., HUTCHINSON, N., AND BLACK, A. 1988. Fine-grained mobility in the Emerald system. *ACM Trans. Comput. Syst.*, *6*, 1 (Feb.) 109–133.

KALET, I., JACKY, J., KROMHOUT-SHIRO, S., NIEHAUS, M., SWEENEY, C., AND UNGER, J. 1991. The Prism radiation treatment planning system. Tech. Rep. 91-10-03, Radiation Oncology Dept., Univ. of Washington, Seattle, Wash.

KOLDINGER, E. J., CHASE, J. S., AND EGGERS, S. J. 1992. Architectural support for single address space operating systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems. ACM SIGOPS Oper. Syst. Rev. 26* (Oct.).

KOTZ, D. AND CROW, P. 1993. The expected lifetime of "single-address-space" operating systems. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '94)*. ACM, New York.

KRAKOWIAK, S., MEYSEMBOURG, M., VAN, H. N., RIVEILL, M., ROISIN, C., AND DE PINA, X. R. 1990. Design and implementation of an object-oriented, strongly typed language for distributed applications. *J. Obj. Orient. Program. 3*, 3 (Sept.), 11–22.

LAMB, C., LANDIS, G., ORENSTEIN, J., AND WEINREB, D. 1991. The Object Store database system. *Commun. ACM 34,* 10 (Oct.).

LEE, R. B. 1989. Precision architecture. *IEEE Comput.* 1 (Jan.), 78–91.

LEVY, H. M. 1984. *Capability-Based Computer Systems.* Digital Press, Bedford, Mass.

MCCABE, T. 1991. Programming with mediators: Developing a graphical mesh environment. Master's thesis, Dept. of Computer Science and Engineering, Univ. of Washington, Seattle, Wash.

MCJONES, P. R. AND SWART, G. F. 1987. Evolving the Unix system interface to support multithreaded programs. Tech. Rep. 21, DEC Systems Research Center, Palo Alto, Calif.

MIPS COMPUTER SYSTEMS. 1991. *MIPS R4000 Microprocessor User's Manual.* 1st ed. MIPS Computer Systems, Inc., Sunnyvale, Calif.

MULLENDER, S. AND TANENBAUM, A. 1986. The design of a capability-based operating system. *Comput. J. 29,* 4, 289–299.

MURRAY, K., SAULSBURY, A., STEIMERLING, T., WILKINSON, T., KELLY, P., AND OSMON, P. 1993. Design and implementation of an object-oriented 64-bit single address space microkernel. In the *2nd USENIX Symposium on Microkernels and Other Architectures.* USENIX Association, Berkeley, Calif.

OKAMOTO, T., SEGAWA, H., SHIN, S., NOZUE, H., MAEDA, K., AND SAITO, M. 1992. A micro-kernel architecture for next generation processors. In *USENIX Workshop on Microkernels and Other Kernel Architectures.* USENIX Association, Berkeley, Calif.

ORGANICK, E. I. 1983. *A Programmer's View of the Intel 432 System.* McGraw-Hill, New York.

POLLACK, F. J., KAHN, K. C., AND WILKINSON, R. M. 1981. The iMAX-432 object filing system. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles.* ACM, New York.

REDELL, D., DALAL, Y., HORSLEY, T., LAUER, H., LYNCH, W., MCJONES, P., MURRAY, H., AND PURCELL, S. 1980. Pilot: An operating system for a personal computer. *Commun. ACM 23,* 2 (Feb.), 81–92.

ROSENBERG, J. 1992. Architectural and operating system support for orthogonal persistence. *Comput. Syst. 5,* 3 (July).

ROSENBERG, J. AND ABRAMSON, D. 1985. MONADS-PC: A capability-based workstation to support software engineering. In *Proceedings of the 18th Hawai International Conference on System Sciences.* ACM, New York.

ROZIER, M., ABROSSIMOV, V., ARMAND, F., BOULE, I., GIEN, M., GUILLEMONT, M., HERRMANN, F., LEONARD, P., LANGLOIS, S., AND NEUHAUSER, W. 1988. Chorus distributed operating systems. *Comput. Syst. 1,* 4.

SCOTT, M. L., LEBLANC, T. J., AND MARSH, B. D. 1990. Multi-model parallel programming in Psyche. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, New York, 70–78.

SHAPIRO, M. 1986. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the 6th International Conference on Distributed Computing Systems.*

SHEKITA, E. AND ZWILLING, M. 1990. Cricket: A mapped, persistent object store. In *Proceedings of the 4th International Workshop on Persistent Object Systems: Design, Implementation and Use.*

SIEWIOREK, D. P., BELL, C. G., AND NEWELL, A. 1982. *Computer Structures: Readings and Examples.* McGraw Hill, New York.

SOLTIS, F. G. 1981. Design of a small business data processing system. *IEEE Comput. 14,* 9 (Sept.).

SULLIVAN, K. AND NOTKIN, D. 1992. Reconciling environment integration and software evolution. *ACM Trans. Softw. Eng. 1,* 3 (July).

SWINEHART, D., ZELLWEGER, P., BEACH, R., AND HAGMANN, R. 1986. A structural view of the Cedar programming environment. *ACM Trans. Program. Lang. Syst. 4,* 8 (Oct.).

VOCHTELLO, J., RUSSELL, S., AND HEISER, G. 1993. Capability-based protection in a persistent global virtual memory system. In the *3rd International Workshop on Object-Oriented Memory Systems.* IEEE, New York.

WILSON, P. R. 1991. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. *ACM SIGARCH Comput. Arch. News 19,* 4 (June).

WULF, W. A., LEVIN, R., AND PIERSON, C. 1975. Overview of the Hydra operating system development. In *Proceedings of the 5th Symposium on Operating Systems Principles*. ACM SIGOPS, New York, 122–131.

YARVIN, C., BUKOWSKI, R., AND ANDERSON, T. 1993. Anonymous RPC: Low Latency protection in a 64-bit address space. In *Proceedings of the Summer USENIX Conference*. USENIX Association, Berkeley, Calif.

YOUNG, M., TEVANIAN, A., RASHID, R., GOLUB, D., EPPINGER, J., CHEW, J., BOLOSKY, W., BLACK, D., AND BARON, R. 1987. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*. ACM, New York, 63–76.

Received August 1993; revised January 1994; accepted January 1994