Managing Capabilities by Dynamic Single Assignment

SRL Technical Report SRL-2000-04

Jonathan S. Shapiro
Johns Hopkins University
Baltimore, MD 21218
shap@eros-os.org

DRAFT of December 7, 2000

Abstract

Performance in capability-based systems is heavily influenced by the cost of capability copies. In a well-optimized implementation, as much as 40% of the null IPC time is due compulsory cache misses associated with these copies. Both reference counting and chained descriptor implementations incur such cache misses. The third common design option, garbage collection, is incompatible with prompt object reclamation, which is a requirement in high-assurance systems.

This paper introduces *dynamic single assignment*, a new mechanism for capability management that eliminates that eliminates these compulsory cache misses without introducing the need for global garbage collection. The new mechanism favors interrupt-style kernels by eliminating both undo overheads and aliased in/out capability parameters to kernel services. In addition to IPC performance improvements, the technique yields significant simplifications in the most complicated paths of the EROS kernel while simultaneously reducing the number of data-dependent control flow decisions in that path.

This paper is a snapshot of a work in progress. Performance claims capture my expectations, but do NOT reflect actual measurements.

1 Introduction

EROS is a capability-based microkernel that incorporates a highly-tuned interprocess communication (IPC) mechanism [SSF99]. Because the most common design pattern in the system is "call with some arguments, return with a result code," nearly half of the interprocess communications (the returns) are "null IPC" operations – IPCs that transfer only register and descriptor arguments. In spite of careful implementation, capability copies account for over 40% of the cost of a null IPC operation. Nearly all of the gap between EROS IPC times and L4 IPC times [Lie93] lies in the cost of capability copies.

This overhead can ultimately be traced to our choice of capability representation. In order to support rapid revocation, EROS capabilities are placed in a doubly linked list rooted at their object. When a capability is copied, the target capability must first be unlinked and the newly created capability must be linked. As neighboring capabilities in the respective linked lists are almost never in *second* level data cache, capability copy causes a minimum of three cache misses. Since four capabilities are copied per invocation, the cost of servicing these misses is a significant component of total IPC cost.

Viable alternatives to the linked list representation are not easy to identify. Singly linked lists do not support efficient unlink. Reference counting carries two compulsory misses (one decrement, one increment) rather than three, and require a background sweep pass to prevent version overflow. Direct pointers do not support efficient revocation, Object tables require full garbage collection to reclaim OT entries. On first examination, incremental real-time collection strategies add prohibitive overhead to capability fabrication (including copies).

This paper introduces *dynamic single assignment*, a mechanism for capability management that eliminates compulsory cache misses for most capability copies while preserving prompt revocation. As a side effect, the mechanism eliminates much of the control flow complexity that arises in interrupt-style kernels from the need to implement operation atomicity. Under the new design, capability fabrications need not be undone when a kernel invocation blocks. While the resulting improvement in IPC performance is signficant, an equally valuable aspect of this paper may be its contribution to the space of design techniques for interrupt style kernels.

We have implemented dynamic single assignment in the EROS kernel, and found that null IPC times are improved by nearly 40%. In spite of the newly introduced overhead of incremental collection, *kernel* invocations are also improved (the percentage varies according to the invocation). Most of this improvement is due to the elimination of cache misses associated with capability copies. Some is due to the elimination of tightly data-dependent control flow in the IPC path and incidental changes in addressing mode selection. These in turn reduce register pressure in the IPC path.

The balance of this paper proceeds as follows. We first briefly present the requirements for a satisfactory capability representation. We describe the original EROS mechanism as a baseline solution. and identify the performance issue latent in the selected representation. We then describe a simplified form of dynamic single assignment. This description establishes the key idea, but must be augmented by several optimizations we describe before it fully meets the requirements. Having described the new mechanism, we identify several other complications in the original EROS implementation that are resolved by the new design. We compare the performance of old and new implementations and conclude.

2 Requirements

A satisfactory in-memory capability representation must meet a number of requirements:

1. The capability should provide efficient access to the object that it names. In practice, this tends to mean that the capability contains a pointer to the object. If necessary, this pointer may be indirected through an object table.
2. There must be an efficient mechanism to locate and revoke all outstanding capabilities when an object is reclaimed. There are two issues here:
   (a) The need to promptly revoke outstanding authorities to a reclaimed object.
   (b) The need to ensure that all outstanding pointers to the object's memory frame are removed before the frame can be reused for another object.
3. Copying capabilities is a frequent operation. It must be inexpensive.
4. Fabricating new capabilities should similarly be inexpensive for frequent cases.

