A Memory Allocation Model For An Embedded Microkernel

Dhammika Elkaduwe Philip Derrin Kevin Elphinstone

National ICT Australia[*] and University of New South Wales
Sydney, Australia
firstname.lastname@nicta.com.au

Abstract

High-end embedded systems featuring millions of lines of code, with varying degrees of assurance, are becoming commonplace. These devices are typically expected to meet diverse application requirements within tight resource budgets. Their growing complexity makes it increasingly difficult to ensure that they are secure and robust.

One approach is to provide strong guarantees of isolation between components — thereby ensuring that the effects of any misbehaviour are confined to the misbehaving component. This paper focuses on an aspect of the system's behaviour that is critical to any such guarantee: management of physical memory resources.

In this paper, we present a secure physical memory management model that gives hard guarantees on physical memory consumption. The model dictates the in-kernel mechanisms for allocation, however the allocation policy is implemented outside the kernel. We also argue that exporting allocation to user-level provides the flexibility necessary to implement the diverse resource management policies needed in embedded systems, while retaining the high-assurance properties of a formally verified kernel.

1 Introduction

Embedded systems are becoming increasingly complex. High-end devices, such as mobile phones, PDAs, entertainment devices, and set-top boxes, feature millions of lines of code with varying degrees of assurance of correctness. These devices are no longer closed systems under control of the manufacturer. They feature third-party components, applications, and even whole operating systems (such as Linux) that can be installed by the manufacturer, suppliers and even the end user. When constructing such devices using traditional unprotected real-time executives, it becomes impossible for embedded system vendors to provide guarantees about the behaviour of the device. Failure or malicious behaviour of a single software component on the device will affect the whole device.

One approach to improving the security and robustness of components on a device is to provide strong isolation guarantees between components — misbehaviour of a component is confined within the component itself. There are many approaches to isolation guarantees, such as classical processes and virtual memory [Fot61], isolation kernels [WSG02], and virtual machines [Wal02], which all provide varying levels of isolation guarantees at different granularity. Ideally, when required, isolation should be at the level of *partitioning* as defined by [Rus99]:

> A partitioned system should provide fault containment equivalent to an idealised system in which each partition is allocated an independent processor and associated peripherals and all inter-partition communications are carried on dedicated lines.

This paper focuses on one aspect of providing isolation guarantees closer to that of partitioning — the management of physical memory on the device. Specifically, the mechanisms used to directly and indirectly control access to physical memory while providing services to software components on the system. Note that we not arguing that partitioning is the most appropriate policy for embedded systems. Our goal is a set of kernel mechanisms that enable system services, where those mechanisms respect and can enforce a domain-specific system allocation policy, where that policy may include partitioning of memory at one extreme, and first-come first-served at the other.

The problem is more complex than simply controlling the size of virtual memory, or resident set size of an application. Services such as pages or threads not only require allocation of memory to directly support the service (a frame or thread control block), service provision also results in the allocation of kernel meta-data to implement the service (such as page tables) or provide the bookkeeping required to reclaim the storage on release. Kernel meta-data must taken into account in any system attempting to provide memory allocation guarantees.

We can summarise our approach to tackling the meta-data issue as simply eliminate all meta-data in the ker-

[*]National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

nel. We achieve this by the following three techniques: careful avoidance of mechanisms requiring bookkeeping, pre-allocation of bookkeeping required within explicitly allocated kernel objects, and the promotion of meta-data into first class kernel objects. The approach reduces the problem of memory allocation to kernel object allocation. We also present a model for the distribution of physical memory, and the creation of kernel objects within that memory.

Kernel services and complexity have a direct consequence on our ability to successfully implement our approach. Thus our overall system design is that of a microkernel-based system, where the microkernel aims to provide a minimal, efficient, flexible kernel with the strong guarantees needed for the foundation of the system. Higher-level system services are provided by user-level servers, running outside the kernel, in their own protection domains. Protection domains and their associated low-level resources are strictly managed by the microkernel. Normal applications access system services by interacting with the servers via interprocess communication. Such a system is not only more amenable to applying our approach to the core kernel, the overall system has the potential to be more robust, as faults are isolated within servers and applications; it is are also flexible and extensible, allowing user-level servers to be added, removed or replaced.

