EROS: A Capability System

Computer and Information Sciences Technical Report MS-CIS-97-03

J. S. Shapiro, J. M. Smith and D. J. Farber[1]

Distributed Systems Laboratory

University of Pennsylvania, Philadelphia, PA 19104-6389

`{shap,jms,farber}@dsl.cis.upenn.edu`

June 23, 1997

Abstract

Capabilities define a uniform semantics for system service invocation, enforce separation of concerns and encapsulation, and allow each program to be restricted to exactly that set of authority it requires (the *principle of least privilege*). Capability systems therefore readily contain and reduce errors at the application level and improve component testability. If carefully architected, a capability system should be both *faster* and *simpler* than a comparable access-control-based system. In practice, implementations have failed to demonstrate such performance.

This paper provides an architectural overview of EROS, the Extremely Reliable Operating System. EROS is a persistent capability system which provides complete accountability for persistent, consumable and multiplexed resources. By choosing abstractions to leverage conventional hardware protecgion, and exploiting hardware support in the implementation, a fast pure capability architecture can be demonstrated. This paper describes the system design and the proposed evaluation method for EROS. An implementation of EROS for the x86 platform is currently underway, and we expect to report benchmark results by mid 1998.

1 Introduction

Control of protected resources may use (*object*, *user*, *authority*) triples, which aggregate as an **access control list**[Sal75] (ACL). ACLs have two disadvantages; they are a static lattice of authorities rather than dynamic, and the notion of *user* introduces many opportunities for unintended sharing of authority.

A **capability** is an unforgeable (*object*, *type*, *authority*) triple [Den66]. A capability system is one in which capabilities provide the system's mechanism for security and access control. To access any object in such a system, a program must hold and invoke a capability for that object. Possession of a capability is a necessary and sufficient proof of authority to perform the operations authorized by the capability on the object that it names.

The capability model is attractive for several reasons: It is possible to construct a formal semantics for such a system, and to prove useful properties from that semantics. Simultaneously, capabilities provide a natural framework in which to *expose* the underlying machine representation in a secure and consistent fashion, allowing programmers to better understand application performance and providing fine-grain accountability for resources with low overhead. Finally, programs in a capability-based system can be *confined* [Lam73]. A confined program cannot leak information to an unauthorized party; it lives in a "black box." Such an environment is a safe container within which to run an untrusted program.

Architecturally, since they eliminate consideration of user identity, capabilities reduce the number of name spaces that must implemented by the supervisor. Capability-based systems are readily secured. The set of objects reachable by a program is readily computable: it is a transitive, reflexive, semi-symmetric closure of objects reachable from the program's initial capability set. If the primitive capabilities are properly designed this set can be shown to be closed.[2] The capability model *should* be faster: all other things being equal there is one less criterion to examine (user identity) and no list to traverse to determine access rights.

EROS (the Extremely Reliable Operating System) is a new capability system which provides these advantages while delivering unusually high performance. Preliminary measurements suggest that EROS deliv-

[2] The presence of a shared mutable name space (the file system) in most ACL systems violates this closure. This defeats the reachability computation, rendering security difficult to impossible in practice.

ers performance on most critical kernel paths that closely approaches the maximum derivable from the underlying hardware.

2 Background

Capabilities were the first protection mechanism to be given a semantically rigorous definition [Den66], and were used by several multiprocessing systems of the 1970's (e.g., C.mmp[Wul81], CAL[Lam76], Plessey 2000[Lev84], Sigma-7, Burroughs B5700[Org73]) as their underlying protection model. Operating systems for these machines focused on two issues:

1. bridging the semantic gap between the operating system and the application, and
2. extending the protection semantics of capabilities to the abstractions defined by the system supervisor [Wul81, Lam76, For94].

The first of these issues motivated the CISC architectures of the late 70's and early 80's.

A variety of architectural shortcomings resulted in low performance for these systems, effectively causing the abandonment[3] of fine-grain capability architectures:

- *Excessive abstraction* and the resulting complex objects (e.g., files) preclude enforcing authorities using the underlying hardware[Col88, Wul81, Lam76].
- *Buffered messages* are complex, slow, and the message buffer is an unaccountably multiplexed resource[Wul81, Lam76, For94].
- *Small granularity* protection domains are not well-supported by hardware, and the difficulty of implementation[Wul81, Lam76]. leads to complexity and its associated performance cost.
- *Encryption*, if used to support unforgeability, has significant time (decryption) and space (extra bits) overhead.
- *Indirection* through "object tables" is costly but often used to allow object deletion without locating all outstanding capabilities[Wul81, Lam76].
- *Dynamic allocation* in the supervisor is complex, and usually improperly accounted for, e.g., in the use of variable-length capability lists

[3] with the notable exception of the IBM AS/400

- *Passive stores* require reconstruction of authorities on restart, which requires persistence, but a common file system name space subverts many advantages of capabilities[Wul81, Lam76].
- *Localized persistence*, where modules decide independently when to store, requires communicating programs to implement their own, often expensive, consistency mechanism (e.g., transactions).

