Kernel Design for Isolation and Assurance of Physical Memory

Dhammika Elkaduwe Philip Derrin Kevin Elphinstone
NICTA[*] and University of New South Wales
Sydney, Australia

ABSTRACT

Embedded systems are evolving into increasingly complex software systems. One approach to managing this software complexity is to divide the system into smaller, tractable components and provide strong isolation guarantees between them. This paper focuses on one aspect of the system's behaviour that is critical to any such guarantee: management of physical memory resources.

We present the design of a kernel that has formally demonstrated the ability to make strong isolation guarantees of physical memory. We also present the macro-level performance characteristics of a kernel implementing the proposed design.

Keywords

isolation, seL4, memory management, embedded systems, microkernels

1. INTRODUCTION

Embedded systems are evolving into increasingly complex software systems. Drivers of this trend include the desire to consolidate functionality previously spread across several platforms onto a single platform, and supporting ever-increasing feature sets. We have moved from single-vendor closed system, to systems incorporating software, sourced from third parties, that possesses varying degrees of assurance. High-end embedded systems now host leagacy operating systems, such as Linux, to further extended the feature set, together with user-installed, untrusted applications. While functional correctness is becoming increasingly difficult and expensive to achieve, it is becoming more important, as safety-critical or mission-critical systems are targetted for consolidation.

One approach to improving the robustness of embedded systems is to divide the system into components and provide strong isolation guarantees between them — the failure of a component is isolated from the rest of the system. There are many approaches to providing isolation guarantees, such as the classical processes and virtual memory [8], separation kernels [24], isolation kernels [30], virtual machines [29] and microkernels [16], all of which provide varying levels of isolation guarantees at different granularities. The level of isolation that can be achieved depends, to a large degree, on the mechanisms provided by the underlying kernel — be it a microkernel, a virtual machine monitor, or an isolation kernel.

This paper focuses on one aspect of providing strong isolation guarantees — 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.

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 metadata to implement the service (such as page tables) or provide the book-keeping required to reclaim the storage on release. Ideally, any kernel mechanism provided to manage physical memory and enforce an particular isolation policy must encompass both physical memory consumed directly by the application, and any physical memory consumed indirectly by the kernel metadata needed to provide services to applications.

We can summarise our approach to this problem as: (a) eliminating the need for all implicit allocation of metadata from kernel, and (b) the promotion of all kernel metadata (including the provision for book keeping) into first-class, explictly-allocated objects. This approach reduces the problem of enforcing a physical memory isolation policy over the components, to that of enforcing isolation over the authority to explicit kernel object creation and object-authority distribution between components, which is achieved by a capability based, decidable access control model.

Kernel services and their implementation complexity have a direct consequence on the ability to successfully apply 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 strong guarantees on the sufficiency and the implementation correctness of the mechanisms required for achieving true isolation. Moreover, in order to realise the proposed model, the system must be equipped with a hardware MMU. Thus the target of our work is mid- to high-end devices of the embedded systems spectrum.

In the remainder of the paper, we discuss the requirements and issues surrounding the management of physical memory in the context of enforcing isolation boundaries, that consequently motivated our approach (Section 2). In Section 3, we introduce the memory management model and a summary of its performance is given in

[*]NICTA 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.

IIES'08 April 1, 2008, Glasgow, UK
Copyright © 2008 ACM 978-1-60558-126-2 ...$5.00.

Section 4. Finally, Section 5 discusses related work, and conclusions and future work are in Section 6.

2. REQUIREMENTS

Memory allocation to support kernel services and associated metadata can have a direct or indirect effect on the spatial and temporal isolation between components, and the efficiency and assurance of the overall system. In the following sections we examine these issues and requirements of memory allocation mechanisms in each of these areas.

2.1 Spatial Isolation

Physical memory is a limited and exhaustible resource. Any limited resource requires precise managment to avoid one task's requests for authorised services from directly or indirectly denying services 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 under-utilisation, due to the overly conservative commitment of resources required to ensure all authorised service requests are satisfied during peak demand. Significantly higher efficiency can be achieved if memory can be safely reassigned to where it can be utilised [29]. Memory allocation mechanisms must support dynamic allocation and safe re-assignment of memory in a controlled manner.

