



MIKES 2007
First International Workshop on
Microkernels for Embedded Systems

Editors: Ihor Kuz and Stefan M. Petters
Program Committee Chair: Kevin Elphinstone

National ICT Australia
223 Anzac Parade
Kensington NSW 2052 Australia
{firstname.lastname}@nicta.com.au
Technical Report January 2007

ISSN 1833-9646

Copyright 2007 National ICT Australia. All rights reserved.
The copyright of this collection is with National ICT Australia.
The copyright of the individual articles remains with their authors.

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.

Table of Contents

Forword.....	4
Formalizing Information Flow in a Haskell Hypervisor..... <i>Rebekah Leslie, Levent Erkok, and Flemming Andersen</i>	5
High-Performance Microkernels and Virtualisation on ARM and Segmented..... Architectures <i>Carl van Schaik and Gernot Heiser</i>	11
Automated Object Layout Optimization in a Portable Microkernel..... <i>Uwe Dannowski</i>	22
A Memory Allocation Model For An Embedded Microkernel..... <i>Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone</i>	29
L4-Based Real Virtual Machines: An API Proposal..... <i>Sebastian Biemueller and Uwe Dannowski</i>	36
A Declarative Approach to Extensible Interface Compilation..... <i>Nicholas FitzRoy-Dale</i>	43
Evolution of the PikeOS Microkernel..... <i>Robert Kaiser and Stephan Wagner</i>	50
Issues on Analysing L4 for its WCET..... <i>Mohit Singal and Stefan M. Petters</i>	58

Foreword

The First International Workshop on MicroKernels for Embedded Systems was conceived to provide a forum for the discussion of issues in the application of microkernels in the embedded systems domain, including emerging problems, novel concepts, analysis approaches, and case studies.

The workshop attracted submissions from both academia and industry. The submitted papers were peer reviewed by an expert program committee. I wish to express my gratitude to the members program committee who contributed their time to provide high quality reviews, which eased the job of selecting a quality program.

The authors were given the opportunity to revise and re-submit final versions of their papers based on the reviews, and on the discussions that developed at the workshop. The papers contained herein are the final versions submitted after the workshop.

I'd like to thank our guest speaker David Kleidermacher of Green Hills Software who provided an interesting overview of a successful commercial microkernel for embedded systems. I'd also like to thank the discussion panel: Gernot Heiser, National ICT Australia and UNSW; Timothy Roscoe, ETH, Zurich; Andrew Tanenbaum, Vrije Universiteit; and David Kleidermacher. The panel discussion proved lively, insightful, and entertaining for all present.

I also wish to thanks Stefan M. Petters and Ihor Kuz, who did much of the organising of the event.

Kevin Elphinstone
Program Chair

Program Committee

Kevin Elphinstone, NICTA/UNSW, Australia (Chair)
Andrew Tanenbaum, Vrije Universiteit, Amsterdam, Netherlands
Gerwin Klein, NICTA/UNSW, Australia
Hermann Haertig, TU Dresden, Germany
Jonathan S. Shapiro, The Johns Hopkins University, USA
Neil Audsley, University of York, UK
Sebastian Schönberg, Intel, Hillsboro, USA
Volkmar Uhlig, IBM Watson, USA

Formalizing Information Flow in a Haskell Hypervisor

Rebekah Leslie
Portland State University

Levent Erkök and Flemming Andersen
Intel Corporation

Abstract—Separation kernels are the holy grail of secure systems, remaining elusive despite years of research into their design, implementation, and analysis. Though separation kernel research has achieved many successes, the disconnect between information flow theory and system implementation is a significant barrier to further progress. In this paper, we show how a particular branch of information flow theory, noninterference, can be utilized to formulate correctness and security properties of a microkernel-style hypervisor. Thus, we not only provide a first step towards a formally verified separation kernel, but also reduce the gap between information flow theory and operating systems practice.

I. INTRODUCTION

Noninterference provides a concise and general way to formalize the information flow relationships between components of a system. A noninterference security policy specifies which components, or domains, may not *interfere* with each other, where a domain u interferes with a domain v if v can observe the effects of u 's execution [1]. The generality of such policies makes them useful for capturing a wide variety of security requirements, including separation. A key benefit of noninterference is that there are existing frameworks for reasoning about systems governed by a noninterference policy [1], [2], thus reducing the barriers to verifying such systems formally.

Our interest in noninterference stems from our efforts to develop a hypervisor with formally verified separation between user-level processes. The design of our hypervisor is similar to secure microkernel APIs such as seL4 [3] and L4sec [4], but we use the term hypervisor to emphasize our intent to employ our system as a platform for secure, separate execution. Such an execution environment is an essential part of many high assurance systems, and is increasingly important in light of recent hardware developments, such as multi-core platforms based on Intel® Virtualization Technology and Intel® Trusted Execution Technology [5].

Following the approach of the Programatica [6] and seL4 [3] projects, we are writing a model of our hypervisor, called HHV, in the functional language Haskell [7]. The mathematical semantics and strong type system of Haskell make our model easier to reason about than a low-level implementation. By combining the use of a high-level functional language with the application of an existing reasoning framework, proving

separation for HHV becomes a more tenable goal than with other techniques.