Collectively, these failings have deprecated capability systems among both researchers and commercial users. However, the security and reliability guarantees of capability *model* are compelling.

The EROS design process had four phases:

1. Identify the principles that should constrain the design.
2. Examine the various possibilities for primitive, supervisor-implemented abstractions suitable for a capability architecture.
3. Reduce these to a minimum basis set that can be realized effectively using the hardware support available on modern architectures.
4. Implement this set efficiently.

Thus our focus was on performance, and the result is a pure capability system providing performance competitive with monolithic systems. The rest of the paper is about how we achieved it.

3 Design Principles

The architecture of the EROS kernel began with a set of design principles, some of which are taken directly from CAL[Lam76].

Protection Domains All code outside the supervisor runs within some protection domain . A protection domain provides a context that holds the authorities accessible to a process. In EROS, each process is an independent protection domain. A related principle is *isolation*: the supervisor should not be compromisable by any action performed by non-supervisor code. Regrettably, this has led us to admit device mini-drivers into the kernel. Protection domains aid with the *principle of least privilege*, which means that applications should have access to only and exactly those services and objects that are necessary to their operation, minimizing the failure scope of the application, simplifying security arguments [Bom92], enforcing encapsulation[Wul81], and easing software maintainance[Lyc78].

Objects The system provides a virtual world composed of objects [Lam76]. These objects come in a variety of types, which are described in Section 4. Objects are encapsulated: each object defines a set of operations that can be performed on that object by means of an invocation. Objects are named exclusively by kernel-protected capabilities. An object can only be manipulated by performing an *invocation* on a capability naming the object.

Active, Single Level Store The capability model must extend uniformly to the disk; there should be no translation required at this layer in the memory hierarchy. EROS therefore implements a single level store. The state of the machine is periodically snapshotted and written to the store using an efficient asynchronous checkpoint mechanism [Lan92]. The store is *active*: processes are included in the checkpoint and are restarted automatically on recovery. This eliminates the need for the startup and shutdown phases of many programs, and reduces the frequency of program fabrication. Perhaps more important, transparent persistence eliminates the need to re-acquire authorities on system restart. An implication for the supervisor is that it should be *recoverable*; there should be no state visibly held by the kernel between units of operation; all persistent state should be in checkpointable objects.[4]

Atomic Units of Operation All services performed by the supervisor consist of a single, atomic unit of operation with a clearly defined commit point. A supervisor operation either executes to completion or does not alter the state of the system. This simplifies the semantics of each operation, facilitates our approach to persistence, reduces the number of boundary conditions that applications need to check, and lays the groundwork for a fully multithreadable supervisor. The alternative is transactions, which have prohibitive overhead [Sel95].

Expose Multiplexing EROS is a real-time kernel. It is therefore essential to expose resource multiplexing and virtualization to user control. This requires both that resource granularity be sufficiently fine-grain (pages, for example, must be named) and that suitable multiplexing controls be accessible to applications in an appropriately secure fasion.

[4]EROS does not quite meet this objective: the list of currently running threads and the list of active reserves is kept by the kernel.

Account for Everything No resource should go unaccounted for. Complex accounting should be minimized, and if possible eliminated. A corollary to this is the elimination of metadata. Metadata is implicitly allocated by other operations, making it difficult to account for properly.

Minimality The supervisor should not perform any operation that is not required to satisfy these design principles.

4 The EROS Architecture

The EROS supervisor implements a small number of primitive objects, which are shown in Table 1

All objects, including supervisor services, are designated by unforgeable capabilities, which are defined as a triple:

$$Object \times Type \times CapInfo$$

The $Type$ and $CapInfo$ fields combine to define the $Rights$ conveyed by the capability. The meaning of the $CapInfo$ field is specific to the capability type. Different capabilities may name the same object but convey different authorities over that object.

4.1 Invocation

To exercise the object designated by a capability, the holding process *invokes* that capability. The object is invoked, performs the requested service, and replies to the invoker. The invocation trap is the only system call implemented by the EROS supervisor.

Every invocation (including interprocess communication) sends up to 64 kilobytes of data, four data register values, and four capabilities. The CALL invocation transmits this information and leaves the process *waiting* for a reply in the same format. The RETURN invocation sends a response and renders the process *available* for further calls.[5] The SEND operation implements a send-only invocation. The sender remains running after the send completes; no response is expected. SEND can be used to implement non-blocking return, or to initiate parallel execution.