2.2 Temporal Isolation

Temporal isolation is an important issue in the context of real-time embedded systems. By temporal isolation we mean the predictability of execution times of a task, irrespective of other system activities. The main issue that arises with memory allocation in this context is the predictability of execution of kernel operations. Memory allocation affects predictability when the kernel's physical memory is treated as a cache to avoid kernel memory starvation. Several operating systems use the kernel's physical memory as a cache of data structures stored at user-level or on disk [3, 25], and thus can always evict cache content to service new requests. While, such a strategy avoids physical-memory based denial-of-service attacks, it can not guarantee the temporal isolation — authorised request from one task might evict an entry from another and thereby cause unexpected system call delays for the latter task. The kernel's physical memory management scheme must be capable of providing predictable access times to data structures associated with those tasks that require temporal predictability.

CPU cache colouring techniques for improving predictable cache behaviour are also dependent on control of the memory allocation of the data structures requiring colouring [17], including the kernel's internal data structures.

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.

2.3 No Single Policy

Modern embedded devices are multipurpose appliances; they are composed of applications with diverse resource requirements. At one extreme, the system must support applications with stringent temporal requirements. For kernel memory allocation, this implies a guaranteed allocation of physical memory (including metadata) to those applications, that cannot be interfered with by any activities of the rest of the system. In the other extreme, system is expected to support best-effort applications where physical memory management is dynamic, on-demand and uncritical. These application specific requirements must be considered in making memory management decisions, so that we can maximise the use of the limited memory resource.

A realistic example of the latter extreme is in efficiently supporting a para-virtualised legacy operating system [1] on the microkernel. Ideally, we want to isolate the critical components of the system from the legacy OS (and its applications), while facilitating the legacy OS to make its own memory management decisions — the legacy OS is in the best position to determine the allocation of page tables, pages, frames, and thread control blocks for its clients. 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 [6, 10].

2.4 Assurance

The ability to reason about the behaviour of components is central to any system attempting to make strong isolation guarantees; one must be capable of deciding whether or not isolation truly holds for a given system configuration and in particular assure that it cannot be violated in any future state. We believe that such assurance can only be provided within a formal framework — a formal model that captures the operational semantics of the kernel together with a mathematical proof of its ability to enforce isolation. In our context, this means (a) the model employed to control the allocation of physical memory must be amenable to formal modelling, and (b) the formal model must be *decidable*.

Moreover, to make such an analysis truly trustworthy, we need to verify the kernel implementation against the formal model used for the analysis. Verification here means a formal refinement proof from the model down to kernel code. Our desire (and efforts [5]) to formally verify the embedded kernel introduces another requirement to our design. A verified kernel, implies a relatively static kernel, in the sense of its design. This is motivated by the desire to avoid invalidating any successful verification proofs of the kernel. Ideally, the kernel code, including the memory management model should be fixed. Changing the lowest-level of behaviour; such as that of the heap, requires a significant redo of the proofs. However, given the diversity of embedded applications we expect no single memory allocation policy to suffice. This leads to the conclusion that a verified embedded kernel must minimise the allocation policy in the kernel, and maximise the control higher-level software over manging physical memory within the kernel.

3. SEL4 MEMORY ALLOCATION MODEL

The seL4 project proposes a novel kernel memory management model to meet the requirements identified in Section 2. The design of our model is inspired by early hardware-based capability machines (such as CAP [22]), where capabilities control access to physical memory; the KeyKOS and EROS systems [11, 25], with their controls on dissemination of capabilities; and the L4 microkernel [16], where the semantics of virtual memory objects are implemented outside of the kernel.

We start with an overview of the model together with a rationale of our design choices (see Section 3.1), which is followed by a discussion of issues and solutions related to realising the model (see Section 3.2).

3.1 Overview

In our model, all metadata are promoted into first-class kernel objects, that are allocated upon explicit user requests. We explain how users request allocation of these objects later. Each kernel object implements a particular abstraction, and supports one or more operations related to that abstraction. For instance, abstraction of threads is provided by the *TCB* object, and thread-related system calls are the operations supported by the TCB object.