In the remainder of the paper, we discuss the requirements and issues surrounding the construction of complex embedded systems that consequently motivated our approach and influenced its design (Section 2). In Section 3, we then overview our microkernel *seL4* (secure embedded L4), and describe in detail its in-kernel memory allocation model. Finally, Section 4 discusses related work, and conclusions and future work are in Section 5.

2 Memory Allocation

Memory allocation to support kernel services and associated meta-data can have a direct or indirect effect on the security, real-time properties, efficiency, and assurance of the overall system. The following sections examine describe issues and requirements of a memory allocation mechanism in each of these areas.

2.1 Memory Utilisation

Physical memory is a limited and exhaustible resource. Any limited resource requires precisely controlled allocation to avoid one task's requests for authorised services from directly or indirectly denying service to another task. Simple per-task quota-based schemes or static preallocation would suffice in a statically structured system. However, any dynamic variation in system structure or resource requirements leads to underutilisation, due to the overly conservative commitment of resources required to ensure all authorised service requests are satisfied during peak demand.

An illustrative example of this utilisation issue in practice are virtual machine monitors. Guest operating systems are an ideal candidate for a fixed preallocated amount of physical memory (in fact guest OSes usually assume it). However, significantly higher efficiency can be achieved if memory can be safely reassigned to where it can be utilised [Wal02]. Memory allocation mechanisms must support dynamic allocation and re-assignment of memory.

2.2 Security

To avoid the denial-of-service attack described in the previous section, the memory allocation mechanism must be able to enforce a desired physical memory allocation policy. Additionally, the mechanism must not have by-design overt storage channels that can be used to violate confinement guarantees.

2.3 Real-time

Real-time behaviour is an important issue in the context of embedded systems. The main issue that arises with memory allocation in the real-time context is predictability of execution times of kernel operations. Predictable execution times are a prerequisite for schedulability analysis. Memory allocation affects predictability when physical memory caching is used to avoid kernel memory starvation, be it implemented with virtual memory or managed explicitly. Several operating systems use kernel physical memory as a cache of data structures stored at user-level or on disk [SSF99,CD94], and thus can always evict cache content to service new requests to avoid physical memory-based denial of service. However, such a strategy is not readily amenable to execution time analysis, and if it was, would result in too pessimistic an estimate to be useful. A memory allocation mechanism must be able to guarantee allocations to real-time components.

The structure of bookkeeping in a kernel managing allocation also affects predictability of interrupt or event latencies. Traversal of lists or trees can result in varying or unreasonably long executing times for kernel operations traversing the list. Ideally, all system calls would complete in constant time, or at least be preemptable to minimise interrupt latency.

CPU cache colouring techniques for improving predictable cache behaviour are also dependent on control of the memory allocation of the data structures requiring colouring [LHH97], including the kernel's internal data structures.

2.4 No Single Policy

Given the wide variety of potential application domains, we expect no single kernel memory allocation policy to suffice in all situations. At one extreme, we must support simple static systems where the main requirement is spatial partitioning of components of the system [Rus99]. For kernel memory allocation, this implies a guaranteed fixed allocation of physical memory (including meta-data) to each partition that cannot be interfered with by any activities of any partition. In the other extreme, we expect to support best-effort embedded operating systems where physical memory management is dynamic, on-demand, and uncritical.

A realistic example of the latter extreme is in efficiently supporting para-virtualised legacy operating systems on the microkernel [BDF+03]. The legacy OS is in the best position to determine the allocation of page tables, pages, frames, and thread control blocks. This scenario is just a specialised case of the more general argument that application self-management of resources can lead to more efficient use of those resources [Han99, EGK95].

In addition to supporting a specific policy suitable for a particular application domain, we expect the kernel to potentially support several kernel physical memory allocation policies concurrently. A realistic example is a system providing one or more critical partitions, while at the same time, efficiently supporting an best-effort legacy operating system. Ideally, the kernel memory allocation mechanism will be different depending on whether the kernel is servicing a request from a critical application, legacy operating system, or an application hosted by that legacy operating system.

2.5 Assurance

Ideally, for a truly trustworthy embedded kernel, a high degree of assurance is required of any model and implementation of the kernel. Assurance here meaning proof of having the desired properties given a model of the kernel, and a proof that the implementation behaves as the model specifies. Without a high degree of assurance of this basic low-level system functionality, it is impossible to provide a high degree of assurance for the higher-level software stack built upon the kernel. Theorem proving tools have grown powerful enough, and microkernels are small enough, for such a degree of assurance to be feasible [TKH05].