4.2 Numbers

A **number capability** is a capability holding 96 bits of data. Number capabilities serve several roles:

[5]The RETURN operation implements "return and wait for next call."

number	a self-describing 96 bit quantity.
page	the basic unit of storage for user data.
capability page	a unit of storage for capabilities.
node	an alternative unit of storage for capabilities.
address space	a mapping from offsets to pages
process	the container for a computation, and also the unit of protection domain.

Table 1: EROS Object Types

- The zero number capability is used in all places where a "void capability" might be expected.
- The zero number capability serves to identify invalid address ranges.
- Number capabilities are used to hold process register values when a process is not executing.

The only operation supported by a number capability is the operation that obtains it's value.

4.3 Basic Units of Storage

The basic units of storage in the EROS architecture are the data page (**page**) and the capability page (**node**). A node contains 32 capabilities, in the same way that a page holds 4096 bytes (your architecture may vary). Capability pages cannot be mapped into a process address space or otherwise manipulated directly by the application. This partitioning ensures that they are unforgeable.

Pages and nodes are named by **page capabilities** and **node capabilities**, which may carry a "read-only" restriction. Node capabilities may in addition carry "no-call" and "weak" restrictions. The "no-call" restriction suppress the invocation of untrusted address space exception handlers (See Section 4.7). Any capability fetched via a "weak" capability is weakened according to the rules shown in Table 2. The combination of weak and read-only restrictions therefore guarantees *transitive* read-only access.

Input	**output**
NumberCap	NumberCap
PageCap	RO PageCap
NodeCap	RO, NC, WK NodeCap
SpaceCap	RO, NC, WK SpaceCap
all others	NumberCap(0)

Table 2: The weaken operation

Using pages and nodes as the two fundamental units of storage, we construct the remaining abstractions needed for a general purpose, extensible operating system.

4.4 Address Spaces

An **address space** is a tree of nodes whose leaves are pages (Figure 1). Unmapped pages are indicated by a **Number Capability** holding the value zero. If set, the read-only bit in each capability indicates that the associated subtree cannot be written in this address space; sense capabilities are considered read-only.

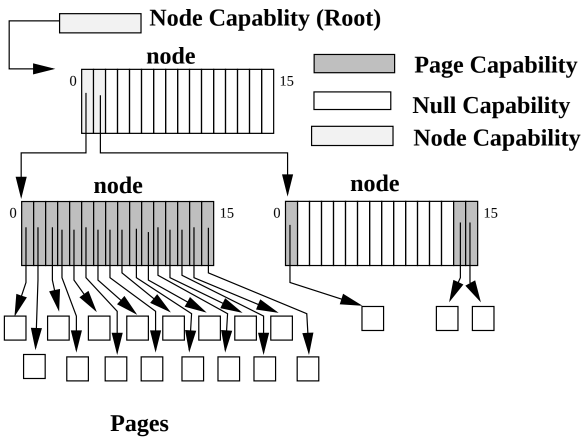

Figure 1: A 19 page address space

Address spaces are recursively defined. One process can construct a new address space that maps a subset of its current address space and pass this new address space to another party. The recipient can then merge the transfered address space into its main address space, establishing a shared mapping.

It is sometimes useful for the internal structure of an address space to be opaque to its user. To facilitate this, we introduced a specialization of a node capability known as an **address space capability**. Node capabilities and address space capabilities can be used interchangeably in defining the address space tree. The difference between an address space capability and a node capability is that the address space capability is opaque: slots of the named node cannot be examined by means of an address space capability. Most address spaces are constructed exclusively from node, page, and null capabilities.

For page, node, and address space capabilities, the $CapInfo$ field contains the **read-only**, **no-call** (suppresses fault handler invocation), and **blss** information about the object. BLSS (biased log space size) is defined as

$$BLSS \equiv log_2(Address\ Space\ Size) - 1$$

and defines the size of the subspace defined by a given page, node, or address space capability. Recording the size in the capability allows the address space fabricator to short-circuit levels in the mapping tree. One consequence is that most mapping trees for native EROS domains are only two or three nodes tall, reducing the number of object faults necessary to resolve an address.

4.5 Processes

A process is constructed as a suitable arrangement of nodes (Figure 2). Every process has a node that acts

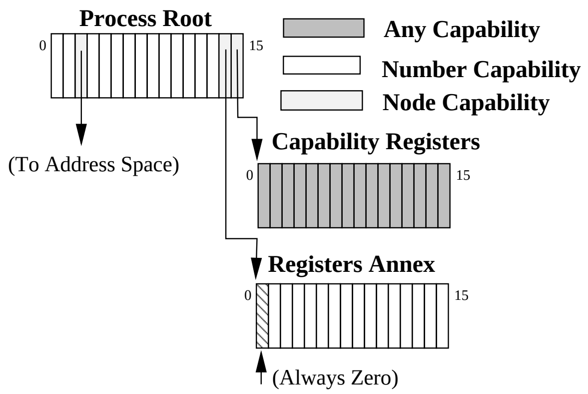

Figure 2: An EROS Process

as the root of the process structure, an additional node that holds the (kernel-implemented) capability registers for that process, and some architecture dependent number of annex register nodes that hold the register values of the process when it is not running. These register values are stored in **number capabilities**.