In this paper, we concentrate on the formulation of correctness and security properties of the communication mechanisms in HHV, because these are the source of all legal information flow in our design. We formally characterize the information flow relationships induced by these communication primitives using a notation based on higher-order logic, similar to the P-logic programming logic for Haskell [8]. More importantly, we develop specification patterns for expressing correctness properties in terms of noninterference concepts. In defining these patterns, we take advantage of higher-order functions to extract common aspects of noninterference-style properties. For instance, we abstract over the kernel state and kernel operations so that we can instantiate the patterns in different contexts and formulate assertions about particular operations in a generic way. Hence, our work provides a link between the theory of noninterference and the actual practice of building secure microkernels.

We organize the remainder of the paper as follows. Section II describes the foundational concepts of HHV and outlines the communication mechanisms. Section III presents the system model used by our hypervisor. Section IV introduces the noninterference specification patterns that are the basis of our property formulations. Section V defines the desired information flow behavior of HHV, both as an informal specification and as a set of formal properties. We discuss related work in Section VI and present our conclusions in Section VII.

II. COMMUNICATION IN HHV

The fundamental abstractions in HHV are *protection domains*—the basic unit of resource protection—and *execution contexts*—the unit of execution. A protection domain (PD) corresponds to an address space in other systems; an execution context (EC) corresponds to a thread. HHV is a migrating thread system [9], so there is a single execution context per processor that moves between protection domains with the logical flow of control.

Each EC contains a representation of the processor state—such as the general purpose registers and the instruction pointer—for the running domain. We store this hardware context using a record type, called `Context`, which contains a field for each hardware register. When a domain is not running, HHV preserves the processor state in a region of memory called the save/restore area (SRA).

Levent Erkök is now at Galois Connections, Inc.

Protection domains communicate through uni-directional channels called *portals*. A *portal traversal* causes a context switch from the initiator of the traversal (the source domain) to the target of the traversal (the destination domain), potentially transferring a message from the source to the destination in the process. There are two modes of communication in HHV: direct data transfer via registers and indirect data transfer through shared resources, such as memory pages.¹ In this paper, we focus on the information flow properties of direct data transfer, although we have also formally characterized the aspects of portal traversal that deal with resource sharing.

HHV uses a dynamically configurable set of message registers. This approach results in an extremely flexible communication mechanism, and also allows us to overcome the security issues induced by the use of a migrating-thread model [9], [10]. Specifically, we must guarantee that a portal traversal does not leak information that the source domain wishes to keep private and that a portal traversal does not overwrite data that the destination wishes to preserve. By allowing both the source and destination domains to control which registers are part of a message, we enable the domains to protect their state against unwanted observation and modification. To this end, we introduce the concept of *portal masks*, which are fine-grained guards used by the source and destination domains to control information flow.

A traversal potentially affects every register in the execution context, so a mask (represented by the type `Mask`) contains a flag for every register in the hardware context. There are two masks involved in each portal traversal: the source domain uses a *transfer mask* to control which registers are sent during a traversal and the destination uses a *pass mask* to control which registers are modified. HHV only copies a register value from the source to the destination if both masks permit the transfer.

We define a context switch primitive that encapsulates the direct data transfer aspect of portal traversal. The exact behavior of a switch depends on the portal mask settings, but the fundamental algorithm consists of three basic steps.

- 1) Filter the running EC to remove any data that the source does not wish to send to the destination, as specified in the transfer mask, and save this data to the source SRA.
- 2) Perform a context switch to the destination.
- 3) Load the saved state from the destination’s SRA into the EC, as dictated by the pass mask settings.

We implement these steps using several state-manipulation functions, the type signatures for which are shown in Figure 1: `saveContext` writes the registers that are not part of the message to the source SRA, `computeTransfer` builds a new context containing only the message registers, `loadContext` restores the state of the destination from its SRA, and `computePass` combines the restored registers with those sent by the source domain. The top-level context switch operation, `switch`, weaves together these state-manipulation functions to achieve the data transfer and preservation behavior specified

by the portal masks.

```

saveContext      :: Mask → Context → PDID → Kernel ()
computeTransfer :: Mask → Context → Context
loadContext     :: Portal → SRA → Context
computePass     :: Mask → Context → Context → Context

```

Fig. 1. Functions for preserving domain state and performing a message transfer. These functions are used by the context switch operation to implement the dynamic message-transfer semantics specified by the portal masks.

The implementation of `switch` depends on the kernel state, information about the source and destination domains, and the portal configuration (which includes the mask settings). These dependencies are reflected in the type signature of `switch`:

```
switch :: PDID → PDID → Portal → Kernel ()
```

The parameters are the source domain (of type `PDID`), the destination domain (of type `PDID`), and the portal configuration (of type `Portal`). The operation runs in a monad called `Kernel`, which encapsulates the kernel state. We will examine the details of this monad in the next section; for now it is only important to note that `Kernel` contains a state component for the running EC.