2.1 The Old Solution

In this paper, we are concerned only with capabilities to in-memory objects. It is useful for all capabilities to be the same size, but the handling of object faults and capability format interconversion is outside the scope of this paper. For those interested, a discussion of object faulting and capability conversion are addressed elsewhere [SSF99, SFS96].

The original EROS (and KeyKOS [Har85]) implementation placed all in-memory capabilities for a given object in a doubly linked list rooted on the object. In addition, an in-memory capability contains a pointer directly to its object and a word describing the capability type and associated access rights. This representation meets requirements (1) and (2) and (3). When considered relative to the competing systems of the time, Mach [GDFR90] and Chorus [RAA$^+$91], the copy and fabrication performance was also acceptable.

Today's systems are somewhat more carefully architected for performance. Liedtke, in particular, has reported specialied IPC times in the range of 135 cycles [Lie95] on Pentium-class machines. Relative to this design, the linked capability strategy is no longer competative. The problem is that three neighbors must be updated (shaded in Figure 1), and that these neighbors are rarely found in either the first *or second* level cache. Given that a second level cache miss takes approximately 15–20 cycles to satisfy, it is easy to see that there is a performance problem here. Each capability copy will take 45–60 cycles from compulsory miss processing. EROS copies four capabilities per capability invocation.

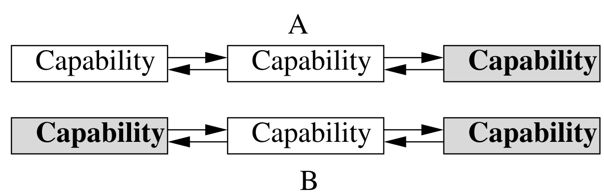

Figure 1: Chain updated misses.

We briefly experimented with an object-table based representation [SFS96], but concluded that garbage collection introduced more problems in the design than it solves.

2.2 Related Issues

There are other aspects of the EROS design that impact the choice of alternatives. While it is difficult to reduce this concern to a set of precisely stated requirements, it is desirable to find a solution that enhances – or at least does not penalize – an interrupt style kernel.

EROS capabilities may name either processes or kernel-implemented objects. When a kernel capability is invoked, capabilities are returned to the recipient from the kernel. In the usual case, the recipient and the caller are the same. The locations of the returned capabilities (i.e. the "registers" into which they are received) are determined by the recipient. Frequently, the recipient elects to accept the returned capabilities in a way that overwrites the argument capabilities.

For some common kernel invocations it is possible that the operation will block in some intermediate state (e.g. for an object fault) and that the caller will be forced to restart the request when the impediment to completion is removed. Because EROS is an interrupt-style kernel, the implementation is not permitted to overwrite recipient capability registers until completion is guaranteed. The effect of this is that new capabilities are fabricated into a scratchpad area and then copied to the recipient process as the operation completes. Maintainability dictates a uniform style of processing kernel requests. The end result is that all requests use the scratchpad, and all consequently suffer from capability copy overheads. The percentage varies considerably with the complexity of the operation, but it the most commonly invoked operations are (unsurprisingly) relatively fast, and these operations pay a higher proportional penalty.

3 Dynamic Single Assignment

Dynamic single assignment is an adaptation of register renaming [Log72] for software use. By way of introduction, we first review register renaming and then describe its adaptation for EROS.

3.1 Register Renaming

Register renaming originated on the IBM System 360/91 as a means to support internal parallelism in the floating point unit. The S360/91 floating point unit provided only four architected floating point registers. This design led to instruction issue stalls, as a new operation targeting a given register could not be issued until all uses of that register had successfully completed. Tomasulo's solution to this problem [Tom67], later simplified by Logrippo [Log72], is to provide a larger register file (known as the *physical registers*), and arrange that each assignment to a register proceed as follows:

1. Allocate an available physical register to be the destination of the assignment.
2. Rebind the targeted architectural register to point to this physical register.

The added level of indirection allows multiple operations to be speculatively issued. If an exception occurs, any rebindings of architected registers following the faulting instruction are undone. When a physical register ceases to be needed as a possible input to some future operation, the physical register is returned to the free list.