Our desire (and efforts [DEK+06]) to formally verify our embedded kernel introduces another requirement to our design. In order to avoid invalidating any successful verification effort, any model addressing kernel memory allocation must ideally be fixed.

However, we have argued that in the embedded domain the operating system needs to support a wide range of memory allocation policies. By "operating system" we mean the microkernel and user-level servers that run outside the kernel and provide services to application programs. For this to be feasible, the microkernel itself must support diverse allocation policies over its in-kernel physical memory.

These potentially conflicting requirements lead to the conclusion that any kernel model that expects to *remain* verified must minimise the allocation policy in the kernel, and maximise the control higher-level software has over the management of physical memory in the kernel. If such a model exists then we can enforce diverse allocation policies over kernel memory by modifying higher-level software, rather than the kernel.

3 The seL4 Design

To meet the requirements discussed in Section 2, the seL4 project proposes a design inspired by early hardware-based capability machines (such as CAP [NW77]), where capabilities control access to physical memory; the KeyKOS and EROS systems [Har85, SSF99], with their controls on dissemination of capabilities; and the L4 microkernel [Lie95], where the semantics of virtual memory objects are implemented outside of the kernel.

In this section, we provide a brief overview of the seL4 microkernel, and a description of the mechanisms it provides for in-kernel memory management (see Section 3.1); then, in Section 3.2 we will discuss the benefits of our scheme.

3.1 Overview

Similar to its predecessor, the L4 microkernel [Lie95], seL4 provides three basic abstractions: threads, address spaces and inter-process communication. In addition, seL4 introduces a novel abstraction called *untyped memory* — an abstraction of a region of physical memory which we will later describe precisely.

These abstractions are provided via named, first-class kernel objects. Each kernel object implements a particular abstraction and supports one or more operations related to the abstraction it provides. Authorised users can obtain kernel services by invoking operations on kernel objects.

Authority over objects are conferred via capabilities [DVH66]. Capabilities are tamper-proof: they are stored inside kernel objects called *CNodes* — arrays of capabilities, which may be inspected and modified only via invocation of the CNode object itself — and therefore are guarded against user tampering. CNodes are similar to KeyKOS *nodes*, except that they vary in size in powers of 2, and are composed similar to *guarded page tables* [Lie94], to form a local capability address space called the *CSpace*. Capabilities are immutable; while user-level programs may specify some of the capability's properties at the time it is created, those properties may only be changed by removing the capability and replacing it with another.

System calls are invocations of capabilities. Users specify a capability as an index into a local capability address space, that would translate the given index to a capability. Tasks have no intrinsic authority beyond what they possess as capabilities.

3.1.1 Memory Allocation Model

At boot time, seL4 preallocates all the memory required for the kernel to run, including code, data, stack sections (seL4 is a single kernel-stack operating system). The remainder of the memory is given to the first task in the form of capabilities to *untyped memory* (UM), and some additional capabilities to kernel objects that were required to bootstrap the task. UM is restricted in size to powers of 2, and is used as the basis for creating all other objects in the system.

A capability to UM (a parent) can be refined into capabilities to smaller power-of-2 sized UM (children) via the *retype* method of UM. Retype has the following two restrictions:

1. the refined child capabilities must refer to non-overlapping UM objects of size less than or equal to the original, and
2. the parent capability must have no previously refined child capability derived from it.

The first restriction is obviously required; the need for the second restriction will be explained later.

In addition to smaller subdivided UM, the retype operation can also retype UM into a kernel object of a specific type. The seL4 API defines seven types of kernel objects, associated with the abstractions it provides. All kernel primitives (system calls) are invocations of these objects.

TCB (Thread Control Block) objects implement threads, which are seL4's basic unit of execution.

Endpoint objects implement inter-process communication *(IPC)*. Users send and receive messages by invoking capabilities to these objects. Like L4 IPC, this operation is synchronous and unbuffered.

Asynchronous Endpoint objects are used to implement asynchronous IPC. Rather than having a queue of threads waiting to send, they contain a buffer which is used to store the content of the message after a sender has resumed execution.