The first step of `switch` is to perform the source-side state-manipulations; this involves calling `saveContext` to store the registers that are not being sent and calling `computeTransfer` to clear the value of those registers in the running EC. The variable `ec` corresponds to the running EC (read from the kernel state by the `get` function).

```

switch src dst port
= do ec ← get
    saveContext (transfer port) ec src
    let ec' = computeTransfer (transfer port) ec

```

At this point, `ec'` contains only the data that the source wants to send to the destination, so we perform a context switch to the destination by setting the active domain to be the destination. Next, we perform the destination-side state-manipulations. We invoke `loadContext` to restore the registers that the destination does not wish to receive from the source. The final register values are obtained by merging the sent context with the restored context using `computePass`.

```

setActiveDomain dst
dstSRA ← getSRA dst
let rc = loadContext port dstSRA
set (computePass (pass port) ec' rc)

```

We omit the details hidden by the sub-computations of `switch`, because they are not essential for formalizing the behavior of HHV. The important idea is that each of these steps should respect the intended information-flow semantics of portal traversal, which we define in Section V.

III. HHV SYSTEM MODEL

The implementation of portal traversal—and other HHV operations—requires access to the running execution context and to the state of the protection domains. The domain state

¹Portal traversal is the only mechanism for sharing resources, hence we do not consider shared resources to be a separate source of information flow.

resides in memory, so we introduce a simple virtual memory model consisting of a memory datatype, `VirtMem`, and basic operations on this type (such as read and write). In addition to the user state stored in the EC and memory, HHV keeps track of certain internal state, represented by the type `HHV`, that the kernel operations potentially modify. We incorporate these state components into our model using a layered monadic approach that is similar to the techniques previously applied in the construction of modular interpreters [11], [12].

Using monad transformers, we construct a monad with EC, `VirtMem`, and `HHV` state components (`ST s m` adds a state component of type `s` to monad `m`, resulting in a new monad). Every computation that runs in this monad will share a single EC, memory, and HHV structure; the operations available to computations in this monad include reading and writing to each of the state components.

```
type Kernel = ST EC (ST VirtMem (ST HHV Id))
```

We often reference values for the three state components of the `Kernel` monad together, so we introduce a new type, `KS`, to represent the kernel state.

```
data KS = KS { ec :: EC, mem :: VirtMem, hhv :: HHV }
```

Running a `Kernel` computation with an initial value for the kernel state allows us to examine the result of the computation and the new state that the computation produces. We are primarily concerned with the state effects of HHV operations, so we define a run function that returns the state produced by executing a kernel computation (throwing away the result of the computation).

```
runKS :: Kernel a → KS → KS
runKS k s
  = let ((_, c), m), h) = unId (k `runST` (ec s)
                                `runST` (mem s)
                                `runST` (hhv s))
      in KS c m h
```

The library function `runST` peels off a single layer from our monad transformer stack. Thus, we use three applications of `runST` in the definition of `runKS`, one for each state component of the `Kernel` monad. We will use `runKS` frequently in our property formulations to compare the impact that two distinct monadic computations have on the state.

IV. NONINTERFERENCE

The properties we wish to formulate of the context switch operation all deal with the absence of information flow between domains. Noninterference [1] is a mathematical formalism for expressing exactly this kind of property. Noninterference captures the idea that a given operation is not allowed to change the state of a particular domain. For example, “switch does not change the state of any domain except the source and destination” can be thought of as a noninterference property.

By employing noninterference to formalize our system, we can take advantage of the existing mathematical frameworks for reasoning about noninterference security policies [1], [2], [13], [14]. Noninterference frameworks establish a correspondence between properties of individual operations, called

unwinding conditions, and the information flow that occurs during the execution of a system. In this section, we define two generic noninterference-style properties, which we will later instantiate to capture specific information flow properties of communication in HHV. These properties are analogous to unwinding conditions, so noninterference frameworks provide us with confidence that the properties will also hold for sequences of HHV operations.

The property *NoStateEffect* (see Figure 2) captures the notion that some operation, `k`, does not modify the kernel state. However, we do not wish to say that the kernel state does not change at all, rather, we want to express that a certain component of the state does not change (such as a particular domain’s SRA or the page-table memory). For this reason, *NoStateEffect* takes a function argument, `extract`, whose role is to extract a portion of the kernel state. This extraction function is polymorphic in its return type and can perform arbitrary (pure) computation. We say that an operation has *NoStateEffect* if the extraction function produces the same result in the initial state and in the state that results from running the operation.

Note that we do not expect *NoStateEffect* to be a general property of HHV, rather, we will use *NoStateEffect* to describe the effects of particular operations on particular components of the kernel state. As a simple example, consider the `setActiveDomain` operation, which changes the running domain. Performing this operation should not change the instruction pointer stored in the running EC (among other things). In this case, the extraction function simply projects the `eip` field from the EC component of the state: $(\lambda s \rightarrow eip (ec s))$.² Having determined the extraction function, we can formalize the assertion as: $\forall s, dom. NoStateEffect(s, \lambda s \rightarrow eip (ec s), setActiveDomain dom)$.