Several enabling conditions make it possible to adapt this design to software use in the EROS kernel:

- Every EROS process has a fixed number of software-implemented capability registers. Those familiar with other capability systems may find it more familiar to think of this as a fixed-size per-process C-list, but this "mental set" may not lead as readily to considering microprocessor implementation techniques as a source of software mechanisms.
- Nearly all capability copies target a capability register. Explicit capability stores to other locations are low frequency events. A solution that applies only to operations on capability registers should therefore significantly improve overall performance.
- Ignoring the reply authority, interprocess communication never fabricates new capabilities. The total pool of distinct values in all capability registers cannot *grow* as the result of an IPC operation.
- For each kernel invocation, there is a bounded bounded number of capabilities that can be created (in principle four, but in practice never more than two).

3.2 Software Adaptation

To adapt the Tomasulo design, we first isolate all of the software capability registers into a side table. Capabilities in the side table remain chained to their object by a doubly linked list. We replace the existing registers with pointers into this table [FIGURE NEEDED]. Since each process has 32 capability registers, the resulting table would ordinarily have $32N$ entries (where N is the number of processes). To support incremental reclamation of unused physical registers, we add an additional $M + 4$ slots. The 4 accounts for the maximum number of capabilities that can be produced by a kernel call.

To ensure that a slot in the capability register pool is available whenever a new capability is created, each slot allocation is forced to perform a fixed amount of progress on an incremental garbage collection of the capability register pool. The overhead of this collection is determined by the choice of M. If $M == N$, then each slot allocation must advance the "mark" pass of the garbage collection by one full process (i.e. by 32 registers). If $M == 2N$, only 16 registers must be swept, and so forth. In the implementation described here, we have chosen $M == 8N$.

3.3 Impact

With this rearrangement made, we can now implement capability copies by simply copying the *pointer* to the capability from one process to the other. This of course takes no unnecessary cache misses. As a further benefit, the reduction in size of the per-register data structure allows us to use scaled-index addressing modes on many processors. It also eliminates the need for format checks before performing the copy. There is one significant difference between this and Logrippo's method. Logrippo never allows more than one architected register to name a physical register, because aliasing is difficult to manage using only combinational logic. Our design relies on incremental garbage collection, and actively encourages aliasing as a means to reduce register pressure on the capability register pool.

In the EROS design, all capabilities fabricated by kernel invocations are returned to capability registers in some recipient. When a kernel invocation must fabricate a new capability, it allocates a new register slot from the capability register pool. The new capability is fabricated into this currently unreferenced slot. If the kernel invocation runs to completion, the capabilities are "copied" back to the recipient process by updating the pointers in the recipient process register pointer array. If the kernel invocation is not completed successfully, the value is simply left in the register pool unreferenced for subsequent reclamation.

4 Optimizations

While the scheme outlined so far is usable and a significant improvement on the current EROS kernel, there remains one bottleneck and one opportunity for significant improvement.

4.1 Resume Capabilities

Nearly 50% of all EROS capability invocations are CALL operations. These are IPC invocations that expect a reply. In order to allow the recipient to perform this reply, the CALL operation fabricates a special capability known as a ***resume capability***. In the design described to this point, every resume capability fabrication would require reclamation of $32N/M$ register pointers. Each of these marks entails a probable cache miss. Much of the anticipated benefit is lost in this overhead if these marks are required.

As a further complication, resume capabilities are unusual in that they are consumed on use. This is done to ensure that there is only one reply for each call. The kernel specification requires that "all copies of a resume capability are efficiently invalidated when any copy is invoked."

Our solution is to replace the capability register pointers with a two-word structure consisting of a pointer and an auxiliary word. If the capability in a given register is a resume capability, the pointer points directly to the destination task control block rather than to an entry in the capability table. [FIGURE] To eliminate the need for special recognition of this case, we arrange a representation pun.

In the original design, object data structures are arranged such that their first few words are the anchor of the capability link chain. The object pointer therefore also points implicitly to the link chain root. To avoid the need to special-case the unlink code, a representation pun is preserved: the anchor layout is "compatible" with the capability layout, by which we mean that the "next" and "previous" pointers appear at the same offsets in both the object header and the capability. In the revised design, most special-case treatment of resume capability pointers is eliminated by carrying the representation pun one step further: the first four words of a task control block now consist of a valid resume capability to that task control block. This template capability serves double duty as the head of the capability chain for this process, and is never revoked.

For resume capabilities, the auxiliary word is used as a counter to determine whether a given resume capability remains valid. The word *after* the template resume capability in the task control block is also a counter. If the counter in the capability register and the counter in the task control block match, the resume capability in question is still valid, otherwise, it conveys no authority.