Authority over objects are conferred via tamper-proof (partitioned) capabilities [4]. Authorised users obtain kernel services by invoking capabilities.

Other than these explicitly-allocated kernel objects, the kernel allocates no metadata — there are no implicit allocations within the kernel. We achieved this by carefully avoiding, where possible, mechanisms that require in-kernel bookkeeping; and by including the essential bookkeeping data in the explicitly allocated kernel objects themselves.

There are two aspects that are worth mentioning here. Firstly, explicit allocation provides the required flexibility — it is the users that instruct the kernel to allocate a particular object type. Secondly, the approach reduces the problem of memory allocation to kernel-object allocation. Thus, for achieving isolation, we simply require precise control of kernel-object allocation.

Capability-based systems and their formal properties are well studied in literature [2, 20, 26, 27]. When coupled with a suitable authority transfer scheme; such as the *take-grant* [18], these systems yield a *decidable* access control model. Decidable here means that future access rights a task may obtain can be decided by analysing the initial system state. Moreover, with the take-grant model, enforcing some invariants on the initial distribution of capabilities guarantees isolation of authority.

While the primary concern of take-grant (and the like) is controlling access to user-level resources, its properties match to our requirements for control of kernel-object allocation. seL4 leverages this similarity by extending its *take-grant-like* capability protection model to control the allocation of kernel objects.

In seL4, allocation of kernel objects is performed by the *retype* method, implemented by a special object type called *untyped memory* (UM) — an abstraction of a power-of-two sized, size-aligned region of physical memory. Possession of an UM capability provides sufficient authority to allocate kernel objects in the corresponding region — by invoking a UM capability, a user-level application can request that the kernel refines that region into other kernel objects (including smaller UM objects).

As such, the set of UM capabilities possessed by an application defines the precise amount of physical memory that can be directly or indirectly consumed by the application. Moreover, the explicit allocation scheme provides user-level applications the freedom to make policy decisions on how to use the UM capabilities in its possession. We introduce the mechanisms used to restrict the number of UM capabilities an application possesses later in Section 3.2.3.

3.2 The seL4 Kernel

In this section, we provide a detailed description of our memory management approach as applied to the seL4 kernel.

3.2.1 Memory Allocation

At boot time, seL4 preallocates all the memory required for the kernel itself, including the code, data, and 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.

A capability to UM can be used to create new kernel objects that are smaller power-of-two sized, size-aligned objects via the retype method, which returns a capability to the new object. These capabilities to the new objects are called *children*, and the original UM capability the *parent*. Retype enforces the following invariants:

1. the child objects must be wholely contained within the original parent UM, and
2. the child objects must be distinct and non-overlapping.

The child objects may be UM (which may themselves eventually become parents of their children), or they may be kernel objects of other types that implement kernel services. The details of the other types depend on the services provided by the kernel, and can be ignored for the purposes of our discussion.

The user-level application (manager) that creates an object via retype receives full 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 the object is preallocated within the object at the time of its creation, and does not exceed the size of the UM it was refined from. This eliminates the need for the kernel to dynamically allocate memory to satisfy requests from user-level tasks.

For obvious security reasons, kernel data must be protected from user access. The seL4 kernel prevents such access by using two mechanisms. First, the above allocation policy guarantees that typed objects never overlap. Second, the kernel ensures that each physical frame mapped by the MMU at a user-accessible address corresponds to a typed *frame object*; frame objects contain no kernel data, so direct user access to kernel data is not possible.

3.2.2 Re-using Memory

The model described thus far is sufficient for allocating kernel objects, distributing authority among clients, and obtaining various kernel services provided by these objects. This alone is sufficient for a simple static system configuration.

In order to re-use memory in a dynamic system, 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, via linking derived capabilities through themselves to form 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 avoid dynamic allocation of storage for CDT nodes, the CDT is implemented as a doubly-linked list stored within the capabilities themselves. The list is equivalent to the post-order traversal of the logical tree. In order to reconstruct the tree from the list, the kernel uses a combination of tag bits and physical address comparisons to determine whether an adjacent pair of capabilities have a parent-child relationship. The CDT adds two words to each capability, resulting in a capability size of four words; we view this as a reasonably low cost for enabling memory reuse.