The function `readPage` is a more realistic example of the kind of extraction function we will use in our property formulations. Given a domain and a virtual-page number, `readPage` looks up the corresponding frame number in the virtual address space of that domain.

```
readPage :: PDID → VirtIdx → KS → Page
readPage did vi s
  = let m = mem s
      (pi, _) = pageTables m did vi
      in lookupPage m pi
```

We define extraction functions for reading page-table entries (`readPageTable`) and for reading the save/restore area of a domain (`readSRA`) in a similar fashion.

Clearly, the polymorphic nature of *NoStateEffect* makes it a powerful tool for expressing a range of noninterference properties. We will use *NoStateEffect* to write many of the information flow properties of portal traversal, but in some cases we need something even more general. In particular, for successful portal traversals, we need to express that a computation may change the state, but only in a controlled

²In Haskell syntax, $\lambda x \rightarrow E$ defines an anonymous function, which will evaluate the expression `E` with the argument `x`.

$$\text{NoStateEffect}(s :: \text{KS}, \text{extract} :: \text{KS} \rightarrow a, k :: \text{Kernel } b) = \\ \text{extract} (\text{runKs } k \ s) \equiv \text{extract } s$$

$$\text{ControlledStateEffect}(s :: \text{KS}, \text{extract} :: \text{KS} \rightarrow a, k1 :: \text{Kernel } b, k2 :: \text{Kernel } b) = \\ \text{extract} (\text{runKs } k1 \ s) \equiv \text{extract} (\text{runKs } k2 \ s)$$

Fig. 2. Patterns for formulating noninterference properties of monadic Haskell programs. *NoStateEffect* describes kernel operations that do not modify a particular region of the kernel state. *ControlledStateEffect* describes kernel operations that may modify the kernel state, but only in a constrained manner. The first argument to both patterns corresponds to the kernel state. The argument called *extract* is a function that extracts a particular region of the kernel state from one or more of the kernel state components (such as the page-table memory). In the case of *NoStateEffect*, *k* is the kernel operation of interest. For *ControlledStateEffect*, *k1* is the kernel operation and *k2* is the reference computation. Note that the relation \equiv captures the denotational equivalence of two terms, rather than structural equality.

way. *ControlledStateEffect* is a generalization of *NoStateEffect* that captures this idea.

The definition of *ControlledStateEffect* (see Figure 2) is very similar to *NoStateEffect*, except that it compares the state produced by a given operation to the state produced by a reference computation. The intended use of this property is that the reference computation will perform only the allowed effect. Thus, if the resulting states are compatible, then the effects of the first computation must be limited to those effects that are explicitly allowed. The state-dependent nature of the reference computation makes *ControlledStateEffect* an instance of a dynamic noninterference property [2].

V. PROPERTIES OF PORTAL TRAVERSAL

Portal masks provide fine-grained control over the data transfer that occurs during a portal traversal. The correct implementation of this control mechanism is the fundamental property of `switch`; there should not be any information flow that is not expressly permitted by the portal masks. We divide this high-level requirement into three categories:

- **Destination State Preservation:** The value of a field only passes from the source domain to the destination domain if both the transfer mask and the pass mask allow the transfer. Only the source domain can leak information to the destination.
- **Source State Preservation:** No information flows into the source domain. The SRA of the source only changes as specified by the transfer mask of the portal.
- **Memory Preservation:** Executing `switch` does not modify the memory pages or the page-table of any domain in the system. For every domain except the source and destination, the SRA pages do not change.

We will formulate these properties using the noninterference specification patterns defined in the previous section.

A. Destination state preservation

Portal masks are the principal mechanism afforded to domains for protecting their state during a portal traversal. If either the source or the destination wishes to protect a field, then HHV must ensure that the value of that field in the source EC does not leak to the destination EC. To formalize this property, we need a mechanism for identifying protected fields and a formal way to capture that a field value is not leaked. The predicate `noLeakage` (see Figure 3) determines if a particular

field is protected; it is true if either mask blocks the transfer of that field.

We would like to use *NoStateEffect* to express the idea that the execution of `switch` does not affect the value of protected registers. Thus, we need to determine the value that a register should have when no leakage occurs. The function `loadField` performs this task, loading a single field value from a given SRA. *FieldNotLeaked*, defined in Figure 3, formalizes the desired register protection property by instantiating *NoStateEffect* with `loadField` as the extraction function and `switch` as the computation of interest.

The first argument to *FieldNotLeaked* is a function that projects a field from the context of the running EC. The second argument is the corresponding projection function for the `Mask` type (e.g., `beip` returns the mask value that corresponds to the `eip` field of a context). Parameterizing *FieldNotLeaked* in this way allows us to write a single property that can be instantiated for each field of the context. Figure 3 shows an example of one such property, *EIPNotLeaked*, which specifies that HHV does not transfer the instruction pointer register unless both the source and destination indicate that the transfer should occur.

The field-preservation properties capture information flow into the destination through the EC, which is the only component of the destination state that may legally change during a traversal. We expect the remaining components of the destination state—the saved context and non-SRA memory—to be unaffected by the execution of `switch`. We define *DestSavedContextUnchanged* to express that the saved context does not change (Figure 3), we again use *NoStateEffect*. In this case, the extraction function is simply the SRA-projection function that returns the saved context (`savedContext`). We address the memory preservation properties in Section V-C.

