17th ACM Symposium on Operating Systems Principles (SOSP '99)
Published as *Operating Systems Review*, **34**(5):170–185, Dec. 1999

EROS: a fast capability system*

Jonathan S. Shapiro† Jonathan M. Smith David J. Farber
Department of Computer and Information Science
University of Pennsylvania
shap,jms,farber@dsl.cis.upenn.edu

Abstract

EROS is a capability-based operating system for commodity processors which uses a single level storage model. The single level store's persistence is transparent to applications. The performance consequences of support for transparent persistence and capability-based architectures are generally believed to be negative. Surprisingly, the basic operations of EROS (such as IPC) are generally comparable in cost to similar operations in conventional systems. This is demonstrated with a set of microbenchmark measurements of semantically similar operations in Linux.

The EROS system achieves its performance by coupling well-chosen abstract objects with caching techniques for those objects. The objects (processes, nodes, and pages) are well-supported by conventional hardware, reducing the overhead of capabilities. Software-managed caching techniques for these objects reduce the cost of persistence. The resulting performance suggests that composing protected subsystems may be less costly than commonly believed.

1 Introduction

EROS is a capability-based microkernel with a single-level storage model. The single-level store's persistence is transparent to applications. Storage allocation, scheduling, and fault handling policies are exported from the kernel to allow multiple operating environments and application customized resource management. To simplify security assurance, all code having either direct access to the hardware or the ability to directly transform the system's security state is collected in the kernel. Bottom-half device drivers and the single-level store are therefore implemented within the kernel.

1.1 History of EROS

EROS is the third implementation of the GNOSIS (later renamed KeyKOS) architecture [15] created by TymShare, Inc. First deployed in 1982, the goal of GNOSIS was to support secure time sharing and controlled collaboration among mutually adversarial users. The first GNOSIS kernel (later renamed KeyKOS/370) was implemented in 370 assembler language, with application code in PL/1. This system ran VISA transaction processing and networking applications. In the late 1980s the kernel and selected components were reimplemented in C for the Motorola 88000 family (KeyKOS/88k). Development ceased in 1991 when the company closed. The EROS project started shortly thereafter as a clean-room reimplementation of the GNOSIS architecture in C++, and has since made minor enhancements to the architecture. EROS for the x86 may be obtained from the EROS web site [41].

1.2 Components and capabilities

Large-scale software architectures are evolving (e.g., CORBA), in which applications can be viewed as capability connected components. This suggests that the study of capability systems is of increasing importance, since by design, capability systems provide support for protected subsystems, non-hierarchical protection domains, and typed objects [26].

Experience with systems such as the IBM AS/400 [47] (a.k.a. System 38), the Plessey System 250 [19, 29], and KeyKOS [22] suggests that reliability and security can be improved at both application and system levels if objects are separated into distinct protected subsystems to provide both fault isolation and discretionary access control. Language-based approaches to enforcing protected subsystems have proven difficult to implement [54], and require expensive conversion of existing applications to the new language. An OS-based solution, possibly augmented by language level mechanisms, is therefore preferable.

However, the desirable features of transparent persistence and capability-based protection are widely believed to have prohibitive performance. This belief is largely justified by experience. The performance issues of the i432 have been examined by Colwell [7], and those of Mach by Ford [14]. While the IBM AS/400 [47] (a.k.a. System 38) has been a large-scale commercial success, its performance depends on

* This research was supported by DARPA under Contracts #N66001-96-C-852, #MDA972-95-1-0013, and #DABT63-95-C-0073. Additional support was provided by the AT&T Foundation, and the Hewlett-Packard, Tandem Computer and Intel Corporations.

† Author now with the IBM T. J. Watson Research Center.

an augmented processor instruction set and a tagged memory system.

1.3 EROS and what we show in this paper

This paper presents the EROS architecture, and describes an efficient reduction to practice of this architecture for a commodity processor family: Intel's Pentium[23]. We discuss the impact of persistence on the system's design in both structure and complexity. The EROS design and implementation applies the architectural ideas of microkernels to obtain performance in a system that is secure, capability-based, and transparently persistent.

We evaluate the performance of the resulting system using microbenchmarks. While the microbenchmarks currently available for EROS do not support general conclusions about application performance, they suggest that EROS will achieve reasonable performance on end user applications. The results suggest that capabilities are a reasonable substrate for a high-performance, high-security system on commodity hardware.

2 Capabilities

A ***capability*** is an unforgeable pair made up of an object identifier and a set of authorized operations (an interface) on that object [9]. UNIX file descriptors [51], for example, are capabilities.

In a capability system, each process holds capabilities, and can perform those operations authorized by its capabilities. Security is assured by three properties:

1. capabilities are unforgeable and tamper proof,
2. processes are able to obtain capabilities only by using authorized interfaces, and
3. capabilities are only given to processes that are authorized to hold them.

A ***protection domain*** is the set of capabilities accessible to a subsystem. An essential idea of capability-based system design is that both applications and operating system should be divided into cleanly separated components, each of which resides in its own protection domain.

Subject to constraint by an external reference monitor (Figure 1), capabilities may be transferred from one protection domain to another and may be written to objects in the persistent store. Access rights, in addition to data, can therefore be saved for use across instantiations of one or more programs.

2.1 Implications for persistence

When persistent objects are permitted to contain capabilities, they take on certain characteristics of file metadata. In conventional file systems, correct stabilization of files depends on the order of data and metadata writes: data must be written before the metadata that references it [18]. Similarly, objects in a capability system must be written before the capabilities that reference those objects. Unlike a file system, however, the update dependency graph in a capability system is unconstrained (and potentially circular).

Plausible mechanisms to ensure a consistent system image in the store include application-managed transactions or some form of periodic consistent snapshot of the machine state. For reasons of simplicity and correctness, EROS uses a periodic checkpoint similar to the KeyKOS mechanism described by Landau [28].

2.2 Design challenges

There are five key challenges in architecting a capability system.

First, transferring control across protection domain boundaries is expensive unless great care is taken in implementing protected control transfers. By default, no access should be shared across such a boundary. In a conventional architecture, this requires that the preceding context (the TLB and cache contents) be made unreachable by means of hardware tagging, cache flush, or other hardware-enforced means of isolation. Strategies for performing efficient context switch have been developed [30, 32], and we have applied these ideas for protection domain crossings and presented the results in [44].

Second, we would like a uniform protection mechanism for *all* system resources. Choosing primitive protected objects and access rights that reduce directly to the hardware with minimal overhead is critical. If too large a unit of protection (e.g., memory regions) is selected, as in Mach [1], software is required to translate this abstraction into something that the hardware can implement. If too small a unit of protection (e.g. words) is used, the hardware protection mechanisms cannot be used directly and protection must be realized by software emulation.

Third, a collection of system services must be identified that compose these primitive objects into higher level abstractions efficiently and securely.

Fourth, the system must ensure that a correct initial arrangement of entities and access relationships is achieved when the system starts up. In particular, the correctness of this arrangement must be maintained at the stable storage boundary. Many capability systems use conventional file systems protected by access control lists. In doing so they give up the security advantages of capabilities.

Finally, capability systems have difficulty with traceability and selective revocation: determining who has what access and removing a particular user's access to an object. To solve this problem, hybrid designs using both capabilities and access control lists have been proposed in [4] and [25]. In a pure capability system like EROS, this issue must be addressed by the reference monitor.

2.3 Mandatory access controls

EROS provides a primitive mechanism for revoking access to objects. Both objects and their capabilities have a version number. If the version numbers do not match, the capability is invalid and conveys no authority. Mandatory access control (MAC) policies require a finer mechanism: one that provides selective revocation and access traceability. The EROS constructor (Section 5.3) enforces a discretionary policy [46] similar to Lampson's confinement policy [27]. This policy allows secure reference monitors to be built at user level, as has been done in KeySafe [38].

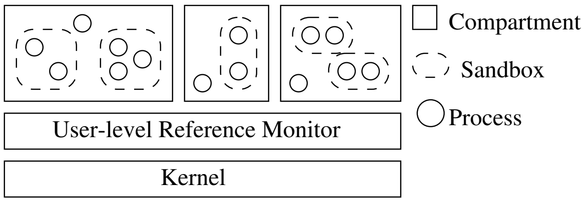

Figure 1. Components within compartments

The KeySafe design divides a secure system into protected compartments (Figure 1). Communication between these compartments is mediated by a reference monitor, which inserts transparent forwarding objects in front of all capabilities that cross compartment boundaries. To rescind the access rights of a compartment, the reference monitor rescinds the forwarding object.

A particularly attractive aspect of the KeySafe design is that it allows straightforward revision of mandatory access policies. Appropriate mandatory access control policies depend on the objects controlled, and are therefore application dependent. The KeySafe design facilitates modification of the mandatory access controls as the system evolves.