Possession of the original UM capability that was used to allocate kernel objects provides 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 preemptible.

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 child objects in 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 metadata.

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.

3.2.3 Isolation

We have described that our approach creates a direct relationship between the posession of authority (via capabilities) and direct and indirect access to physical memory. We now show how we exploit this relationship to guarantee isolation of physical memory via the isolation of authority.

Enforcing isolation at the granularity of an "application" is overly restrictive; in some case, we want to allow a set of applications to share physical memory, but isolate that set from the rest of the system. A realistic example of this is when hosting a paravirtualised operating system — we want to allow the sharing of physical memory between the guest OS kernel and the set of client applications that are directly supported by it, while isolating that set from the rest of the system.

To facilitate discussion we define the term *isolation domain* — a collection of applications whose direct and indirect physical memory access is isolated from the rest of the system. However, all applications within an isolation domain can, if need be, share resources. In more precise terms, within an isolation domain, capabilities can propagate freely, but can not cross an isolation domain boundary. Isolation domains must be *mandatory*; that means, a capability can not ever cross an isolation domain not because of the unwillingness of the applications, but because it is not allowed by the system configuration.

There are three important questions we need to answer:

- Are the seL4 mechanisms sufficient to create and enforce isolation domains?
- What invariants are required on the distribution of authority to achieve isolation?
- Are these invariants reasonable in practice?

To answer the first two questions, we formalised the seL4's protection model in the automated theorem prover *Isabelle/HOL* [23]. Introducing this formalism and the formal treatment of the above two questions is beyond the scope of this paper. Informally, seL4's protection model is a variant of the classical take-grant protection model — the variation arises mainly from the introduction of UM capabilities to control the *create* operation of take-grant. In addition, seL4 discriminates between information flow channels and authority flow channels. This means we have two types of interprocess communication (IPC) channels; channels that only allow information to flow, and channels that allow both information and authority flows. For clarity we call the latter *grant IPC* channels.

Based on our formalism, we formally proved that seL4's mechanisms are sufficient to enforce isolation domains. Moreover, the proof also identifies the invariants that should be enforced over the initial system configuration so that isolation domains hold for any future state.

From a practical point of view, the proof states that, if we enforce the following restrictions on the initial system configuration, then isolation domains are guaranteed to hold in future state derived from it, by executing any sequence of system commands. The restrictions are as follows:

- An application can only be in one isolation domain.
- Writable page tables and writable CNodes (capability storage) cannot be shared between applications in different isolation domains.
- There should not be any grant IPC channel between any two applications in different isolation domains.
- Any application in one isolation domain should not have the authority to modify the authority of a thread (i.e. a capability with a specific right to the TCB object implementing the thread) in another isolation domain. Or simply, we should not share capabilities to TCBs across isolation domains.

To demonstrate that these invariants are not overly restrictive in practice, we have constructed an example system that runs a legacy OS kernel in an isolation domain (see Section 4). We believe these invariants are not overly restrictive in constructing almost all practical systems.

4. PERFORMANCE EVALUATION

We have implemented our kernel memory-management model in an experimental kernel called *seL4::Pistachio*, based on *L4-embedded* [21], which runs on ARM11-based CPUs.

To examine the performance characteristics of the proposed memory management scheme, we ported *Wombat* [15] — a paravirtualised Linux 2.6 kernel that runs on the *L4/Iguana* operating system, to seL4::Pistachio system. Hereafter, we use the term *seL4::Wombat* to refer to the Linux version running on seL4::Pistachio and *L4::Wombat* refers to the Linux kernel on top of *L4/Iguana*. We evaluate the performance characteristics of the model by running the lmbench [19] suite on both the systems. Benefits of virtualization in the context of embedded systems can be found elsewhere [12], here we are using seL4::Wombat as an illustrative example of a complex software system.

All results were obtained on a *KZM-ARM11* evaluation board, which comprises an ARM1136-based (Freescale i.MX31) processor running at 532MHz with an L2 cache of 128KB and with 128MB of RAM.