B. Source state preservation

A portal traversal does not permit any information flow into the source domain. However, the source state is not entirely unaffected by the traversal, because HHV will potentially save register values to the source SRA. In other words, executing `switch` modifies the source SRA in the same way as `saveContext`. We use *ControlledStateEffect* to express this property, as shown in Figure 4; `readSRA` is the extraction function, `switch` is the computation of interest, and `saveContext` is the reference computation.

```
noLeakage :: Mask → Mask → (Mask → Bool) → Bool
noLeakage transfer pass p = not (p transfer && p pass)
```

```
FieldNotLeaked(f :: Context → a, p :: Mask → Bool, src :: PDID, dst :: PDID, port :: Portal) =
  ∀ s. NoStateEffect(s, λ s → loadField (pass port) f p (readSRA dst s), switch src dst port)
```

```
EIPNotLeaked =
```

```
  ∀ src, dst, port. noLeakage (transfer port) (pass port) bEip ⇒ FieldNotLeaked(eip, bEip, src, dst, port)
```

```
DestSavedContextUnchanged =
```

```
  ∀ s, src, dst, port. NoStateEffect(s, λ s → savedContext (readSRA dst s), switch src dst port)
```

Fig. 3. Destination state preservation properties. *EIPNotLeaked* is an example of a field-preservation property built from the predicate *noLeakage* and the abstract property *FieldNotLeaked*. *DestSavedContextUnchanged* captures the property that a context switch does not change the destination’s SRA. We quantify over the kernel state (*s*), the source and destination domains (*src* and *dst*), the portal being traversed (*port*).

```
NoFlowIntoSource =
```

```
  ∀ s, src, dst, port. ControlledStateEffect(s, readSRA src, switch src dst port,
      saveContext (transfer port) (ec s) src)
```

Fig. 4. Source state preservation property: executing a context switch potentially modifies the saved context of the source domain, but all other components of the source state are unchanged. The quantified variables have the same meaning as in Figure 3.

The page-table and non-SRA memory pages of the source should also be preserved by *switch*, as we shall see in the next section.

C. Memory preservation

The only memory regions that a context switch will modify are the source and destination SRAs. We capture this notion using three properties: when *switch* executes, the page-table of every domain remains the same, the mapped memory of every domain remains the same, and the SRA memory of every domain except the source and destination remains the same. The definitions of these properties are presented in Figure 5.

Recall from Section IV that *readPageTable* projects the page-table component of the kernel state. We use *NoStateEffect* with *readPageTable* as the extraction function to express that *switch* does not modify the page-tables in the property *PageTablesUnchanged*.

Similarly, *readPage* projects a specified memory page from the kernel state. We quantify over all virtual-page numbers to express that none of the memory pages visible to user-level domains are affected by a context switch.

For the SRA memory pages, we again use *NoStateEffect*, but in this case the extraction function is *readSRA*. We quantify over domain identifiers, but exclude the source and destination because the SRA of these domains might in fact change.

VI. RELATED WORK

OS verification has a long history filled with mixed successes, an overview of which is provided by Tuch, et al. [15]. More recent verification efforts can be divided into two categories: attempts to verify low-level source code written in C/C++ and attempts to verify functional language models/implementations. The VFiasco [16] and L4Verified [17] projects both pursue the former option in their effort to prove properties of L4 [18] implementations. The results of these projects are

limited to proofs about sub-systems of L4, rather than top-level separation properties, due to the inherent complexities involved in reasoning about C and security issues in the L4 design. Furthermore, these projects do not employ a general information flow framework, which we consider to be an important contribution of our work.

The use of functional languages together with formal property specifications is a relatively recent development, but there is already a large body of work on the subject. Elphinstone, et al. are using Haskell models in the the development of seL4 [19], [20], a secure redesign of the L4 microkernel [18], and are already making progress towards verifying properties of this model. They have translated their model into Isabelle/HOL, thus guaranteeing termination, but have not yet formalized any separation properties.

The Programatica project formally verified a virtual memory model written in Haskell [21]. They have also specified an axiomatic semantics for an implementation of a safe, Haskell interface to hardware [22]. We view this work as highly compatible with ours because proving our separation properties will depend on guarantees about the behavior of the underlying hardware.

There is also existing work on the instantiation of noninterference frameworks for concrete systems. In his foundational noninterference paper [1], Rushby applied noninterference to a simple access control mechanism. Subsequently, Schellhorn et al. applied Rushby’s work to prove security properties of their generic formal model of operating systems for multiapplicative smart cards [23]. Von Oheimb used noninterference to analyze the security of the Infineon SLE66 smart card processor [13]. The key difference between our work and these earlier efforts is that the complexities of general-purpose operating systems make them more difficult to integrate with a theoretical framework. Also, the access control and smart card work did not

```

PageTablesUnchanged =
  ∀ s, src, dst, port, did, vi. NoStateEffect(s, readPageTable did vi, switch src dst port)

UserMemoryUnchanged =
  ∀ s, src, dst, port, did, vi. NoStateEffect(s, readPage did vi, switch src dst port)

SRAMemoryUnchanged =
  ∀ s, src, dst, port, did. (did /= src) && (did /= dst) ⇒ NoStateEffect(s, readSRA did, switch src dst port)