Evaluators spent several months in 1986 examining KeyKOS/KeySAFE, and encouraged its submission for B2 evaluation [53]. The B2 security requirements specifically cover both traceability and revocation, and the NCSC team felt that the KeySafe design was sound.

3 The EROS kernel

The EROS architecture is divided into a kernel that implements a small number of primitive capability types: numbers, nodes, data pages, capability pages, processes, entry and resume capabilities, and a few miscellaneous kernel services. These are described in this section. The architecture also includes a collection of system services that are implemented by non-privileged applications. These services provide higher-level abstractions such as files, directories, and memory objects.

The kernel presents a fairly direct virtualization of the underlying hardware via capability-protected abstractions. Both data and capabilities are stored in ***pages***, whose size is dictated by the underlying hardware. Capabilities are also stored in ***nodes***. Nodes hold 32 capabilities, and serve a function equivalent to metadata in conventional systems. All state visible to applications is stored in pages and nodes.

The kernel enforces a partition between data and capabilities by tagging. Data can be read/written only to data pages and capabilities to capability pages. A capability page is never mapped in such a way that access is permitted from user mode programs. Capability load and store instructions are emulated in supervisor software, and check the per-page type tag.

The kernel also implements LRU paging and the dispatch portion of a scheduler based on capacity reserves [35]. As these ideas are not new, they are not discussed further in this paper.

3.1 Address translation

Like other recent microkernels, EROS views an address space as a set of mappings of the form:

$$vpage \longrightarrow ppage \times \{r, w\} \times handler$$

where $handler$ is an ***entry capability*** to a process (Section 3.2). When an addressing exception occurs, the fault handler may either alter the structure of the address space and restart the process or pass the fault to the process fault handler for further handling.

The EROS kernel captures address space mappings in machine-independent fashion using nodes.[1] Address spaces are formed by building a tree of nodes whose leaves are data or capability pages. Each node contains 32 capabilities, which may point to either nodes or to pages (Figure 2). Node capabilities encode the height of the tree that they name, enabling short-circuit traversal similar to that of the MC68851 [36], and avoiding the need for tall translation trees.

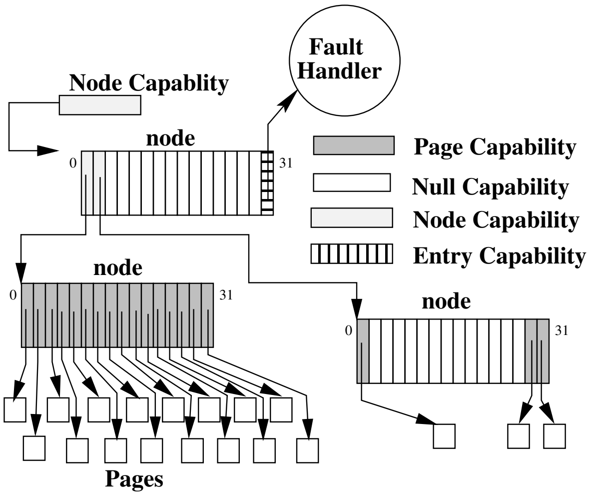

Figure 2. EROS memory tree

Every EROS process includes a capability that names the root of its address space tree. The mapping described by this tree is translated on demand into the representation required by the underlying hardware. When a page fault occurs, the EROS kernel first attempts to traverse the address space tree to establish a valid mapping, performing any necessary object faults to load pages and nodes from the disk. If a valid mapping is found, this mapping is installed in the hardware

[1] To those familiar with earlier capability systems, a node may be thought of as a fixed-size c-list [29].

mapping table and the operation is restarted. Otherwise, the fault is reflected using an upcall to a user-level fault handler specified by the address space (if present) or the process (otherwise).

Using a tree of nodes rather than page tables to describe address space mappings permits relatively fine-grain specification of fault handlers (Figure 2). Information about fault handlers is stored in the node-based mapping tree, but is not directly captured by the hardware mapping tables.

3.2 Processes

EROS process state includes the user-mode registers of the underlying processor, including the user-accessible portion of the processor status word. In addition, each process has a capability register set implemented by the kernel. The capability registers contain those capabilities that the process can directly invoke.

The kernel exports the process abstraction to application code via two types of capabilities: ***process capabilities***, which provide operations to manipulate the process itself, and ***entry capabilities***, which allow the holder to invoke the services provided by a program within a particular process.

Because EROS is a persistent system, all process state must ultimately reside in structures that can be stored to disk. In EROS, these structures are pages and nodes. Process state is recorded using nodes (Figure 3). Data register values are stored to the node using ***number capabilities***, which name an unsigned value and implement *read* operations.

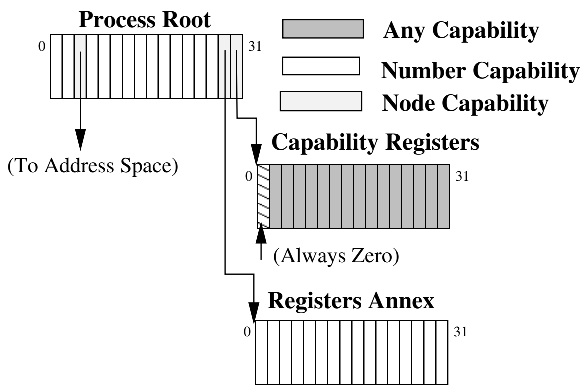

Figure 3. Process structure

EROS makes no distinction between processes and threads. All processes are single-threaded. Two processes may share an address space; each of these processes can hold distinct capabilities. Multithreaded services are implemented by having several processes that share a common address space. Most of these processes serve as worker threads. A distinguished process publishes the externally visible entry point to the service. This process accepts requests and forwards them to the worker processes.[2]

In EROS, each process is a protection domain. Applications are designed as multiple cooperating processes. Application-transparent persistence ensures that consistency between processes is preserved across system restarts. Because the arrangement and consistency of these processes is not lost in the event of a system crash, the associated interprocess relationships need not be reconstructed every time the application is accessed.

3.3 Capability invocation

All resource access in EROS is ultimately performed by capability invocation.[3] If authorized by the capability, each invocation causes the named object to perform some object-defined operation specified by the invoker. Even memory accesses performed by load and store instructions may be modeled as an efficiently cached capability invocation. The semantics of memory reference is therefore covered by the capability semantics. Capabilities are used to invoke both kernel implemented services (pages, nodes, processes, numbers) and services or objects implemented by user processes. Process-implemented services are implemented using a "reply and wait" loop; if the service is not available at the time of the call, the caller blocks until the service becomes available.

Capability invocations transmit a small number of data registers (4), a contiguous data string, and a small number of capability registers (4). If desired, the sender can cause a distinguished entry capability called a ***resume capability*** to replace the last capability argument. The resume capability enables the recipient to reply to the sender.

All copies of a resume capability are efficiently consumed when any copy is invoked, ensuring an "at most once" reply. Manifesting the callers continuation as a capability instead of using an implicit stack allows non-hierarchical interprocess control flow, which is useful for thread dispatching. Distinguishing start and resume capabilities (and the associated process states) also allows applications to construct an implicit mutex around extended co-routine interactions by invoking an existing resume capability and generating a new resume capability with each interprocess control transfer. This mechanism is sometimes used to ensure that extended string transfers are not interrupted by other invocations.

Since entry capability invocations (including resume capabilities) are very common, EROS includes a fast interprocess communication (IPC) mechanism described in [43] and [44]. Although the implementation reported here has not been carefully optimized, earlier implementations have exactly matched the performance of L4's IPC primitive [44].

An unusual aspect of the architecture is that capability invocation is the *only* "system call" implemented by the kernel. Because there are no other system calls, all actions taken by a process are implicitly access checked. Objects imple-

[2] Orran Krieger has noted that this implementation has unfortunate processor locality implications on large SMP systems. A planned kernel-implemented dispatcher object will address this.

[3] For emulation of other operating systems, the authority to explicitly invoke capabilities can be disabled by a per-process mode bit. If capability invocations are disabled, the invocation trap instruction is treated as a conventional exception and reflected via upcall to the per-process fault handler.

mented by the kernel are accessed by invoking their capabilities. All capabilities take the same arguments at the trap interface. In consequence, processes implementing either mediation or request logging can be transparently interposed in front of most objects. In particular, mandatory access control across confined subsystems can be enforced using either indirection using the kernel-implemented indirector object or simulation by a transparently interposed filter process, as in KeySafe [38].

3.4 Limiting access propagation

Capability systems permit capabilities to be transferred over authorized channels. As a result, security arguments must consider both the propagation of information and the propagation of access rights. Capability access rights can be surprisingly powerful; a read-only capability to a node permits a read-write capability residing in that node to be extracted, which in turn allows other data structures to be modified.

The EROS architecture uses two mechanisms to address this issue: the ***weak*** access right and access indirection.