A pleasant side benefit of this change is that the presence of the "template" capability eliminates the need to fabricate a resume capability as part of each CALL operation, which is another source of expense in the IPC path.

When an incremental mark pass is underway, resume capability pointers can be distinguished from other capability pointers by performing a range check on the pointer value. If the pointer falls outside the capability register pool, it is a resume capability. In practice, a more efficient test is possible: if the capability pointer points to the same location as the object pointer within the capability, it is a resume capability.

4.2 Type Conversions

A second common type of operation in the EROS system is capability type conversions. These operations include creating read-only capabilities from read-write capabilities, fabricating process and start capabilities from node capabilities, and so forth. These distinct capability types differ only in the the value assigned to the word that contains type and access rights. We take advantage of this by using the auxiliary register word to hold these fields. As a result, operations that create variants from capabilities that are already in registers and return them to a recipient process do not allocate new entries in the capability pool. We therefore do not require that such operations contribute to incremental reclamation of the capability register pool.

4.3 Other Cases

EROS uses a distinguished capability that serves as a capability conveying no authority. As a special case, this capability has a dedicated slot in the capability table and is never collected.

5 Costs

In choosing representations, it is frequently possible to choose data structures that move costs to low frequency paths. As Robert Heinlein so clearly puts it, however, "there ain't no such thing as a free lunch." It is perhaps useful to understand where the costs end up in a dynamic single assignment system, if only to understand what conditions must exist for the technique to be useful.

In essence, the dynamic single assignment mechanism reduces copy costs by increasing fabrication costs. The increase in fabrication costs can in turn be traded against an increase in storage costs. In our final implementation, new capability fabrication costs are reduced to a worst case of 4 cache misses by spending 25% more space on storage for capability registers. The further optimizations described in Section 4 ensure that these costs are incurred only for low-frequency cases. With these optimizations, a higher per-fabrication cost with lower storage overhead would probably be acceptable in practice. Two potentially significant operations, however, are penalized: object allocation and task control block allocation.

5.1 Object Allocation

When a new object is allocated, there exists no outstanding capability to that object. A new capability must be fabricated, and this fabrication incurs an incremental garbage collection overhead (four cache misses). In the EROS implementation, this cost is recovered almost immediately, because the next operation after fabricating a new object is invariably an IPC operation that copies the new capability to some other process, and the marginal cost of fabrication is made up in the reduced cost of the copy. Object allocation mechanisms that are not accompanied by a copy are penalized, and different system designs might be more or less sensitive to this performance impact.

5.2 TCB Allocation

EROS treats it's task array as a cache. This task control block cache contains the active subset of the processes in the system. In principle, the number of processes is bounded only by available disk space. When a process not already assigned to a task control block is activated, it must first be loaded. This requires that 32 new capabilities be allocated, and in consequence that $32N/M$ capability registers must be swept. The solution is to configure an appropriately sized process cache in order to reduce or eliminate task control block loads.

Choosing an appropriate number of task control blocks is a function of memory size and the expected number of active processes. It is certainly possible to configure a system in such a way that the process cache is inadequate, but under normal circumstances this does not occur. EROS currently runs with a default process cache of 2048 task control blocks. In practice, we have never seen *any* general-purpose computing machine that can handle this number of tasks without thrashing, and the marginal cost of the process cache loads would not be the first source of concern in such systems.

6 Performance

7 Related Work

Register Rename (Tomasulo, RS/6000).

8 Acknowledgements

9 Conclusion

References

[GDFR90] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. UNIX as an application program. In *Proc. USENIX Summer Conference*, pages 87–96, June 1990.

[Har85] Norman Hardy. The KeyKOS architecture. *Operating Systems Review*, pages 8–25, October 1985.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proc. 14th ACM Symposium on Operating System Principles*, pages 175–188. ACM, 1993.

[Lie95] Jochen Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Technical Report GMD TR 933, GMD, November 1995.

[Log72] L. Logrippo. Renaming in program schemas. In *Proc. IEEE 13th Annual Symposium on Switching and Automata Theory*, pages 67–70. IEEE, October 1972.

[RAA+91] 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.

[SFS96] 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. In *Proc. 7th International Workshop on Persistent Object Systems*, pages 88–100, Cape May, NJ, USA, 1996.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Kiawah Island Resort, near Charleston, SC, USA, December 1999. ACM.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploring multiple arithmetic units. In *IBM Journal of Research and Development*, volume 11, pages 25–33. IBM, January 1967.