While node capabilities often serve as address space capabilities, they are not suitable for use as process capabilities. The capabilities in a process node must be of specific types for the process to be *well-formed.*[6] In addition, there is a reserved slot in the process root known as the *brand.* The brand is part of the EROS confinement mechanism, and must not be visible or modifiable by manipulators of the process.

A **process capability** conveys the authority to start and stop a process, to examine its registers (both data and capability), to alter or replace its address space, to change its scheduling authority, or to alter its fault handler. This enables the holder of a process capability to alter essentially anything about the process; its advantage over a node capability is that it enforces the capability type restrictions for well-formed processes and protects the brand.

4.6 User-Level Objects

Just as processes are named by process capabilities, the programs embodied within these processes are named by **start capabilities**. Performing a CALL on a capability causes the supervisor to fabricate a **resume capability** to the caller. This capability is provided to the recipient, and conveys the right to reply to this call. As with all other capabilities, the resume capability can be transferred or copied. All copies of a resume capability are efficiently invalidated when any of them is invoked, which ensures that every CALL receives at most one return.

The $CapInfo$ field of the start capability is set when it is fabricated, and is provided to the recipient by the kernel whenever the recipient is invoked. This allows the process to provide different interfaces to different callers, and can be used to distinguish service versions, service classes (e.g. administrator v/s user), or distinct "objects" implemented by the process.

Invoking a start or resume capability to a badly-formed process behaves as though one had invoked the zero number key. All invocations of the zero number key return a result code that (by convention) is not used by any other object.

[6]The *kernel* is not jeopardized by malformed processes. Such processes do not run, but the effects of invoking one are well-defined.

4.7 Exceptions

All exceptions taken by a process are redirected by the kernel to a process known as a *keeper*. Both address spaces and processes can designate a keeper by placing a start capability in the appropriate slot of the address space or process. When an exception occurs, the kernel synthesizes an upcall to the keeper process with a message describing the exception conditions. Memory exceptions are directed to the appropriate address space keeper, or if none is defined, to the process keeper. All other exceptions are reported to the process keeper.

5 The Store

Much of the benefit of the EROS system derives from careful organization of the store and an efficient consistent snapshot mechanism.

5.1 Organization of the Store

All objects in EROS can be reduced to node and page objects. Every page and node is uniquely identified by an **object identifier** (OID) which encodes its location in the store. The store itself is managed as a set of (possibly duplexed) ranges of consecutively numbered pages and nodes (Figure 3). For simplicity

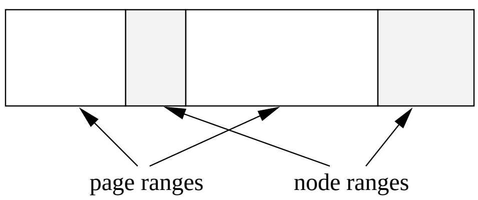

Figure 3: Disk Ranges

of I/O management, nodes are clustered into page-size units for purposes of storage on the disk.

On startup, the kernel scans all attached disks to load their range tables, and constructs an in-core master table of object ranges. Each range table entry is of the form

$$\{page, node\} \times [Start\ OID, End\ OID)$$

The in-core table is relatively small, and is kept sorted by starting OID.

When a capability for an object is first invoked, the object must be brought in from the disk. To do so, the kernel consults the master range table to find the physical location of the containing range in the store, computes the offset within that range of the disk page containing the object, and initiates a disk read operation. Once fetched, the object resides in an in-memory object cache, and the capability is *prepared* by modifying it to point directly to the in-core object. To facilitate capability depreparation on object removal, the capability is placed in a doubly-linked list whose head is the object itself. The preparation and depreparation of capabilities is entirely transparent to application code.

This storage organization has several advantages, notably in the absence of any "indirection blocks". In a system having a 2 terabyte store of 1000 2 gigabyte disks, each having 64 ranges (an unusually expensive configuration), the total overhead to locate the containing disk page is 16 *in-memory* comparisons to traverse the sorted range table. This multi-terabyte system an in-core table overhead of 1.2 megabytes, which is considerably less than would be used by an equivalent number of file systems.[7]

5.2 The Checkpoint Mechanism

In addition to page and and node ranges, the store also contains **log ranges**. Before an object in the main memory cache can be modified, space is reserved for it within the log range. Objects within the log are not stored in OID order; the log contains overhead pages that provide a directory of objects in the log. This directory is not referenced during normal operation, as a complete directory of the log is maintained in memory.