Weak access is similar to read-only access, except that capabilities fetched via a weak capability are diminished in authority so as to be both read-only and weak. The net effect is that *transitive* read-only access is ensured. The default copy-on-write pager implementation described in Section 5, for example, remains in the "trusted computing base" for reasons of integrity, but is unable to leak information because it holds only a weak capability to the original memory object. The EROS weak access right is a generalization of the KeyKOS sense capability [22]; sense capabilities are read-only, weak capabilities need not be. A write via a weak capability stores a diminished (i.e. read-only and weak) form of the stored capability.

Access indirection can be used to implement selective revocation. When sensitive capabilities are granted by a reference monitor, the monitor can either transparently forward requests on those capabilities or supply an indirection object in place of the real capability. Access can later be revoked by destroying the indirection object.

3.5 Checkpointing and persistence

The correctness of the EROS operating system relies on the fact that all operations performed by the kernel result in a correct system state so long as the initial state was correct. The legality of an operation depends on the state of the system, which in turn depends on previous operations. This implies that causal order of operations must be maintained.

The challenge in causal ordering lies in ensuring that a correct system state is recovered when an *unplanned* shut down occurs (i.e., a crash). To achieve this, the kernel must periodically arrange for a consistent image to be saved without application support, and without unduly intruding on application performance. The difficulty lies in ensuring that the image written to the disk is correct; once committed, a bad checkpoint cannot be undone. The EROS implementation guarantees that a correct state exists on the disk by means of a transparent persistence mechanism evolved from KeyKOS [28].

3.5.1 Snapshot

As in KeyKOS, the EROS implementation takes a periodic snapshot of the entire machine. This state is written to the disk asynchronously. On restart the system proceeds from the previously saved system image. Because the checkpointed image is globally consistent, causal ordering is maintained.[4]

The snapshot mechanism introduces minimal disruption of execution. While all processes are halted during the snapshot, this phase performs a consistency check and marks all objects copy-on-write (even to the kernel). Care is taken by the snapshot procedure to ensure that capabilities remain in their optimized form (Section 4.1). Memory mappings must be marked read-only to support the kernel copy-on-write implementation, but the mapping structures are not dismantled as a side effect of checkpointing.

An important difference between the snapshot mechanism of EROS (and KeyKOS) and those of L3 [31] or Fluke [52] is that the EROS snapshot mechanism performs a consistency check before the snapshot is taken. Critical kernel data structures are checked to ensure that their pointers point to objects of appropriate type, allegedly read-only objects in the object cache are checksummed to verify that they have not changed, every modified object must have an entry in the in-core checkpoint directory, and the types of capabilities in process slots are checked. If any of these checks fail, the system is rebooted without committing the current checkpoint.

Once committed, an inconsistent checkpoint lives forever. Aside from errors that have occurred while debugging the stabilization code itself, neither KeyKOS nor EROS has been observed to write an inconsistent checkpoint in seventeen calendar years of operation. The check procedure in EROS has caught innumerable bugs in the implementation, including several obscure boundary conditions and a number of stray pointer errors. This proved sufficiently useful that EROS now performs these checks continuously as a low-priority background task.

In the current implementation, the duration of the snapshot phase is a function of physical memory size. On systems with 256 megabytes of physical memory the snapshot takes less than 50 ms to perform. This time includes execution of the system sanity checker described above. A more incremental design would use copy-on-write techniques to implement the snapshot itself. This would make the snapshot phase solely a function of the process table size (Section 4.3), which should reduce the synchronous phase to roughly a millisecond.

[4] A journaling mechanism (also described in [28]) may be used by databases to ensure that committed state does not roll back. The journaling operation violates causal ordering, but is restricted to data objects. Because journaling does not violate the causal ordering of protection state, it does not violate the protection model.

3.5.2 *Stabilization*

Once a snapshot has been captured, the state is written to the disk asynchronously. To avoid checkpoint-induced delays, stabilization must complete before the next checkpoint, which is typically set at 5 minute intervals. To prevent overrun when applications perform unusually large numbers of object mutations, a checkpoint is also forced when 65% of the total checkpoint area has been allocated to the current checkpoint. Experience with the KeyKOS design [28] suggests that this design can scale in memory capacity with the bandwidth of available I/O channels.

3.5.3 *Bootstrap and installation*

The checkpoint mechanism is used both for startup and for installation. In EROS, an initial system disk image is compiled using a cross compilation environment. The image generation tools link processes by capabilities in much the way that a conventional link editor performs relocation as it builds a binary image. The resulting checkpoint area contains a list of running processes that should be (re)started when the system resumes.

While installation tools are not yet implemented, the built-in checkpoint and replication mechanisms make this straightforward. The program on the installation diskette formats new ranges on the hard disk corresponding to those on the floppy and mounts them. Because they match existing ranges (the ones on the floppy) but have old sequence numbers, the kernel recovers the "stale" range by sequentially marking all objects in the newly duplexed range "dirty" and allowing the replication logic to rewrite them to both replicates.[5] The installation tool simply waits for mirror recovery to complete and forces a checkpoint. The end result is that the image on the hard disk is now a valid bootable image. This mechanism is considerably simpler than the "big bang" used in KeyKOS installation [22].

3.5.4 *Supporting design rules*

Checkpointing is facilitated by two kernel design rules:

- All state resides in pages and nodes.
- All kernel operations are atomic.[6]

Much of the EROS design is predicated on the first restriction, which limits the number of disk object types that must be managed by the persistence subsystem. With the sole exception of the list of stalled processes, this objective is actually satisfied by the current implementation; the EROS kernel owns no state. Restricting the number of data types at this level also reduces the number of storage allocators that must be managed by the implementation.

The atomicity rule is a corollary of the pages and nodes requirement. When a kernel invocation must stall, as when waiting for page I/O, a small, in-kernel "thread" structure containing a capability to the stalled process is queued on the in-kernel stall queue. The invoker's program counter is adjusted to retry the invocation when awakened. This design is similar to that adopted in Fluke [12] and MIT's ITS system [10, 3], and simplifies checkpointing, as checkpointed kernel state requires special handling by the checkpoint mechanism. In the current implementation, the only kernel state that must be checkpointed is the table of stall queue structures. The stall queue structures are also the only kernel resource that can be exhausted by action of applications.

While it is not presently implemented, such exhaustion can be managed by a trusted decongester application. When stall queue structures are oversubscribed, the kernel constructs fault capabilities (a specialized resume capability used by user-level fault handlers to restart a faulted process without changing its state) to the lowest priority processes and passes these fault capabilities to the decongester application for later resumption.

4 Implementation

This section describes how the abstractions of Section 3 are mapped to conventional hardware mechanisms.

The definitive representation for all EROS objects is the one that resides in pages and nodes on the disk. Pages and nodes are cached at two levels of abstraction by the EROS kernel (Figure 4). The first level is for efficient access by the processor. The second level is simply an object cache, which is a fully associative, write-back cache of the on-disk objects. Address translation causes the contents of some nodes to be cached in hardware mapping tables. Process invocation causes other nodes to be cached in the process table. The process table is managed as a write-back cache that sits in front of the object cache. Hardware mapping structures are treated as read-only. Both are flushed when the capabilities they cache are invalidated.

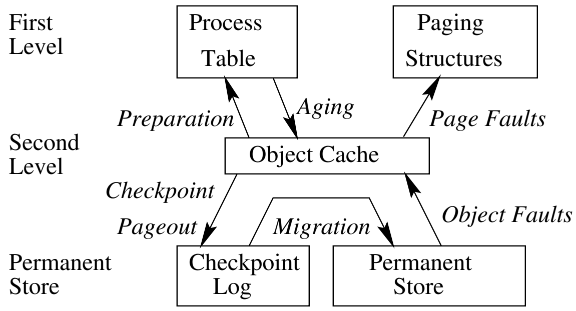

Figure 4. Layered caching. Arrow labels indicate the operations that induce load and flush of each cache.

Use of software caching at multiple levels of the system means that the representation of performance-critical data structures should be easily customizable to the processor architecture. Also, performance-critical portions of the implementation are free to use either the machine specific or the machine independent representation. In several cases, a hybrid approach has been used to improve performance. Fast paths use the machine-specific form, and are backed by a general path using the machine independent structures.

[5] Replication is currently implemented; mirror recovery is not.

[6] We exclude observability of data mutation from our definition of atomicity, which is necessary in multiprocessor implementations. The essential point is that access right updates be atomic.

4.1 Capabilities and object loading

An object capability logically contains a 64-bit unique object identifier for a node or page, a 32-bit version number, and a type field. In the Pentium implementation, each capability occupies 32 bytes. Every node and page has a version number; if the capability version and the object version do not match the capability is invalid. Nodes additionally carry a 32-bit "call count", giving a total node size of 528 bytes. Discussion of call counts has been omitted for lack of space, and may be found in [45].

As stored on the disk, an object capability actually contains the unique object identifier and version number. The first time a capability is used, it is ***prepared***. The object it names is brought into memory and the capability is converted into optimized form (Figure 5). All optimized capabilities point directly to the object that they name, and are placed on a linked list rooted at the object.