CNode objects are arrays of 2^n (where $n > 0$) capabilities. They constitute the CSpace — constructed as a tree of CNodes. Invoking a CNode allows a user-level server to manipulate a region of CSpace mapped by the tree of which that CNode is the root.

VNode objects are used to implement the data address space, or the VSpace. The exact structure of these objects would depend on the architecture. However, operations on these objects are essentially a subset of the CNode operations, subject to the restrictions enforced by the MMU. As such, we ignore these objects for now.

Frame objects provide storage to back virtual memory pages accessible to the user. Their size may be any power of two which is at least as large as the smallest possible virtual memory mapping on the host architecture.

Interrupt objects are used to store the bookkeeping required to associate interrupt delivery with an asynchronous endpoint.

The user-level manager that creates an object via retype will get the full set of authority over the object. It can then delegate all or part of the authority it possesses over the object to one or more of its clients. This is done by granting each client a capability to the kernel object, thereby allowing the client to obtain kernel services by invoking the object.

All the physical memory required to implement and bookkeep the object is pre-allocated within the object at the time of its creation, and does not exceed the size of the UM it was refined from. This means that there are no implicit allocations within the kernel — the kernel does not allocate any memory at the time of any object invocation.

Now returning to the second restriction above, it should be clear that to guarantee the integrity of kernel objects, a region of memory must implement a single type at a time. To ensure this, the retype operation needs to ensure that no parent of the previously refined capability undergoing the retype was refined into a type, nor any child of any parent. The second restriction above reduces this check to ensuring there is no child of the current capability. This ensures the operation is O(1), short lived, and requires no preemption point — i.e., it improves real-time properties of the kernel and reduces complexity of formal verification.

3.1.2 Re-using Memory

The model described thus far is sufficient for an initial task to subdivide UM and any refined kernel objects amongst its clients if the typed memory associated with kernel objects is never re-used. Similarly, clients can form subsystems with their own clients with their own policy enforced on physical memory consumption based on applying their policy on delegating the initial authority received.

In order to re-use memory, the kernel needs to guarantee that there are no outstanding valid capabilities to the objects implemented by that memory.

seL4 facilitates this by tracking capability derivations, which it records in a tree structure called the *Capability Derivation Tree* or *CDT*. As an illustrative example, when a user creates new kernel objects using an untyped capability, the newly created capabilities would be inserted into the CDT as children of the untyped capability. Similarly, any copy made from a capability would become a CDT child of the original.

To save memory, and avoid dynamic allocation of storage for CDT nodes, the CDT is implemented as a doubly-linked list stored within the (now larger) capabilities themselves. The list is equivalent to the post-order traversal of the logical tree. In order to reconstruct the tree from the list, each entry is tagged with its depth in the logical tree. The CDT adds two words to each capability, resulting in a capability size of four words; we view this as a reasonable trade-off.

Possession of the original untyped capability, that was used to allocate kernel objects, is sufficient authority to delete those objects. By calling a *revoke* operation on the original untyped capability, users can remove all its children — all the capabilities that are pointing to objects in the memory region covered by the UM object. This operation is a potentially long running operation, and thus is preemptable. The operation is still atomic as defined by Ford *et al* [FHL$^+$99], via restarting the system call after preemption while ensuring at least one child is revoked per restart.

Revoking the last capability to a kernel object is easily detectable, and triggers the *destroy* operation on the now unreferenced object. Destroy simply deactivates the object if it was active, and breaks any in-kernel dependencies between it and other objects. The ease of detection of revocation with the CDT avoids reference counting, which is an issue for objects without space to store the count (e.g. page tables) in a system without meta-data.

Once the revoke operation on the untyped capability is complete, the memory region can be re-used to allocate other kernel objects. Before re-assigning memory, the kernel affirms there are no outstanding capabilities. The CDT provides a simple mechanism to establish this — the untyped capability should not have any children.

For obvious security reasons kernel data must be protected from user access. The seL4 kernel prevents such access by using two mechanisms. Firstly, the above allocation policy guarantees that there are no overlapping *typed* objects. By typed objects, we mean any object other than a UM object. Secondly, before inserting a frame object mapping into the hardware MMU, the kernel checks the size of the object against the MMU page size.