On a periodic basis, a dedicated service process initiates a **checkpoint** operation. The checkpoint operation stabilizes a systemwide consistent image via a three phase process: snapshot, stabilization, and migration.

Snapshot The snapshot phase efficiently creates a consistent snapshot of the entire system:

1. All dirty objects are marked copy on write.
2. All page table entries are marked read-only.
3. The thread list is traversed, and the in-core checkpoint directory entry for the root node of each active process is updated to reflect the fact

[7] One of our early users has an application that generates 14 terabytes of new data per year, all of which must be online for periods of many years.

Case	Sup I-refs	Sup D-refs	Sup ITLB miss	Sup DTLB miss	Sup cycles	S+U time
Null IPC	28.6	58.14	0	1	206	2.485 μS
14 byte IPC	36.51	82.04	1.5	1.5	333.64	3.68 μS
Page Fault	???	???	???	???	???	?~3 μS

Table 3: Supervisor footprint – p120

that the process was running at the time of the checkpoint.

Snapshot is a synchronous operation. The current implementation takes comfortably under 100ms on a slow i486, and can be shrunk by further exploitation of lazy marking to a few microseconds. Having constructed a consistent snapshot, execution is permitted to resume.

Stabilization Once a snapshot has been captured, the frozen objects are asynchronously written to the previously reserved space in the checkpoint log. When all objects frozen by the checkpoint have gone to the log, the checkpoint directory is written to the log. Finally, a new checkpoint header is written to indicate that the checkpoint has committed.

Stabilization does not violate the real-time requirement. The cost of stabilization is built in to the cost of dirtying objects; every object dirtied will cause at most two stabilization writes for objects of that type. These writes are asynchronous and non-blocking, provided that there is sufficient main memory and log space to allow a new object to be dirtied.

Migration After the checkpoint has committed, a **migrator** is started to copy the objects back to their official locations within page and node ranges. Because the migrator works on large collections of objects whose disk locations can be sorted, the current EROS migrator is capable of saturating the sustainable disk subsystem bandwidth on most machines. The principle difficulty in migration is slowing it down enough to use the machine for other purposes (*Yes, this is hyperbole. It will be replaced by real numbers when we have them in a few weeks*).

It is rare for a checkpoint to occupy more than 30% of the log before committing. No single checkpoint is permitted to occupy more than 70% of the available log space. Allowing this much of the log to be committed to a single checkpoint serves to dampen jitter in page dirty rates across checkpoints. Restricting the checkpoints ensures that a stable rate of two-stage migration throughput is achieved for large object mutations.

6 Reserves

In addition to capabilities for real resources, EROS implements capabilities for consumable resources and resource virtualization in the form of processor capacity reserves and working set reserves.

6.1 Processor Capacity Reserves

A **processor capacity reserve** is a 6-tuple of the form: (period, duration, quanta, start time, active priority, inactive priority). The *duration* specifies the number of guaranteed nanoseconds of computation within each *period*, beginning at the given *start time* (immediately if the start time is zero). The *quanta* specifies how often a process running under this reserve will be preempted in favor of other ready processes under the same reserve. The *active priority* gives the priority of this process during its reserved period. The *inactive priority* indicates the priority (if any) with which the process runs when its period has been depleted. Reserves whose inactive priority is below that of the kernel idle thread (which is always ready to run) will not execute outside of their reserved duration.

Processor reserves are allocated by an user-level agent known as the **reserve manager**. The reserve manager accepts requests for reserves, determines if those requests can be met without violating previous reservations, and if so, returns a capability to an appropriate kernel reserve. This reserve capability can be placed in the schedule slot of a process's root node; all processes sharing a given reserve capability will run under the named reserve.

In contrast to previous implementations [Kit93, Mer93], Processor reserves are *not* inherited across IPC operations. Priority inheritance raises a host of protection domain violation issues – if reserves are inherited, a caller might call a service and then shut off its reserve, denying service to other callers. Reserve inheritance lends itself to "QoS cross-talk," [Les96] (which manifests as variance), and understanding the behavior of nested server calls in the face of priority inheritance is difficult. Instead, EROS enables each application to construct a single end-to-end reservation by instancing its services under whatever processor reserve is appropriate.

The EROS CPU reserve implementation differs from previous implementations by unifying reserves and priority management. Depending on the selected admission control mechanism, reserves can be used to provide isochronous scheduling guarantees or softer periodic service at high priority. We have found it occasionally useful to run infrequently executed, unreserved processes at higher priority than the reserved processes, most notably those processes associated with system recovery.