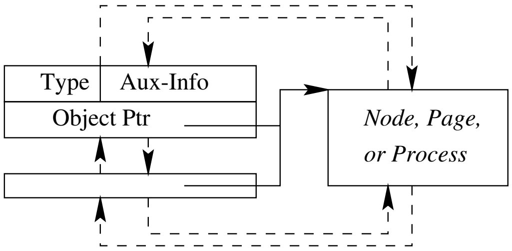

Figure 5. Prepared capability

Preparing a capability also causes the target object to be prepared for use. For node and page capabilities, preparation forces the named object into memory. For process related capabilities, preparing the capability brings in the associated nodes and loads the process into the process table (Section 4.3).

Preparing a capability is one of two operations that causes objects to be converted into hardware specific representations. The other is address translation.

4.2 Address translation

An EROS address space is defined by a tree of nodes. Each node must be obtained from a storage manager. The space occupied by mapping structures is therefore fully accounted for.

The node-based address map must be converted into page table entries to be used by the hardware's MMU. As address exceptions occur, a lazy translation of the state in the node tree into page table entries is performed. The resulting hardware mapping entries are managed as a read-only cache of state that resides in the node trees.

A simple implementation would perform address translation by traversing the node tree, generating a mapping, and updating the page tables. To facilitate invalidation when the node slots are modified, it must record a mapping from capability addresses to the generated page table entries (the ***depend table***).

There are two issues with this implementation:

- Full traversals are expensive. The node tree contains information stored in capabilities. A fair amount of data driven control flow is required to decode this information.
- The naive design does not leverage shared mapping tables on architectures that support them.

The following sections describe how this implementation is optimized.

4.2.1 Reducing traversal cost

The Pentium family uses a two-level hierarchical translation table. EROS also uses a hierarchical mapping architecture; a correspondence can be established between the two mapping hierarchies (Figure 6). Since nodes contain 32 entries, and Pentium mapping tables contain 1024 entries, two node levels are used to describe each of the Pentium mapping table levels. Wherever a valid mapping exists in the hardware page tables, this mapping agrees with the mapping in the node tree. The address translation algorithm exploits this correspondence.

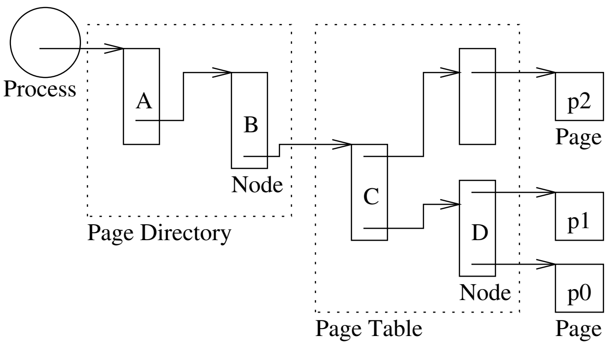

Figure 6. Memory translation for hierarchical MMU

In Figure 6, given a successful translation of page $p0$, a subsequent translation fault on $p1$ or $p2$ must also traverse nodes A and B. Because a valid hardware mapping entry exists in the page directory, the hardware mapping may be used instead *provided* that there exists some means to discover that node C corresponds to the page *table*.

Every core page frame has an associated bookkeeping structure containing various information about the frame. For page frames that contain mapping tables, this information includes a pointer to the ***producer*** of that mapping table. The producer is that node in the mapping tree such that (a) its span in pages is less than or equal to that of the mapping table rounded up to the nearest power of 32 and (b) it has the *largest* span of the candidate nodes under rule (a). That is, we are looking for a node whose span is no larger than that of the corresponding mapping table. The rounding complication addresses the situation in which the hardware mapping table spans are not an integer power of the node spans. It is not relevant to the Pentium family.

In Figure 6, node A is the producer of the page directory, and node C is the producer of the page table. If $p0$ has been successfully translated, a subsequent translation fault on $p1$ proceeds as follows. The high order bits of the virtual address are used to locate the page directory entry. This entry points to the physical address of the page table, which was constructed when $p0$ was translated. Translation now attempts to use the low order bits of the virtual page address to traverse the page table. It locates the page table entry and finds that it is invalid. It then looks in the per-frame bookkeeping structure associated with the page table frame to find the producer of the page table, which is node C. Traversal now proceeds through nodes C and D in the node tree, resulting in a valid page table entry.

The producer optimization means that the majority of page faults traverse only two layers of the node tree, which cuts the average translation cost in half. This is particularly important for fault handling, where at least *part* of the traversal must be done to identify the target fault handler. Producer tracking also means that page tables can be reclaimed by first locating their producer, then locating all valid in-memory capabilities to that producer by traversing the capability chain, and then invalidating all page tables associated with the addresses of the node slots (the array entries) containing those capabilities.

4.2.2 *Shared mapping tables*

Every producer has an associated list of ***products***, which is the list of page tables constructed from that producer. A producer may have multiple products if a short-circuited tree has been constructed. If an address space is described by a single node, the node will be the producer of the page table, the read-only page directory, and the read-write page directory. A producer may also have multiple products if protection bits are not provided at some level of the translation tree. On the Pentium, the address space register does not implement a write protection, so both read-only and read-write versions of the page directory must be constructed following a checkpoint to ensure that copy-on-write occurs while stabilization is in progress.

The product chain is used to ensure that page tables are shared where possible (Figure 7). If either process has successfully translated $p0$, $p1$, or $p2$, then a page table corresponding to nodes E, F, and G has been created and placed on the product chain of node E. When the other process first references any of these pages, the translation logic will find an invalid entry in its page directory. It will then locate the producer for the page directory (either A or C), and traverse the upper portion of the node tree (either $A :: B$ or $C :: D$). In either case, it will discover that node E is at a height corresponding to a page table, and check the product list of E to see if a page table with appropriate permission constraints already exists. Finding the page table created by the previous successful translation, it will reuse this page table rather than create a new one.

4.2.3 *No inverted page tables*

Before a page or page table can be removed, all mapping structures referencing that object must be invalidated. Conventional implementations must maintain a mapping from pages to page table entries (an inverted page table) to accomplish this.

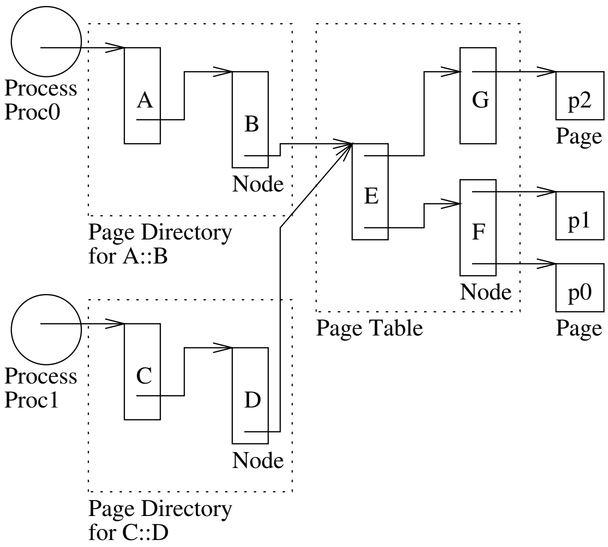

Figure 7. Sharing mapping tables

In EROS, an inverted page table is rendered unnecessary by the capability link chains (Figure 5). If the object to be removed is a page, its prepared capabilities must be traversed to convert them back to unoptimized (i.e., on-disk) form. As these capabilities are traversed for conversion, the previously recorded depend table entries are used to invalidate the page table entries that point to the page. If the hardware mapping architecture is hierarchical, the slots in each address space node correspond to a contiguous region in each of the produced page tables, so only one depend table entry is required per (node, page table) pair. Because of the capability link chain, a per-page entry is required only in the case of a single-page address space (i.e. one with no nodes).

If the object to be removed is a page table, matters are slightly more involved. Note that the producer of the page table spans all of the valid portion of the page table. It follows that the capabilities that *name* the producer dominate the page table entries that point to the page table. Before removing a page table, it suffices to traverse the capability chain of its producer and invalidate the associated depend table entries.

4.2.4 *Small spaces*

Liedtke has shown that creative combinations of segmentation and virtual address translation can yield dramatic performance benefits on some classes of hardware [32]. In such designs, the virtual address space is divided into a single "large space," a universally mapped kernel region, and some number of "small spaces." Boundaries between these spaces are enforced using segmentation rather than page protection.

In effect, this technique uses the segment registers to prepend tag bits to the virtual address. No TLB flush is necessary in control transfers between small spaces. Similarly, no TLB flush is needed when transferring to a large space if that large space is the "current" large space. A flush is required only when the current large space changes or a mapping entry is forcibly invalidated.