3.2 Explicit User-Level Management

Figure 1 illustrates a sample system architecture, with a domain specific OS running at user-level receiving authority to remaining untyped memory after bootstrapping. The domain OS has the freedom to apply many policies depending on the domain, such as subdividing UM for delegation to the guest OS, or withholding UM and providing an interface to applications for them to request specifically typed OS services.

Virtual memory is provided by domain OS, by installing frame objects into vnode objects. Depending on how capabilities are distributed, domain OS could be only virtual memory provider, or the guest OS may have access to vnodes of its applications, or with appropriate

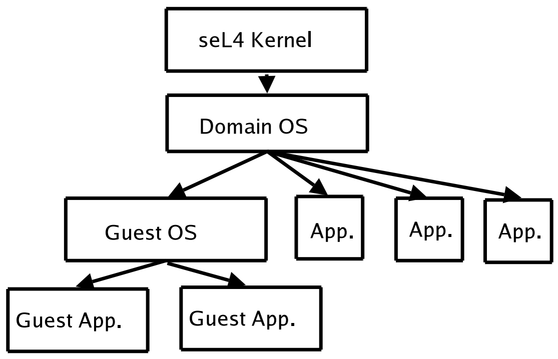

Figure 1: Sample system architecture.

authority, applications could even self-page.

While there are fews restrictions on the architecture of the overall system, what is guaranteed is that no application can exceed the authority it possesses, which also guarantees the precise amount of physical memory indirectly or directly consumed by and available to the application.

Application are at liberty to use a suitable policy to manage the available untyped memory. This can be a simple static or a complex dynamic policy. In the above example for instance, the guest OS might employ a complex and therefore error prone policy to manage its UM objects, in contrast to a simple static policy used by the domain OS. However, since the guest OS can not exceed the authority it possesses, any misbehaviour of the guest OS is isolated from the rest of the system. As a result, the guest applications benefit from the complex memory management policy employed by the guest OS, while the rest of the system is protected from any bugs incurred due to increase in code complexity in doing so.

4 Related Work

The CAP computer system [NW77] is similar to our approach in that capabilities to physical memory are required to create system objects. The most notable differences between CAP and seL4 are that seL4 avoids external memory fragmentation and simplifies bookkeeping by restricting object sizes to powers of 2, is a software-based implementation on modern hardware, and has capability management modelled after that of KeyKOS [Har85].

Eros [SSF99] and the *Cache kernel* [CD94] also manage their kernel data carefully. Both view kernel physical memory as a cache of the kernel data. However, as discussed previously, such an approach is not suitable to systems with temporal requirements.

The *K42* kernel [IBM02] takes advantage of C++ inheritance to control the behaviour of the underlying memory allocator. However, K42's focus is best-effort performance — it does not provide precise physical memory allocation guarantees, and variation of the memory management policies, while easily achieved, would invalidate any implementation proofs, if they where possible given K42's size and complexity.

Exokernel [EKO95] is a *policy free* kernel — its sole responsibility is to securely multiplex the available hardware resources. *Library Operating Systems*, working above the exokernel implement the traditional operating system abstractions. We could find little concrete details of the underlying meta-data management required to bookkeep the current state of the multiplexed hardware resources (e.g. the secure bindings), other than the caching approach to avoid meta-data exhaustion, which we have argued is insufficient.

Haeberlen and Elphinstone [HE03] implemented a scheme of paging kernel memory from user space. When the kernel runs out of memory for a thread, it will be reflected to the corresponding *kpager*. The kpager can then map any page it possesses to the kernel, and later preempt the mapping. However, the kpager is not aware of, and cannot control, the type of data that will be placed in each page and thus can not make an informed decision about which page to revoke. In contrast, user-level resource managers in seL4 are aware of the type of data placed in a page and therefore able to make informed decisions about resource revocation.

The *L4.sec* project at the Dresden University of Technology has similar goals to our own. They divide kernel objects into first-class (addressable via capabilities) and second-class objects (implicitly allocated). Both classes require *kernel memory objects* to provides the memory pool for creation of the objects [Kau05]. Kernel memory objects represent regions of memory used by the kernel for dynamic allocation. System calls requiring memory within the kernel, provide a capability to a kernel memory object. The model however, does not allow direct manipulation of second-class objects such as page tables or capability tables (CNodes). As such, managers are denied much of the flexibility provided by seL4's capability table interface. They also claim their design required locking within the kernel, where as our design is lock-free.