6.2 Working Set Reserves

The capacity to use the processor is of little benefit if the application is not in memory. To ensure that performance-critical application code and data remains resident, EROS provides **working set reserves**. A working set describes the number of total pages and nodes and the number of dirty pages and nodes permitted to the working set reserve. Working sets are subdivisible; an application can fabricate an empty working set and transfer reservation from some existing working set to the new one. A working set may also be marked *non-persistent*, exempting any data in it from the checkpoint mechanism.

Typical applications define a single working set for their process. Applications may also specify an address subspace that operates under its own working set. This enables protocol modules to declare working storage for in-transit packets to be non-persistent, reducing the overhead of restart after checkpoint and reducing the complexity of orderly cleanup after a system restart has occurred.

7 Mapping to Hardware

While pages and nodes are an elegant way of specifying a system, they are not an especially good way of *running* that system efficiently.

Addressing-related capabilities translate directly into hardware page table entries; the associated access rights checks are implemented by the native hardware addressing mechanism. The process and the address space abstractions, however, must be translated into forms that can be efficiently used by the hardware. This is accomplished by two caches: the process cache and the mapping cache. The management of these caches is presented in detail elsewhere [cite POS paper], it is sketched here to facilitate understanding of the benchmarks section.

To facilitate efficient context switching, the state of actively executing processes is loaded into a **process cache**. Each active entry in the process cache contains the complete architected register set of some process. The layout of a process cache entry is optimized for context unload and reload rather than specification convenience. Once loaded into the process cache, a process remains cached until forced out by a more active process. In practice, most processes block most of the time; the process cache is usually able to contain the entire active process set.

The **mapping cache** holds hardware-specific mapping table entries. On tree-structured mapping hardware, it contains page tables. For hash-structured mapping systems, we implement a large second level translation buffer in software and fall back to walking the address space tree.

Hardware-specific mapping structures are constructed by traversing the address space structure and updating the appropriate entry or entries in the mapping cache. To ensure that the mapping entries are properly updated when the address space node slots are altered, a **dependency table** is used to keep track of the projection from the abstract address space to the hardware mapping tables.

8 Proposed Evaluation Method

In the absence of a substantial native application base, research microkernels have been evaluated by comparing the performance of a familiar environment (such as UNIX) to its rehosted equivalent on top of the microkernel. This is an inappropriate evaluation method for EROS for several reasons:

- EROS is designed to support mission-critical applications that demand high availability, hands-off 24/7 operation, very large stores and/or strong fault containment. These are applications for which UNIX and similar systems are inappropriate.

- EROS is designed for applications that place the system under high stress, such as online transaction processing. Many of the simplifying assumptions that are essential to making UNIX fast (e.g. statistically justified overcommitment of swap space) are inappropriate for such an environment.

- Operating systems are usually rehosted for the sake of legacy application support. A nominal (10% or less) degradation of performance is acceptable for such applications.

- Benchmarking rehosted environments does not expose any of the advantages of the new system. If what you want to do is run UNIX, you should run UNIX.

Equally important, both our own past experience and that of others suggests that this evaluation strategy does not lend itself to a correct understanding of the performance of the respective systems [Che93, Bom92].[8]

8.1 Issues in Comparison

Three differences in semantics between UNIX and EROS make direct comparisons of certain operations difficult: accountability, persistence, and the absence of the *fork*() primitive. Neither of these features has a direct equivalent in conventional systems, and each has associated costs and benefits.

Accountability The EROS architecture requires (as a matter of policy) that all storage be provided by the application, and that every piece of application-provided storage have real backing store. This adds to the cost of demand-zero memory regions, as the metadata and content structures (nodes and pages respectively) must be explicitly purchased by some user level application. No analogous accountability is applied by UNIX and similar systems, since the analogous storage is fabricated from a statistically overcommitted resource (the swap area). Transparent persistence of an overcommitted swap area cannot be recovered by restarting.

Persistence By design, EROS has no analogue to the UNIX *sync* operation. Under normal circumstances, applications build structures in memory and rely on the underlying checkpoint mechanism to render them persistent. A separate, fine-grain mechanism exists for use by transactioned facilities.

The overhead associated with file system metadata is substantial and well explored [Gan94, Sel95]. Ultimately, this overhead derives from the fact the runtime state (including that of the file system implementation) is not persistent, and therefore cannot be relied on to resume from a consistent state. The use of a transparent system-wide checkpoint operation removes this impediment, which eliminates the need for transactioned metadata update. Explicit metadata management is relegated to those exceptional applications such as databases that have unusual promptness requirements.