Small spaces have a disproportionate impact on the performance of an EROS system. The most critical operating system service applications fit comfortably in less than 128 KB, and these objects are accessed with high frequency. In particular, all of the address space fault handlers currently included in the system run as small spaces, which halves the number of address space switches required to handle a page fault.

4.3 Processes

In EROS, the majority of capability invocations are IPC operations, whose performance depends critically on the low-level process representation. Software caching is used to convert the process structure of Figure 3 to and from a representation optimized for efficient register save and trap handling.

4.3.1 The process cache

The kernel maintains a boot-time allocated process table. Unlike conventional kernel architectures, this process table is a cache. As with other objects, the loading of process table entries is driven by capability preparation. When process A first invokes a capability to process B, the capability is prepared. As a side effect, a process table entry is allocated and the state of process B is loaded into this entry. When a process blocks, a structure containing a process capability to that process is placed on some queue by the kernel. This queue structure is the only information about the process that must remain in memory while the process is stalled. Processes that are ready to execute are queued on the ready queue. When the stalled processes are awakened, this process capability is prepared, forcing the process back into the process table.

Process table writeback occurs either when an entry in the table is reallocated or when a checkpoint occurs. In either case, all capabilities to the unloaded process are ***deprepared***, restoring them to their disk representation. As processes are restarted following the checkpoint, they are incrementally reloaded into the process table. This in turn causes the process' constituent nodes (Figure 3) to be marked "dirty" in preparation for modification. When a checkpoint is in progress, copy on write is used to ensure that the snapshotted version of the state associated with active processes remains unmodified until it has been stabilized.

4.3.2 The save area

The Pentium architecture saves state to a kernel stack when a trap occurs. Rather than copy this state, the EROS kernel arranges for the kernel stack pointer to point directly into the per-process save area (Figure 8). The hardware trap mechanism spills the trap state directly into the process table entry for the active process.

Control is assumed by the trap handler, which completes the register save, loads kernel segments, adjusts the stack pointer to point to the true kernel stack, and transfers control to the kernel. EROS is an "interrupt style" kernel. When a process is suspended in the kernel it is forced to restart at the trap instruction; there is no per-process kernel stack.

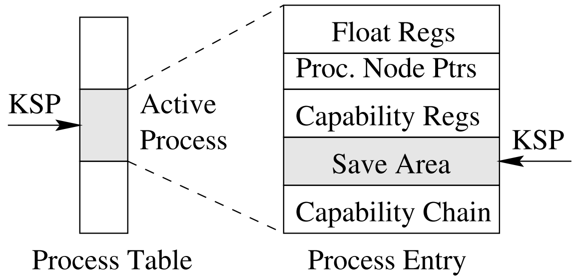

Figure 8. Save area

A Pentium implementation challenges register save design. The saved state has an irregular format (the **error** field is not consistently saved) and is not self describing (the trap number is not made easily available by the hardware). Thus, every hardware interrupt and exception has its own kernel entry point. This entry point appends the error value if necessary and inserts an identifying trap number onto the stack; it then branches to a common interrupt handler. Register save is further complicated by the fact that the processor may be in either "Virtual 8086" mode or protected mode when the interrupt occurs. EROS supports both modes. The state saved to the kernel stack differs according to mode. The kernel therefore keeps track of which mode the application is in and adjusts the kernel save area pointer accordingly before dispatching the process.

4.4 Capability invocation

Capability invocations divide into two cases: fast path interprocess invocation and the general invocation mechanism.

The interprocess invocation fast path is based entirely on the machine specific process structure. It is implemented in assembly code, and handles the case in which the recipient process is prepared and all mappings for the necessary sender and recipient data pages are present in the page tables with valid permissions. If preparation is needed, or if mapping entries need to be constructed, the fast path is abandoned in favor of the general path.

The general invocation path uses the processor specific mapping structures to avoid traversing node trees unnecessarily, but falls back to node tree traversal when necessary. It handles certain cases omitted by the fast path – notably invocations that change the number of running processes (create or delete them). All cases are handled by the general implementation.

4.5 Summary

EROS stores mapping and process state in a machine–independent data structure: the node. This common underlying representation simplifies both checkpointing and storage allocation, and means that system resources “run out” only when the available disk space is exhausted.

This state is converted on demand to the representation required by the hardware for efficient use. Various representation and chaining techniques are used in lieu of conventional data structures to support invalidation and page out. The high-performance paths of the system operate against a machine-specific process structure similar to that of L4 and similar microkernels.

5 System services

Having described the kernel implementation, we can now sketch how these primitives are used to provide higher-level abstractions customarily implemented by supervisor code. To illustrate, we will describe the implementation of three basic services: the storage allocator, virtual copy memory objects and the process constructor. All of these services are implemented by application code.

5.1 Storage allocation

Storage allocation is performed by the ***space bank***, which owns all system storage. The space bank application implements a hierarchy of logical banks, each of which obtains its storage from its parent. The root of this hierarchy is the ***prime space bank***. Every node and page used by an application is allocated from some particular space bank. The fact that all banks are implemented by the same process is not visible to client applications. The term “space bank” is therefore used to refer to a *logical* space bank.

A space bank performs four functions:

- It allocates nodes and pages, optionally imposing an allocation limit.
- It tracks the identities (the OIDs) of the pages and nodes that it has allocated.
- It ensures that all capabilities to a node or page are rendered invalid when the object is deallocated.
- It provides a degree of storage locality. Objects allocated from a given bank are allocated from contiguous extents on the underlying disk.

When a space bank is destroyed, objects allocated by that bank and all sub-banks are either deallocated or returned to the control of its parent bank. Space banks therefore provide a form of explicit storage reclamation. One way to ensure that a subsystem is completely dead is to destroy the space bank from which its storage was allocated.

Space banks manage extents dynamically; applications can ensure locality of storage for two objects by creating per-object sub-banks, creating the objects, and then destroying the bank without reclaiming the storage. Banks are cheap to create and occupy minimal storage (one node per bank). They are therefore well suited to region-based storage management [2]. The unification of extent management and region management provided by this use of space banks extends the locality advantages of region management to permanent storage.

5.2 Virtual copy spaces

A virtual copy space is a copy-on-write version of some other space. Initially, the new space consists of a single node containing a capability to the original object and a capability to a copy-on-write manager. Writes to uncopied pages induce access violations that are directed to this manager. As each page in the new structure is modified, read-write copies are made of the target page and any necessary nodes (Figure 9). Only the modified portion of the structure is copied. Each copied page and node is allocated from the space bank. The bank is provided by the client of the virtual copy object when the copy is first created; storage is accounted to the user.

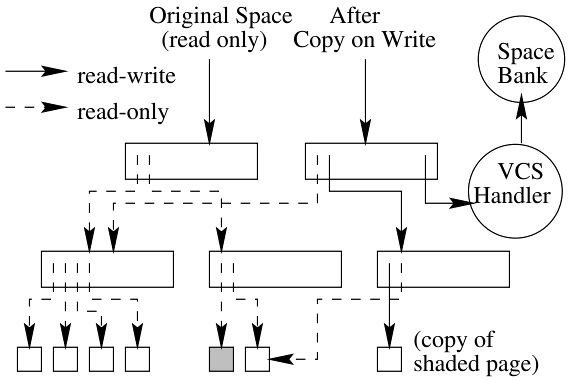

Figure 9. Virtual copy implementation.

To create a lazy copy of an existing memory space, the manager for that space is asked to freeze the space. The manager returns a ***constructor*** (Section 5.3) which will produce copies of the original space. Demand zero objects are realized as virtual copies of the “primordial zero space,” which is part of the hand-constructed initial system image.

We have previously noted that traversing the address space tree is expensive. The virtual copy handler reduces this cost by remembering the location of the last modified page and its containing node. If the next modified page falls within the same leaf node, no traversal is required. Empirically, this simple method matches common memory usage patterns well, and reduces the effective traversal overhead by a factor of 32. In the fast case, a copy-on-write page fault proceeds as follows:

1. Faulting process takes an address translation fault. Kernel walks address space tree, locates no translation.
2. Kernel synthesizes an upcall to the virtual copy handler (context switch).
3. Virtual copy handler purchases a new page from the space bank, initializes it, and installs it at the appropriate offset (two context switches).
4. Virtual copy handler returns to the faulting process, restarting the instruction (context switch).
5. Faulting process takes *another* address translation fault. Kernel walks address space tree, locates a valid

translation, and silently restarts the faulting instruction.

The performance of this implementation is discussed in Section 6.

5.3 Process construction

As with virtual copies, process creation in EROS is performed by application code. Every application has an associated "constructor" that knows how to fabricate new instances of that application. Constructors are themselves applications, which come from the metaconstructor. The metaconstructor is part of the hand-constructed initial system image.

Constructors are trusted objects whose design purpose is to certify properties about the program instances they create. In particular, the constructor is able to tell the client whether the newly fabricated process has any ability to communicate with third parties at the time of its creation. The constructor is able to perform this certification based solely on inspection of the program's initial capabilities, *without inspecting its code*. This means that standard utility programs can be used on sensitive information without risk of information leakage.