Before investigating the results, we describe the architecture, configuration, and the level of isolation of each system. Both L4::Wombat and seL4::Wombat are microkernel-based systems. The underlying microkernel in the former case is based on L4-embedded, which is based on L4 [16], and in the latter case it is seL4::Pistachio. Iguana is a small, capability-based operating system that runs in user mode on top of L4 that provides and enforces the underlying resource management policy in conjunction with L4. The authority to obtain kernel services that require the allocation of kernel metadata is centralised in Iguana — any kernel service that may require the allocation of kernel metadata must be made through, and therefore monitored by Iguana. Through this mechanism, Iguana enforces a strict control over the physical memory consumption of L4::Wombat. In contrast, seL4::Wombat is placed under similar controls by controlling the dissemination of Untyped capabilities. Once seL4::Pistachio has bootstrapped itself, it calls a resource manager which then bootstraps seL4::Wombat. In the process of bootstrapping, the resource manager grants, among others, a set of UM capabilities to seL4::Wombat and thereby allowing it to allocate/deallocate and

Latency	L4 [μs]	seL4 [μs]	% Gain
pagefault	34.4	18.7	45.4
fork	4570	3083	32.5
exec	5022	3440	31.5
shell	29729	19999	32.7
syscall	3.4	2.9	11.1
pipe	53.5	49.0	8.4
tcp	7.3	6.5	10.6
unix	116.5	109.4	6.0
ctx (4k 2)	10.7	9.31	7.6
Bandwidth	[MB/s]	[MB/s]	%
pipe	61.1	61	-0.2
tcp	37.3	35.6	-4.5
unix	13.7	14.5	6.2
mem	364.8	367.1	0.6
mmap	696.4	696.5	0

Table 1: Lmbench performance of L4::Wombat vs. seL4::Wombat

manage kernel objects directly — seL4::Wombat is at liberty to use these UM capabilities as desired. For example, it can use them to create address spaces, threads or frames and so on. However, by enforcing the invariants we identified through the formal analysis, the resource manager guarantees that seL4::Wombat cannot access more UM capabilities.

Lmbench system latency and bandwidth results are shown in Table 1. The first set of results in *latency* shows the latency of operations that require the allocation of physical memory. The second set are the Linux system calls that can be handled without kernel metadata allocation, and the final set of numbers shows the bandwidth for various lmbench tests.

The benchmarks that require allocation of physical memory show the performance benefits of our memory management scheme. By decentralising resource management, seL4 eliminates the need for Linux to proxy these requests through Iguana to the microkernel in order to enforce an isolation policy over the Linux sub-system. At the same time, Linux itself is free to share or isolate the resources it provides to its clients. Removing the need for mediating microkernel system calls in order to enforce an isolation policy has significant performance advantages.

The second group of latency benchmarks, and the bandwidth benchmarks, do not require microkernel memory allocation (and thus mediation) in the performance-critical component, and thus do not show the presence and absence of mediate overheads in the two systems repectively. The performance for these benchmarks is mostly sensitive to the cost of *exception IPC*. When an application traps into the microkernel with a system call number which the kernel does not handle, the kernel generates and sends an exception IPC to the Linux server for it to emulate a Linux system call for the trapping application. For these results, seL4 exhibits much more modest gains. However, these results are biased towards seL4 as we have a hand-optimised assembly exception IPC path, versus L4 which relies on “C” for exception IPC on ARM11.

We are confident that a hand-optimised, assembly exception IPC routine would deliver better the results for L4. But, our experience in optimising the two microkernels suggests that such an optimisation will only be a slight improvement over the *seL4* numbers — the cost of delivering an IPC (as opposed to exception IPC), through a hand-optimised, assembly routine in L4 is 199 machine cycles, and seL4 is not far behind by taking 205 cycles. Validating this claim, however is on going research.

5. RELATED WORK

The CAP computer system [22] 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 and it is a software-based implementation on modern hardware, and has capability management modelled after that of KeyKOS [11].

Eros [25], the *Cache kernel* [3] and the *HiStar* system [31] managed their kernel data carefully. All three systems 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 [13] 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 [7] 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 metadata management required to bookkeep the current state of the multiplexed hardware resources (e.g. the secure bindings), other than the caching approach to avoid metadata exhaustion, which we have argued is insufficient.