```

Fig. 5. Properties describing the absence of information flow via memory during a context switch. As with the previous properties, we quantify over the kernel state (s), the domains involved in the portal traversal (src and dst), and the portal being traversed ($port$). We introduce two new quantified variables: did is the identifier of the domain whose memory we are interested in and vi is a particular page in that domain's virtual address space.

utilize the notion of dynamic noninterference [2], which is essential for reasoning about HHV.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we formalized the information flow behavior of direct communication in a microkernel-style hypervisor using property-specification patterns derived from general noninterference frameworks. The development of these patterns is a significant advance in the realm of operating system verification because it illustrates how to apply information flow theory to a practical system.

The next step is to port our model to a theorem proving environment, such as Isabelle/HOL, and to prove the properties specified in this paper. We anticipate that the use of Haskell will make this task easier because Haskell has a well-defined semantics and there are established connections between Haskell and Isabelle. We also plan to formalize the information flow behavior of the remaining primitives of HHV using the same techniques we applied to communication.

We based the correctness and security properties of our hypervisor on a Haskell model, rather than a low-level implementation, so that we could focus on the conceptual elements of our design. Our model is based on the same specification as the actual C++ implementation, but we have yet to establish a formal connection between the two. Ideally, we would like to derive the model directly from the implementation or refine the model into an implementation. However, our immediate plans focus on the specification and verification of the critical components using the model alone.

ACKNOWLEDGMENTS

Sebastian Schönberg made significant contributions to this work by sharing his expertise in hypervisor design and implementation. We would also like to thank Mark P. Jones and Iavor S. Diatchki for their helpful suggestions regarding the presentation of this paper. This work was done during the first author's internship at Intel Research Labs.

REFERENCES

- [1] J. Rushby, "Noninterference, transitivity, and channel-control security policies," SRI International, Tech. Rep. CSL-92-02, December 1992.
- [2] R. Leslie, "Dynamic intransitive noninterference," in *First IEEE International Symposium on Secure Software Engineering*, 2006.
- [3] "seL4 web site," <http://www.ertos.nicta.com.au/research/sel4>.
- [4] B. Kauer, "L4.sec implementation: Kernel memory management," Diploma Thesis, TU Dresden, 2005.
- [5] D. Grawrock, *The Intel Safer Computing Initiative*. Intel Press, 2006.
- [6] "Programatica web site," <http://programatica.cs.pdx.edu>, 2006.
- [7] S. Peyton Jones, Ed., *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.
- [8] W. L. Harrison and R. B. Kieburtz, "The logic of demand in Haskell," *J. Funct. Program.*, vol. 15, no. 6, pp. 837–891, 2005.
- [9] B. Ford and J. Lepreau, "Evolving Mach 3.0 to a migrating thread model," in *Proceedings of the Winter 1994 USENIX Technical Conference and Exhibition*, 1994, pp. 97–114.
- [10] J. S. Shapiro, "Vulnerabilities in synchronous IPC designs," in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May 2003, pp. 251–262.
- [11] M. P. Jones, "Functional programming with overloading and higher-order polymorphism," in *Advanced Functional Programming, 1st Int. Spring School on Advanced Functional Programming Techniques-Tutorial Text*. London, UK: Springer-Verlag, 1995, pp. 97–136.
- [12] S. Liang, P. Hudak, and M. Jones, "Monad transformers and modular interpreters," in *POPL '95: Proc. 22nd ACM Symp. on Principles of programming languages*, 1995, pp. 333–343.
- [13] D. von Oheimb, "Information flow control revisited: Noninfluence = Noninterference + Nonleakage," in *Computer Security – ESORICS 2004*, ser. LNCS, vol. 3193. Springer, 2004, pp. 225–243.
- [14] J. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.
- [15] H. Tuch, G. Klein, and G. Heiser, "OS verification — now!" in *Proc. 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, M. Seltzer, Ed., 2005.
- [16] M. Hohmuth, H. Tews, and S. G. Stephens, "Applying source-code verification to a microkernel – The VFiasco project," Technische Universität Dresden, Tech. Rep. TUD-FI02-03, March 2002.
- [17] H. Tuch and G. Klein, "Verifying the L4 virtual memory subsystem," in *Proc. NICTA Formal Methods Workshop on Operating Systems Verification*, G. Klein, Ed., NICTA Technical Report 0401005T-1. National ICT Australia, 2004, pp. 73–97.
- [18] L4ka Team, *L4 eXperimental Kernel Reference Manual*, January 2005. [Online]. Available: <http://l4hq.org/docs/manuals/l4-x2-20041209.pdf>
- [19] K. Elphinstone, G. Klein, and R. Kolanski, "Formalising a high-performance microkernel," in *Workshop on Verified Software: Theories, Tools, and Experiments (VSTTE 06)*, ser. Microsoft Research Technical Report MSR-TR-2006-117, R. Leino, Ed., Seattle, USA, 2006, pp. 1–7.
- [20] P. Derrin, K. Elphinstone, G. Klein, D. Cock, and M. M. T. Chakravarty, "Running the manual: an approach to high-assurance microkernel development," in *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*. New York, NY, USA: ACM Press, 2006.
- [21] M. P. Jones, "Bare Metal: A Programatica model of hardware," in *High Confidence Software and Systems Conference*, Baltimore, MD, March 2005.
- [22] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach, "A principled approach to operating system construction in Haskell," in *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM Press, 2005.
- [23] G. Schellhorn, W. Reif, A. Schairer, P. A. Karger, V. Austel, and D. Toll, "Verification of a formal security model for multiapplicative smart cards," in *ESORICS '00: Proceedings of the 6th European Symposium on Research in Computer Security*. London, UK: Springer-Verlag, 2000, pp. 17–36.