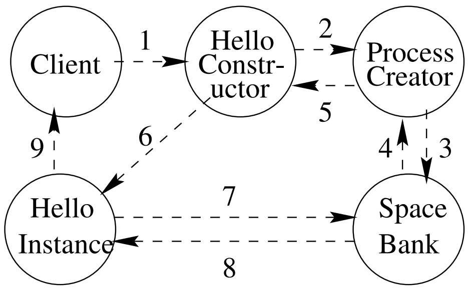

Figure 10. Process creation

To build a new process (Figure 10), the client invokes a constructor (step 1). The constructor invokes its process creator (2), which purchases nodes from a client-supplied space bank (3,4) and returns a new process (5) that initially executes from a constructor-supplied, read-only address space. When started (6) this "construction space" creates a mutable copy of the new program's executable image (either by copying it or by creating a virtual copy), obtaining further space from the space bank as needed (7,8). It transfers control to the new process via a "swap address space and PC" operation. The new instance performs program specific initialization and returns (9) directly to the client process.

Constructors are in practice used as the standard mechanism for packaging and instantiating programs in EROS.

6 Evaluation

The focus of this paper is a capability system design motivated by performance considerations. The best means of comparing its performance to that of other systems would be direct application benchmarks. EROS is new enough that applications have not yet been ported to it, which precludes this comparison at this stage. We have chosen microbenchmarks as our means of evaluation, inspired by those of *lmbench* [34]. Each *lmbench* benchmark is motivated by a real performance bottleneck from some real application. We have examined the constituents of *lmbench* and produced semantically similar tests for EROS. As EROS does not export a raw disk interface and does not yet have a networking implementation we have omitted such benchmarks.

The categories of operations tested by the *lmbench* suite are relatively universal. In many cases, operations with corresponding *semantics* are used by native EROS applications. One way to place the performance of EROS in context is to compare these operations to their nearest Linux equivalents. We have constructed a set of EROS microbenchmarks to do so. The benchmarks are included in the EROS distribution [41].

Measurements were made on a uniprocessor 400 MHz Pentium II, with 192 megabytes of memory. The *lmbench* utilities report memory latencies of 7 ns, 69 ns, and 153 ns for the level one, level two, and main memories, respectively. Linux measurements, where given, are obtained from kernel version 2.2.5-22 on the same hardware using the standard *lmbench* distribution. The Linux kernel measured supports symmetric multiprocessing. Linux measurements reported here are made using a kernel with multiprocessing support compiled out.

The results are summarized in Figure 11. The bars are normalized relative to the Linux timings. Except for the pipe bandwidth benchmark, a shorter bar represents a better result. Actual timings and percent loss/gain are also shown.

6.1 Kernel invocation baseline

The "trivial system call" benchmark is intended to examine the minimum cost of entering and exiting the kernel while performing minimal work. In POSIX this is customarily tested with the *getppid*() operation (0.7 μs). The nearest EROS equivalent is the *typeof* operation on a number capability (1.6 μs).

The difference reflects the difference in the two system architectures. EROS capability invocations must test to verify that the target object is prepared, and must allow for the possibility that control will be returned to some other process. Also, all capability invocations have the same argument structure (Section 3.3). The trivial EROS kernel invocation therefore involves a more complex argument specification and a greater amount of data manipulation. The common argument structure allows processes to transparently interpose on capability invocations, emulating the original object; in this case function was favored over performance in the design.

6.2 Memory handling

Two benchmarks are used to examine memory performance: page fault and heap growth. The page fault benchmark measures the time to reconstruct page table entries for a memory

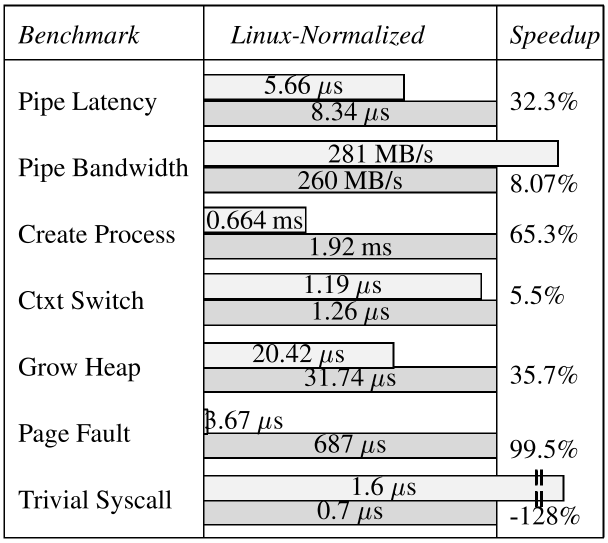

Figure 11. Summary of benchmark results. For pipe bandwidth, larger is better. Linux *lmbench* results appear in dark gray. EROS results are normalized to the Linux numbers, and appear in lighter gray.

object that is valid. That is, it measures the time to convert from the logical mapping structures maintained by the operating system to the hardware mapping representation. The benchmark constructs an object, unmaps it, remaps it, and then measures the time to sum the first word of each page. In Linux, this operation takes 687 μs per page. The corresponding EROS operation takes 3.67 μs per page in the general case, which reflects the basic cost of the node tree traversal. The EROS cost rises to 5.10 μs if the fast traversal optimization of Section 4.2.1 is disabled.

Linux performance on this benchmark has regressed in recent versions of the Linux kernel. The 2.0.34 version of the Linux kernel took 67 μs on this operation.

If the mapped object falls at a page table boundary, the latency under EROS falls to 0.08 μs. This is due to the page table sharing mechanisms described in Section 4.2.2. EROS preserves hardware mapping structures as long as it is feasible to do so. For EROS, this measures a real (and common) case. Page tables may be shared whenever a new instance of an existing application is created. The effect of sharing code space in such cases significantly reduces application startup latencies.

Experience with Mach and Chorus suggested that heap growth may inhibit system performance if it is slow [8, 21]. The POSIX design allows the operating system to grab any available swap page and zero it. We constructed a Linux benchmark using the *lmbench* timing framework, and determined that growing the heap by a page takes 31.74 μs. In EROS, the fault must first be reflected to a user level fault handler which in turn must call a user level storage allocator as described in Section 5.2, which takes 20.42 μs. The virtual copy handler fits within a small (address) space, allowing two of the context switches in the sequence to be done very cheaply (see below).

6.3 Process management

The EROS implementation's "performance core" is very similar in structure to that of L3 [30]. An earlier implementation reported in [44] matches the performance of L3 cycle for cycle. The current implementation is less highly tuned. A directed context switch under EROS takes 1.60 μs in the large space case, and 1.19 μs in the case of a control transfer between a large space and a small space. As measured by *lmbench*, a directed context switch under Linux takes 1.26 μs.

These latencies do not compose in obvious ways because of cache interference. A round trip large-large IPC operation takes 3.21 μs, and a large-small round trip takes 2.38 μs, but a nested sequence of calls such as that seen in the page allocation path (large to small to large and back) takes 6.31 μs.

Where POSIX uses *fork* and *exec* to create new processes, EROS provides a direct method via the constructor. Constructors can guarantee sandboxing, and are the preferred means of starting new program instances. While it is therefore the appropriate mechanism to compare, it should be noted that the *fork/exec* implementation creates an address space that the EROS implementation does not require. Creating a copy of "hello world" under Linux takes 1.916 ms. Using a constructor to create a new copy of "hello world" takes 0.664 ms.

6.4 Interprocess communication

Finally, we examine streaming data transfer using the pipe benchmark. Of the available IPC options, pipes are the most efficient mechanism provided in Linux (8.34 μs latency, 260 MB/sec). The EROS equivalent implements pipes using a process (5.66 μs, 281 MB/sec). The key point to note here is that for bulk data transfer the cost of context switches is much less important than minimizing cache misses. EROS pipe bandwidth is maximized using only 4 KB transfers. This suggests that IPC implementations which impose an upper bound on transfer payload do not impose an inherent inefficiency on data transfer rates. Bounding payloads simplifies the implementation and allows the IPC operation to be performed atomically. It also guarantees that any control transfer operation can make progress given a relatively small amount of real memory, eliminating a memory size dependency at the invocation API.

6.5 Other measurements

While EROS is not yet running applications, previous implementations of the architecture have been deployed with full application environments. An evaluation of the performance of KeyTXF, the KeyKOS transaction manager for the System 370 implementation, is described by Frantz and Landau [16]. Performance on the TP1 benchmark ranged from 2.57 to 25.7 times faster than other protected database systems, and scaled linearly with CPU speed if the I/O system was also upgraded for increased capacity. IBM's TPF was 22% faster (22 transactions per second vs. 18 for KeyTXF), but

TPF was an unprotected system; all TPF applications ran in supervisor mode, and were mutually trusted.