5 Conclusion and Future Work

In this paper, we have presented the a kernel memory management model that is mostly free of policy — it does not require the kernel to make any decisions about how, where or when to allocate kernel memory. Instead, it provides a secure interface for creating, managing, recycling, and destroying kernel objects from user level, using untyped physical memory objects. Kernel operations are (in the context of memory management) either constant time, or preemptable utilising restartable atomic operations, which naturally lends itself to a lock-free kernel implementation. We believe that such strong allocation guarantees, preemptability, and high flexibility are essential in the context of embedded systems, where application requirements are diverse and resources are scarce.

At present, the kernel API is implemented as an executable specification written in *Haskell*. We also have a proof of authority confinement within a model of the system, and thus a proof of physical memory "confinement" for systems meeting certain restrictions on object sharing[1]. We are now progressing towards a native implementation of the API to quantify the performance of our approach.

References

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *19th SOSP*, pages 164–177, Bolton Landing, NY, USA, Oct 2003.

[CD94] David R. Cheriton and K. Duda. A caching model of operating system functionality. In *1st OSDI*, pages 14–17, Monterey, CA, USA, Nov 1994.

[DEK+06] Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *ACM SIGPLAN Haskell WS*, Portland, Oregon, USA, Sep 2006.

[DVH66] J.B. Dennis and E.C. Van Horn. Programming semantics for multiprogrammed computers. *CACM*, 9:143–55, 1966.

[EGK95] Dawson R. Engler, Sandeep K. Gupta, and M. Frans Kaashoek. AVM: Application-level virtual memory. In *5th HotOS*, pages 72–77, May 1995.

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *15th SOSP*, pages 251–266, Copper Mountain, CO, USA, Dec 1995.

[FHL+99] Brian Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and execution models in the Fluke kernel. In *3rd OSDI*, pages 101–115, New Orleans, LA, USA, Feb 1999. USENIX.

[Fot61] J. Fotheringham. Dynamic storage allocation in the Atlas computer, including an automatic use of a backign store. *CACM*, 4:435–436, Oct 1961.

[Han99] Steven M. Hand. Self-paging in the Nemesis operating system. In *3rd OSDI*, pages 73–86, New Orleans, LA, USA, Feb 1999. USENIX.

[Har85] Norman Hardy. KeyKOS architecture. *Operat. Syst. Rev.*, 19(4):8–25, Oct 1985.

[HE03] Andreas Haeberlen and Kevin Elphinstone. User-level management of kernel memory. In *8th Asia-Pacific Comp. Syst. Arch. Conf*, volume 2823 of *LNCS*, Aizu-Wakamatsu City, Japan, Sep 2003. Springer Verlag.

[IBM02] IBM K42 Team. *Utilizing Linux Kernel Components in K42*, Aug 2002. Available from http://www.research.ibm.com/K42/.

[1]A technical report containing the proof will be available prior to the workshop.

[Kau05] Bernhard Kauer. L4.sec implementation — kernel memory management. Dipl. thesis, Dresden University of Technology, May 2005.

[LHH97] Jochen Liedtke, Hermann Hartig, and Michael Hohmuth. OS-controlled cache predictability for real-time systems. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, pages 213–227, Washington - Brussels - Tokyo, Jun 1997. IEEE.

[Lie94] Jochen Liedtke. Page table structures for fine-grain virtual memory. *IEEE Technical Committee on Computer Architecture Newsletter*, 1994.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *15th SOSP*, pages 237–250, Copper Mountain, CO, USA, Dec 1995.

[NW77] R.M. Needham and R.D.H. Walker. The Cambridge CAP computer and its protection system. In *6th SOSP*, pages 1–10. ACM, Nov 1977.

[Rus99] John Rushby. Partitioning for safety and security: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, Jun 1999. Also to be issued by the FAA.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *17th SOSP*, pages 170–185, Charleston, SC, USA, Dec 1999.

[TKH05] Harvey Tuch, Gerwin Klein, and Gernot Heiser. OS verification — now! In *10th HotOS*, pages 7–12, Santa Fe, NM, USA, Jun 2005. USENIX.

[Wal02] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *5th OSDI*, Boston, MA, USA, 2002.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *5th OSDI*, Boston, MA, USA, Dec 2002.