Since this reflects the normal behavior of the system (i.e. what the user will actually see), we view it as a realistic (albeit unfair) comparison. The reliability tradeoffs inherent in this policy are discussed in Section ??.

fork UNIX systems place heavy reliance on the *fork*() primitive, which has no analogous operation in EROS. In benchmarks, and also in real systems, the *fork* call is nearly always followed immediately by *exec*. The end result is the creation of a new process with a new address space. The metric of interest, then, is the *create process* operation. This is the one that should be benchmarked.

8.2 Benchmarks

The performance of applications is dependent on three factors: memory speed and configuration, processor speed, and a weighted measure of the performance of certain critical services:

1. Address space mapping management,
2. Page fault handling,
3. Reading and writing data to the store (file operations), and
4. Interprocess communication performance,

Processor speed and memory subsystem are independent of the choice of operating system. Critical service operations can be compared directly by microbenchmarks. The lmbench benchmark suite does a reasonably effective job of measuring such low-level operations [McV93]. We therefore propose to adapt the lmbench microbenchmark suite to the EROS platform for measurement purposes.

[8] The conclusions of Bershad and Chen in [Che93] are mistaken. Simply reversing the order of user and supervisor memory delays in Figure 2-1 makes it apparent that the operating system structure had essentially *no* effect on application behavior. The graph actually reveals the poor design and architecture of Mach 3.0 and the inefficiency of its IPC mechanism, and strongly suggests that a better microkernel architecture should exceed the performance of the native UNIX system.

For each benchmark, we propose to measure both CPU cycles and cache misses in the instruction, data, and TLB caches. The latter measurements expose deferred post-service overheads that would not otherwise be demonstrated by a microbenchmark.

9 Related Work

This section needs to be expanded.

Most of the early hardware protection mechanisms were capability based [cite Plessey, Sigma-7, C.mmp, B5700], but these systems failed to deliver acceptable performance. The dramatic and highly publicized failure of the Intel i432 led mainstream computer architects to abandon fine grain capability architectures. The Intel/Siemens BiiN architecture (better known as the i960) demonstrated that hardware-based capability systems could be high-performance, but went undeployed due to a contractual breakdown between the principals. The IBM AS/400 is the only widely deployed capability system in use today.

9.1 KeyKOS

KeyKOS is an earlier capability system from which the EROS architecture is derived [Har85]. Originally constructed for the IBM 370, and later ported to the Motorola 88000 family, KeyKOS delivered performance rougly equivalent to that of Mach 2.5 [Bom92]. In addition to formal underpinnings described elsewhere [Sha97], EROS brings the performance of the architecture into line with that of more aggressive systems such as L4.

9.2 Mach 4.0

Bryan Ford has given considerabley attention to the problem of IPC performance in Mach, and has shown that it can be substantially improved MACH4:Migrating. For dekernelized systems, including EROS, IPC operations are a performance critical benchmark. Various elements of the Mach messaging semantics restrict the realizable performance.

9.3 L3 and L4

L3 L3:IPC,L3:SpaceMux, and later L4, have established new standards for interprocess communication times. The work already done on EROS has demonstrated that these times can be matched in the EROS context [Sha96d]. The principle insight to draw from this work, in our opinion, is that context switch and exception handling have much more to do with the hardware architcture than with the specifics of any particular operating system architecture.

10 Conclusions

While there are many ways to construct capability systems, very few are likely to be efficient. EROS achieves high performance by use of a design methodology comprising the steps of:

1. Identify minimal sets of primitive memory objects and operations, which are required to provide system semantics and cannot be constructed from even more primitive objects ans services.
2. Examine these sets in light of architectural support and microbenchmarks to select an efficient *basis* from which the capability system can constructed.
3. Use the hardware performance (e.g., for TLB table updates or disk throughput) as the design target for performance of each of the primitives.

The critical path operations of the EROS system meet or substantially exceed the performance of all other current operating systems. Substantial further improvement on the critical paths is possible only by changes to the underlying hardware; the two most important changes are a fast privilege boundary crossing mechanism and a tagged or software-managed TLB architecture.

The key point to take away from this paper is that capability systems are *not* slower than "conventional" operating systems. EROS provides a clear existence proof, operating at or near the performance limits imposed by the underlying hardware. The performance presented here is achieved by an implementation constructed in a high-level, portable, optimization-challenged source language (C++), augmented by less than 1500 lines of assembly code, which including drivers, compiles to less than 160 kilobytes of code.

Further information on the EROS system can be found via our web page at `http://www.cis.upenn.edu/~eros`.

References

[Bom92] Allen C. Bomberger, A. Peri Frantz, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, Jonathan S. Shapiro. "The KeyKOS NanoKernel Architecture," *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures.* USENIX Association. April 1992. pp 95-112.

[Che93] J. Bradley Chen and Brian N. Bershad. "The Impact of Operating System Structure on Memory System Performance" *Proc. 14th SOSP*, December 1993.