The Motorola implementation of KeyKOS included a full binary-compatible POSIX emulation. A limited performance evaluation was made against Omron's Mach 2.5 implementation (a monolithic kernel) for the same machine [5]. Performance was mixed, ranging from 3.89 times slower than Mach performance on opening and closing files to 14 times faster on memory manipulation. The open/close result reflects a naive implementation, as noted in the paper.

6.6 Performance summary

The benchmark results are summarized graphically in Figure 11. As can be seen, EROS performs favorably on 6 out of 7 benchmarks. We believe that this provides good, although obviously not complete, evidence that composition of protected subsystems (several of which were used in the EROS benchmarks) need not inhibit system performance. In addition, it appears that capability systems can perform well without resorting to specialized hardware assists.

7 Related work

EROS's relation to TymShare's GNOSIS [22] system was discussed in Section 1.1. Most of the GNOSIS architecture is preserved in EROS; the scheduler is a significant departure from the GNOSIS design. Many of the design documents and papers for the GNOSIS system (which was renamed KeyKOS) can be found at the KeyKOS web site [42].

Mach and Chorus: Both Mach [20] and Chorus [39] use capabilities for interprocess communication. Mach also uses them to name memory objects. Both use external memory managers. Neither externalizes storage allocation or provides persistence, and in both cases the external memory manager has proven to be a source of performance difficulties [21, 8]. Both systems are hybrid designs, in that other system calls are present in addition to capability invocation. While Mach originally started with a single *mach_msg* system call, additional system calls were later added for performance.

The Mach ***send-once port*** bears a close resemblance to the EROS ***resume capability***. Send-once ports can be forwarded, but cannot be copied. The absence of special semantics for resume capabilities removes a certain amount of complexity from the EROS IPC path, and slightly simplifies the implementation of debuggers and capability cache objects.

Amoeba: Amoeba [49, 50] is a distributed capability system. Unlike EROS, capabilities are protected by sparsity. Because capabilities are just data, language integration and IPC design are greatly simplified. The cost of this is that capability transfer cannot be detected by a reference monitor. Mandatory access control is therefore impossible to implement outside of the kernel, and metadata update order violations are not detectable by the kernel.

Amoeba does not use capabilities for fine grain system resource such as pages and memory mapping structures. Storage for mapping structures is therefore a hidden overhead rather than clearly accounted for.

L3 and L4: The EROS IPC implementation was heavily influenced by L3 [30, 32] and Mach 4 [14]. The mechanism in EROS is closer to that of L3, but there are some significant differences.

L3 invocation is unauthenticated; in the absence of a chief any process ("task" in L3-speak) may invoke any other process. If required, access controls must be implemented by an intervening process known as a chief, doubling the cost of an authenticated IPC. The most recent iteration of Lava (the L4 successor) incorporates an IPC redirection mechanism that is very similar to capabilities [24]. Even in the newest implementation, however, capabilities are not transferable. Transferring an authority from one process to another requires interaction with the indirection table manager.

L3 is also persistent [31], but lacks an equivalent to the EROS/KeyKOS consistency check. Its checkpoint mechanism is implemented outside the kernel by a memory manager that has direct access to kernel data structures. Data structures included in the checkpoint are allocated out of pageable memory. The kernel may page fault on these structures, and is designed to recover when it does so. Because the kernel directly uses process structure addresses (i.e. there is no indirection), any change in the size of the kernel process table or other checkpointed kernel tables renders previous checkpoints unusable. This is not a limitation of the current EROS mechanism.

Fluke: Fluke [13] is a capability kernel that also provides persistence. Its hierarchical resource management mechanisms are similar to those provided by the EROS space bank and the meter mechanism of KeyKOS. Like L3, Fluke's persistence is implemented by a user-level pager, but its performance is unreported. Also like L3, Fluke's persistence mechanism lacks a consistency check.

Cache Kernel: The Cache Kernel [6] uses a caching approach in some ways similar to EROS. Where EROS writes operating system objects (Processes) back to protected structures (Nodes), the Cache Kernel writes this state back to untrusted application kernels.

Hydra and CAL/TSS: Hydra [56] and CAL/TSS are early software capability systems. Neither was persistent. HYDRA incorporates a port-based store and forward messaging system, which reduces its messaging performance. CAL's messaging is unbuffered and portless, and provides no multicast mechanism. CAL also provides an "event" mechanism, allowing a bounded number of one word messages to be sent asynchronously to the recipient. Of the early capability systems, the CAL design most closely resembles that of EROS.

CAP: The Cambridge CAP computer [55] is a hardware capability system in which capabilities name both protection domains and memory segments. The hardware assist provided by the capability cache provided adequate performance for large grain objects, but use of memory capabilities to describe language-grain objects (e.g., structures rather than pages) leads to higher-frequency dereferencing of capabilities than the hardware can effectively support. Also, the incorporation of *nested* memory capabilities in the architecture, while not leading to in-cache complexity, considerably complicates capability cache miss processing.

Intel i432: The Intel i432 [37] was originally designed as an ADA machine, and ultimately turned into an object-based hardware system. A derivative segmentation architecture can be seen today in the i386. The i432 was designed as a high level language machine. Architecturally, it is a very complex machine, and its performance suffers dramatically as a result [7].

ExOS: MIT's Exokernel [11] provides an alternative approach to application-level resource management. Where the EROS kernel exports kernel-protected entities, the exokernel architecture reorganizes critical data structures to allow processes direct access, and pins these structures in memory. This approach eliminates the need for data copies in many cases, at the cost of exposing a more complex interface contract to the application. What maintainability implications this may have for long-term compatibility as interfaces evolve remain unclear. Greg Ganger, one of the Exokernel designers, has noted that the design of generally usable interfaces in exokernel remains a challenge [17].

The Exokernel uses hierarchically-named credentials called capabilities [33], but these are not capabilities in the traditional sense. They do not name target objects, nor directly authorize access, but rather contain encoded principals that the kernel checks against access-control lists. Exokernel capabilities are more akin to an extensible uid/gid abstraction.

8 Conclusion

EROS' basic abstractions and implementation techniques can realize an efficient capability system on commodity hardware. Microbenchmark performance measurements in comparison with Linux are quite good. Results from other microkernels show that microbenchmark results do not always translate into strong application performance.

A full validation of our performance claim requires application–level benchmarks. The construction of a native EROS execution environment is expected to take several people another year, after which it will be possible to perform such measurements. The conclusion at this stage is that building such an environment is worth pursuing. Experience with KeyKOS [5, 16] suggests that the microbenchmark results reported here approximately predict real system performance.

The design presented here incorporates both user-level fault handling, which is not uncommon in modern microkernels, and, uniquely as far as we know, user-level storage allocation. Performance does not appear to suffer from doing so. We believe this result stems from four aspects of the design:

1. The primitive abstractions implemented by the kernel map directly to the abstractions supported by the hardware. The "semantic gap" between the two is minimal.
2. EROS fault handlers – in particular memory handlers – are less complex than those of similarly user-managed designs. The memory handler is responsible for address validation and storage provisioning, but does *not* make decisions about paging policy or resource arbitration among competing clients. This is in contrast to Mach [21] and Lava, where all of these policies are implemented by a single piece of code. It is similar in this regard to Fluke [13].
3. While encapsulated, the kernel directly exports a machine-independent interface to address mapping structures. This allows portable memory managers to perform optimizations that are not possible when this metadata is opaque to the handler. In contrast, Exokernel's exported memory interface is machine-specific [11].
4. EROS's selection of basic abstractions enables a software caching design model, which in turn allows the implementation to use machine-dependent representations when performance is critical and machine-independent representations for low-likelihood cases or more portable code.

Roughly 22% of the current kernel code is Pentium-specific. An additional 9% relates to hierarchical page table management or node tree traversal, and would be reused for similar translation architectures, such as the Motorola 68000 or 88000 or the Sun SPARC.

The EROS constructor mechanism provides a basic building block for EROS's user-level mandatory access control mechanism. Its correctness has been proven in [46].

The principal failing of capabilities is difficulty of administration. In the absence of a reference monitor, there is no direct way to say "Fred may not use this object." The KeySafe system [38] provides one means to do so within a capability framework. Mechanisms appropriate for flexible mandatory access controls have been explored in DTOS [40] and Flask [48]. With EROS illustrating how to implement capability designs efficiently on commodity hardware, a logical next step is to pursue unifying the two mechanisms efficiently.

The EROS system currently runs on Pentium class hardware. The system is available for download from the EROS web site [41]. The downloadable packages include the benchmarks presented here. A number of groups are currently proceeding with various enhancements of the system, including one that is developing a commercial product built on EROS.