Haeberlen and Elphinstone [9] implemented a scheme fro 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 *Fluke* kernel provides mechanisms to export the kernel state to user-level applications, which has been used to implement check-pointing [28]. However, the kernel memory itself cannot be controlled. In contrary, the main objective of seL4 is to provide precise control over the kernel memory consumption, rather than exporting the kernel state to user-level applications.

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 provide the memory pool for creation of the objects [14]. 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, dynamically reallocating page table memory from an idle task to other tasks is not possible without destroying the former task. Additionally, L4.sec is denied much of the flexibility provided by seL4’s capability table interface — it is not possible to share capability storage between tasks.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a kernel design that is capable of providing strong guarantees on the physical memory consumption of an application. We achieve this by extending the access control mechanism of the kernel to include all physical memory that an application can directly or indirectly consume — including kernel's metadata. The kernel itself is mostly free of memory management policy — it does not require the kernel to make any decisions about how, where or when to allocate kernel memory. Instead, user-level applications create, manage, recycle and destroy kernel objects via the secure access control scheme. Thereby provide strong allocation guarantees and the flexibility required to manage the scarce memory resource of an embedded device.

We have evaluated the design in two aspects; its formal properties and its performance. To evaluate the formal properties, we developed a machine-checked, abstract specification of the kernel model in *Isabelle/HOL* and formally proved that the design is sufficient to achieve physical memory isolation. We have characterised the performance of the design by using it as a platform for hosting a para-virtualised Linux kernel, which shows significant performance gains for benchmarks that require allocation of physical memory. At present, we are working on improving the performance numbers and making a more concrete comparison with existing systems.

7. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *19th SOSP*, pages 164–177, Bolton Landing, NY, USA, Oct 2003.

[2] A. C. Bromberger, A. P. Frantz, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *USENIX WS Microkernels & other Kernel Arch.*, pages 95–112, Seattle, WA, USA, Apr 1992.

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

[4] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *CACM*, 9:143–155, 1966.

[5] P. Derrin, K. Elphinstone, G. Klein, D. Cock, and M. M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *ACM SIGPLAN Haskell WS*, Portland, OR, USA, Sep 2006.

[6] D. R. Engler, S. K. Gupta, and M. F. Kaashoek. AVM: Application-level virtual memory. In *5th HotOS*, pages 72–77, May 1995.

[7] D. R. Engler, M. F. Kaashoek, and J. 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.

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

[9] A. Haeberlen and K. 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.

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

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

[12] G. Heiser. The role of virtualization in embedded systems. In *1st IIES*, Glasgow, UK, Apr 2008. ACM SIGOPS.

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

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

[15] B. Leslie, C. van Schaik, and G. Heiser. Wombat: A portable user-mode Linux for embedded systems. In *6th Linux.Conf.Au*, Canberra, Apr 2005.

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

[17] J. Liedtke, H. Härtig, and M. 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, Montreal, Canada, Jun 1997. IEEE.

[18] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *J. ACM*, 24(3):455–464, 1977.

[19] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *1996 Ann. USENIX*, San Diego, CA, USA, Jan 1996.

[20] N. H. Minsky. Selective and locally controlled transport of privileges. *ACM Trans. Program. Lang. Syst.*, 6(4):573–602, 1984.

[21] National ICT Australia. *NICTA L4-embedded Kernel Reference Manual Version N1*, Oct 2005. http://ertos.nicta.com.au/Software/systems/kenge/pistachio/refman.pdf.

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

[23] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[24] J. M. Rushby. Design and verification of secure systems. In *8th SOSP*, pages 12–21, 1981.

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

[26] J. S. Shapiro and S. Weber. Verifying the EROS confinement mechanism. In *Symp. Security & Privacy*, 2000.

[27] L. Snyder. Formal models of capability-based protection systems. *IEEE Trans. Comput.*, 30(3):172–181, 1981.

[28] P. Tullmann, J. Lepreau, B. Ford, and M. Hibler. User-level checkpointing through exportable kernel state. In *IWOOOS '96: proc 5th International Workshop on Object Orientation in Operating Systems*, pages 85–88, Washington, DC, USA, 1996. IEEE Computer Society.

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

[30] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *5th OSDI*, Boston, MA, USA, Dec 2002.

[31] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *7th OSDI*, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association.