[Col88] Robert P. Colwell, Edward F. Gehringer, E. Douglas Jensen, "Performance Effects of Architectural Complexity in the Intel 432." *ACM Transactions on Computer Systems*, **6**(3), August 1988, pp. 296–339.

[Den66] J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations" in *Communications of the ACM*, vol. 9, pp. 143–154, March 1966.

[For94] Bryan Ford and Jay Lepreau, "Evolving Mach 3.0 to a Migrating Threads Model," *Proceedings of the Winter USENIX Conference*, January 1994. `ftp://mancos.cs.utah.edu/papers/thread-migrate.ps.Z`

[Gan94] Gregory R. Ganger and Yale N. Patt. "Metadata Update Performance in File Systems" in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, Nov. 1994, pp. 49-60.

[Har85] Norman Hardy. "The KeyKOS Architecture" *Operating Systems Review.* Oct. 1985, pp 8-25.

[Key86] Key Logic, Inc. *U.S. Patent 4,584,639: Computer Security System.*

[Kit93] Takuro Kitayama, Tatsuo Nakajima, Hiroshi Arakawa, and Hideyuki Tokuda. "Integrated Management of Priority Inversion in Real-Time Mach." *IEEE Real-Time Systems Symposium.* December 1993

[Lam73] Butler W. Lampson. "A Note on the Confinement Problem." *Communications of the ACM* Vol 16, No 10, 1973

[Lam76] Butler W. Lampson and Howard E. Sturgis. "Reflections on an Operating System Design." *Communications of the ACM* Vol 19, No 5, May 1976.

[Lan92] Charles R. Landau. "The Checkpoint Mechanism in KeyKOS," *Proceedings of the Second International Workshop on Object Orientation in Operating Systems.* IEEE. September 1992. pp 86-91.

[Les96] *Need citation here*

[Lie93] Jochen Liedtke. "Improving IPC by Kernel Design." *Proceedings of the 14th ACM Symposium on Operating System Principles*, ACM 1993.

[Lie95] Jochen Liedtke. *Improved Address-Space Switching on Pentium Processors by Transparently Multiplexing User Address Spaces*, GMD TR 933, November 1995.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems.* Digital Press. 1984.

[Lyc78] H. Lycklama and D. L. Bayer, "The MERT Operating System," *Bell System Technical Journal*, **57**(6, part 2), pp. 2049–2086, July/August 1978.

[McV93] Larry McVoy, and C. Staelin. "lmbench: Portable Tools for Performance Analysus" in *Proceedings of the 1196 USENIX Technical Conference.* San Diego, CA, January 1996, pp. 279-295.

[Mer93] 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*, October 1993.

[Org73] "Computer System Organization: The B5700/B6700 Series," Academic Press, 1973.

[Sal75] Jerome H. Saltzer and Michael D. Schroeder. "The Protection of Information in Computer Systems," in *Proceedings of the IEEE*, **63**(9), Sept. 1975, pp. 1278–1308.

[Sel95] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan, "File System Logging versus Clustering: A Performance Comparison", *Proceedings of the 1995 USENIX Technical Conference*, January 1995, New Orleans, LA, pp. 249–264.

[Sel95] Margo Seltzer, Yasuhiro Endo, Christopher Small, Keith Smith, "Dealing with

Disaster: Surviving Misbehaved Kernel Extensions" in *Proceedings of the 1996 Symposium on Operating System Design and Implementation.*

[Tho78] K. Thompson, "UNIX Implementation," *Bell System Technical Journal*, **57**(6, part 2), pp. 1931–1946, July/August 1978.

[Sha96a] Jonathan S. Shapiro. *A Programmer's Introduction to EROS.* Available via the EROS home page at `http://www.cis.upenn.edu/~eros`

[Sha96b] Jonathan S. Shapiro. *The EROS Object Reference Manual.* In progress. Draft available via the EROS home page at `http://www.cis.upenn.edu/~eros`

[Sha96c] Jonathan S. Shapiro, David J. Farber, and Jonathan M. Smith. "State Caching in the EROS Kernel – Implementing Efficient Orthogonal Persistence in a Pure Capability System", *Proceedings of the 7th International Workshop on Persistent Object Systems*, Cape May, N.J. 1996

[Sha96d] Jonathan S. Shapiro, David J. Farber, Jonathan M. Smith. "The Measured Performance of a Fast Local IPC". *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems*, Seattle, Washington, November 1996, IEEE

[Sha97] Jonathan S. Shapiro and Sam Weber. *Verifying Operating System Security.* Department of Computer and Information Science Technical Report MS-CIS-97-26, Forthcoming.

[Wul81] William A. Wulf, Roy Levin, and Samuel P. Harbison. *HYDRA/C.mmp: An Experimental Computer System* McGraw Hill, 1981.