9 Acknowledgements

Several people offered comments and feedback that significantly improved this paper. We particularly wish to thank Leendert Van Doorn, Marc Auslander, and Bruce Lindsay of IBM, Norman Hardy of Agorics, Inc., Charlie Landau of Compaq Computer, and Bill Frantz of Electronic Communities, Inc. Maria Ebling of IBM did a thoughtful critique that significantly helped the clarity of the paper. Jay Lepreau of University of Utah was kind enough to shepherd this paper, and provided useful feedback. Mike Hibler, also of University of Utah, suggested several useful clarifications. We also wish to thank Roger Needham for discussions about CAP and the use of capability systems in active networking applications.

References

[1] M. Acceta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian Jr., and M. W. Young. Mach: A new kernel foundation for UNIX development. In *Proc. 1986 USENIX Summer Technical Conference*, pages 93–112, June 1986.

[2] A. Aiken and D. Gay. Memory management with explicit regions. In *Proc. Principles of Programming Languages*, pages 313–323, Montreal, Canada, June 1998. ACM.

[3] A. Bawden. PCLSRing: Keeping process state modular. Unpublished report. ftp://ftp.ai.mit.edu/pub/alan/-pclsr.memo, 1989.

[4] W. E. Boebert. On the inability of an unmodified capability machine to enforce the *-property. In *Proc. 7th DoD/NBS Computer Security Conference*, pages 291–293, Gaithersburg, MD, USA, Sept. 1984. National Bureau of Standards.

[5] A. C. Bomberger, A. P. Frantz, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *Proc. USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112. USENIX Association, Apr. 1992.

[6] D. R. Cheriton and K. J. Duda. A caching model of operating system kernel functionality. In *Proc. USENIX Symposium on Operating Systems Design and Implementation*, pages 179–193. USENIX Association, Nov. 1994.

[7] R. P. Colwell, E. F. Gehringer, and E. D. Jensen. Performance effects of architectural complexity in the intel 432. *ACM Transactions on Computer Systems*, 6(3):296–339, Aug. 1988.

[8] R. Dean and F. Armand. Data movement in kernelized systems. In *Proc. USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 243–262, Seattle, WA, USA, Apr. 1992.

[9] J. B. Dennis and E. C. van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–154, Mar. 1966.

[10] D. Eastlake, R. Greenblatt, J. Holloway, T. Knight, and S. Nelson. Its 1.5 reference manual. Memo 161a, MIT AI Lab, July 1969.

[11] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. 15th Symposium on Operating Systems Principles*, pages 251–266, Dec. 1995.

[12] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. Interface and execution models in the fluke kernel. In *Proc. 3rd Symposium on Operating System Design and Implementation*, pages 101–115, Feb. 1999.

[13] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Beck, and S. Clawson. Microkernels meet recursive virtual machines. In *Proc. USENIX Symposium on Operating Systems Design and Implementation*, pages 137–151. USENIX Association, Oct. 1996.

[14] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating threads model. In *Proc. Winter USENIX Conference*, pages 97–114, Jan. 1994.

[15] W. Frantz, C. Landau, and N. Hardy. Gnosis: A secure operating system for the '90s. *SHARE Proceedings*, 1983.

[16] W. S. Frantz and C. R. Landau. Object oriented transaction processing in the KeyKOS microkernel. In *Proc. USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 13–26. USENIX Association, Sept. 1993.

[17] G. R. Ganger. Personal Communication, Sept. 1999.

[18] G. R. Ganger and Y. N. Patt. Metadata update performance in file systems. In *Proc. USENIX Symposium on Operating Systems Design and Implementation*, pages 49–60. USENIX Association, Nov. 1994.

[19] E. F. Gehringer. *Capability Architectures and Small Objects*. UMI Research Press, Ann Arbor, Michigan, USA, 1982.

[20] D. Golub, R. Dean, A. Forin, and R. Rashid. UNIX as an application program. In *Proc. USENIX Summer Conference*, pages 87–96, June 1990.

[21] D. B. Golub and R. P. Draves. Moving the default memory manager out of the Mach kernel. In *Proc of the Usenix Mach Symposium*, pages 177–188, Nov. 1991.

[22] N. Hardy. The KeyKOS architecture. *Operating Systems Review*, pages 8–25, Oct. 1985.

[23] Intel Corporation. *Pentium Processor Family User's Manual*. Intel Corporation, Mt. Prospect, Illinois, USA, 1994.

[24] T. Jaeger, K. Elphinstone, J. Liedtke, V. Panteleenko, and Y. Park. Flexible access control using IPC redirection. In *Proc. 7th Workshop on Hot Topics in Operating Systems*, pages 191–196. IEEE, Mar. 1999.

[25] P. Karger. *Improving Security and Performance for Capability Systems*. PhD thesis, University of Cambridge, Oct. 1988. Technical Report No. 149.

[26] P. A. Karger and A. J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proc. of the 1984 IEEE Symposium on Security and Privacy*, pages 2–12, Oakland, CA, Apr. 1984. IEEE.

[27] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[28] C. R. Landau. The checkpoint mechanism in KeyKOS. In *Proc. Second International Workshop on Object Orientation in Operating Systems*, pages 86–91. IEEE, Sept. 1992.

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

[30] J. Liedtke. Improving IPC by kernel design. In *Proc. 14th ACM Symposium on Operating System Principles*, pages 175–188. ACM, 1993.

[31] J. Liedtke. A persistent system in real use – experiences of the first 13 years. In *Proc. 3rd International Workshop on Object-Orientation in Operating Systems*, pages 2–11, Asheville, N.C., 1993.

[32] J. Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Technical Report GMD TR 933, GMD, Nov. 1995.

[33] D. Mazières and M. F. Kaashoek. Secure applications need flexible operating systems. In *Proc. 6th Workshop on Hot Topics in Operating Systems*, pages 56–61. IEEE, May 1997.

[34] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX 1996 Technical Conference*, pages 279–295, San Diego, CA, USA, Jan. 1996.

[35] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: An abstraction for managing processor usage. In *Proc. 4th Workshop on Workstation Operating Systems*, pages 129–134, Oct. 1993.

[36] Motorola, Inc. *MC68851 Paged Memory Management Unit User's Manual*. Englewood Cliffs, New Jersey, USA, 1986.

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

[38] S. A. Rajunas. The KeyKOS/KeySAFE system design. Technical Report SEC009-01, Key Logic, Inc., Mar. 1989. http://www.cis.upenn.edu/˜KeyKOS.

[39] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Hermann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the Chorus distributed system. Technical Report CS-TR-90-25, Chorus Systemes, F-78182 St. Quentin-en-Yvelines Cedex, France, 1991.

[40] Secure Computing Corporation. DTOS lessons learned report. Technical Report 87-0902025A006, June 1997. "http://www.securecomputing.com/randt/technical-docs.html".

[41] J. S. Shapiro. *The EROS Web Site*. http://www.eros-os.org.

[42] J. S. Shapiro. *The KeyKOS Web Site*. http://www.cis.upenn.edu/˜KeyKOS.

[43] J. S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, Philadelphia, PA 19104, 1999.

[44] J. S. Shapiro, D. J. Farber, and J. M. Smith. The measured performance of a fast local IPC. In *Proc. 5th International Workshop on Object Orientation in Operating Systems*, pages 89–94, Seattle, WA, USA, Nov. 1996. IEEE.

[45] J. S. Shapiro, D. J. Farber, and J. M. Smith. State caching in the eros kernel – implementing efficient orthogonal persistence in a pure capability system. In *Proc. 7th International Workshop on Persistent Object Systems*, pages 88–100, Cape May, NJ, USA, 1996.

[46] J. S. Shapiro and S. Weber. Verifying operating system security. Technical Report MS-CIS-97-26, University of Pennsylvania, Philadelphia, PA, USA, 1997.

[47] F. G. Soltis. *Inside the AS/400*. Duke Press, Loveland, Colorado, 1996.

[48] R. Spencer, S. Smalley, P. Losocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *Proc. 8th USENIX Security Symposium*, pages 123–139, Aug. 1999.

[49] A. S. Tannenbaum, S. J. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *Proc. 9th International Symposium on Distributed Computing Systems*, pages 558–563. IEEE, 1986.

[50] A. S. Tannenebaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, Dec. 1990.

[51] K. Thompson. UNIX implementation. *The Bell System Technical Journal*, 57(6):1931–1946, July 1978.

[52] P. Tullmann, J. Lepreau, B. Ford, and M. Hibler. User-level checkpointing through exportable kernel state. In *Proc. 5th IEEE International Workshop on Object-Orientation in Operating Systems*, pages 85–88, Oct. 1996.

[53] U.S. Department of Defense. *Trusted Computer System Evaluation Criteria*, 1985.

[54] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. In *Proc. 16th ACM Symposium on Operating Systems Principles*, pages 116–128, Saint-Malo, France, Oct. 1997.

[55] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and its Operating System*. Elsevier North Holland, 1979.

[56] W. A. Wulf, R. Levin, and S. P. Harbison. *HYDRA/C.mmp: An Experimental Computer System*. McGraw Hill, 1981.