High-Performance Microkernels and Virtualisation on ARM and Segmented Architectures

Carl van Schaik[†] and Gernot Heiser^{†‡§}

[†] Open Kernel Labs

[‡] National ICT Australia*

[§] University of New South Wales

cvansch@ok-labs.com

Abstract

This paper describes the techniques used to achieve high context-switching performance on ARM processors for the L4 microkernel and a para-virtualised Linux running on top. We examine how the previously-published techniques can be used in L4 with minimal changes to the kernel API. We also propose future API changes which make it easier to maximise memory-management performance, not only on ARM but also on architectures supporting a segmented memory model.

1 Introduction

ARM [Jag95] is a processor architecture particularly popular for battery-powered devices with moderate CPU performance requirements. It has been adopted in a wide range of applications from automotive to mobile phones, PDAs and networking gear.

Historically, most applications using the ARM architecture have been implemented on simple real-time executives with no memory protection. Increasingly, however, security and isolation requirements have driven the need for running systems with memory protection on the ARM processor. Unfortunately, many such uses suffer from high context-switching costs due to idiosyncrasies of the widely-deployed cores conforming to versions 4 and 5 of the ARM architecture (ARM v4/v5 cores).

L4 [Lie95] is a high-performance microkernel that aims to provide a minimal but efficient set of abstractions, general enough to implement almost arbitrary systems on top. It is increasingly deployed in embedded products, particularly on ARM processors, as a real-time kernel and virtualisation platform. This makes it highly important that the kernel minimises overheads on ARM v4/v5, in particular for context switches.

A number of techniques have been developed over the years that allow L4 to achieve excellent context-

switching performance on ARM processors [WH00, WTUH03]; these techniques are collectively called *fast address-space switching* (FASS).

This paper discusses work done at National ICT Australia (NICTA) and Open Kernel Labs (OKL) on enhancements to the L4 API that allow us to make the best possible use of hardware mechanisms, in particular for minimising the overheads of virtualisation. At the same time we aim to retain a high degree of architecture-independence of the API, and thus attempt to develop a model that will map cleanly to related mechanisms on architectures other than ARM, specifically PowerPC and Itanium.

This work is reflected in the evolution of the NICTA and OKL versions of the L4 API — the N-series API [NIC05], and its implementation in NICTA::Pistachio-embedded and OKL4, which are descendants of L4Ka::Pistachio. We describe the changes made to the API and implementation and then demonstrate the results on our Wombat server, a mostly architecture-independent para-virtualised Linux system running on top of L4 [LvSH05].

We also provide an overview of recent and forthcoming API changes aimed at improving the suitability of L4 for resource-restricted embedded systems, particularly systems with small memories. In particular, we show that the FASS techniques will enable a significant reduction of the memory required for the ARM's hardware-walked page-tables.

2 ARM v4/v5 architecture

Since this paper deals mostly with the ARM v4/v5 MMU and ways to provide general abstractions for its use, we will focus our overview of the architecture on aspects of its MMU.

It is important to note that the ARM v6 architecture introduces a number of changes to the MMU which avoid many of the problems of v4/v5. However, ARM-v6 compliant cores tend to be significantly larger and thus more expensive and resource-hungry than v5 implementations. Therefore, v5 cores will continue to re-

*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.

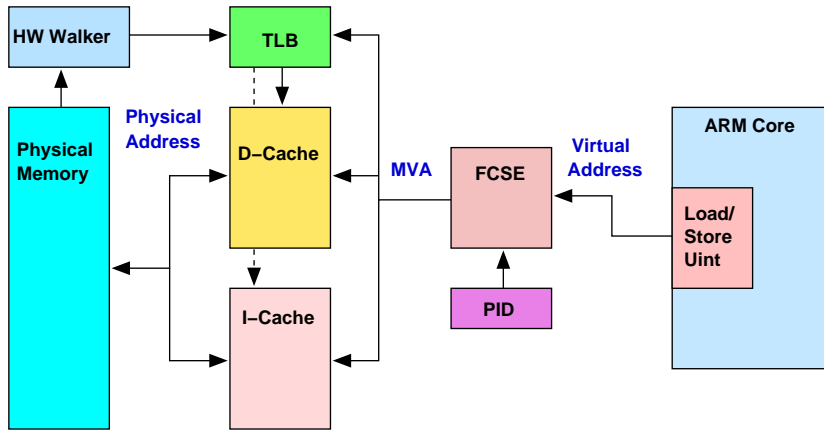


Figure 1: ARM MMU structure.

main popular for low-power and low-cost applications for years to come.

Furthermore, ARMv6 is backwards compatible with v5, and the same mechanisms can still be employed. While they are no longer required on v6 cores, owing to changes in the memory architecture, the techniques discussed in Section 3.3 will still be beneficial for supporting sharing and for reducing the kernel’s memory overhead.

From here on we will simply refer to the “ARM architecture” when talking about the ARM v4/v5.

Shown in Figure 1, on the surface, ARM implements a fairly traditional MMU structure. The MMU consists of a translation-lookaside buffer (TLB), a split or unified L1 Cache and a hardware page-table walker. However, on further inspection it becomes evident that the ARM MMU contains a number of features that contribute to performance problems. It also provides mechanisms for avoiding high overheads, but they are difficult to use.

2.1 Caches

The ARM architecture specifies that caches are virtually indexed and virtually tagged (VIVT caches). This is done to reduce cache latencies by removing the requirement for a TLB lookup before accessing the cache. Furthermore, the caches do not contain any information that associates cache lines with address spaces.

This means that data cannot be kept in the cache when switching to another address space which uses the same virtual addresses. Otherwise, that address space could read valid cache data belonging to another address space, and worse, write data into another address space’s memory.

Since Unix-like operating systems use the same address-space layout for all user processes, specifically mapping the text segment to a fixed address in each address space, operating systems typically flush the cache on each address space switch. The direct cost of flushing the cache depends on the cache size and memory band-

width, but typically costs 10 to 100 times more than the operating cost of the address space switch.

Interestingly, ARM caches keep the physical address of each cache line in a secondary hidden tag, which gets updated in the background after a virtual address has been translated in the TLB. This removes the need of looking up the address in the TLB when writing back data. But since the tag is not used during data access from the cache, it is of no use for avoiding cache flushes.

2.2 TLB

The ARM TLB is a typical content-addressed memory (CAM) for translating virtual addresses to physical addresses. Its entries also control cache behaviour, by specifying the cache write policy and whether the cache is to be bypassed for a particular page.

Unlike most processor architectures, the ARM TLB does not contain an address-space identifier. Consequently, operating systems avoid mixing mappings from different address spaces in the TLB, and hence flush the TLB on each context switch.

While the direct costs of flushing the TLB is low, the indirect cost of reloading the TLB through page faults is significant, and has a major performance impact.

2.3 Page Tables

Since ARM processors use a hardware page-table walker, the page-table format is fixed by processor design.

The ARM architecture has a two-level page-table format and supports four page sizes (1MiB, 64KiB, 4KiB and 1KiB). The top level is a 16KiB array containing 4096 4-byte entries, each covering 1MiB of the 4GiB address space. Each top-level entry may either represent a single 1MiB page or may be a pointer to a second-level table containing smaller pages. The second level may either be a 1KiB array containing 64KiB and 4KiB page sizes or a 4KiB array which additionally supports 1KiB

page sizes. The table is always indexed with the smallest supported page size; entries for larger-sized pages are replicated so that the hardware walker requires only a single lookup.

In spite of the use of a hardware walker, TLB reload costs are high, as page-table pointers are physical addresses which bypass the cache. Hence reloads typically require two memory accesses. This is a main reason for the high indirect costs of TLB flushes.

2.4 Domains

The ARM architecture has an interesting feature in that in addition to protection information, regions of memory at a 1MiB granularity can be tagged with a *domain ID*. Altogether there are 16 domain IDs provided by the hardware. The processor contains a domain access control register (DACR) which contains an array of 2-bit permissions for each domain number. The permissions field allows a domain to be marked as *no-access*, *manager mode* or *client mode*. *No-access* prevents access to any page in this domain, regardless of page permissions, *manager mode* bypasses all page permissions and allows full RWX access, and *client mode* respects the permissions of the pages tagged with the domain.

Typically, a process is given access to only a single domain such that trying to access pages tagged with a different domain causes a domain fault. Since the DACR contains a domain-mask array, it is possible to give more than one process access to the same domain. We call this *domain sharing*.

2.5 Fast-Context-Switch Extension

In v4 of the architecture, ARM introduced a feature called the *fast context switching extension* (FCSE) which was originally developed for supporting Windows CE [Mur98]. This feature uses a 6-bit (7-bit on v5) *process identifier* (PID) to re-map the bottom end of the address space.

The re-mapping works by replacing the 7 most significant bits of the address, if they are zero, by the contents of the PID register, effectively mapping the lowest 32MiB of address space into a different 32MiB slot. This re-mapping happens prior to the virtual address translation and the resulting *modified virtual address* (MVA) is seen by the TLB and caches. The feature thus allows up to 128 small address spaces, each using a traditional Unix-style layout, to be transparently re-mapped to another slot in virtual memory, which avoids address-space overlap between processes, and thus prevents cache alias problems.

FCSE avoids the need for flushing caches and TLB on address-space switches, and the scheme is used by Windows CE [Mur98]. Without further effort, however, this leads to a loss of memory protection.

FCSE can be used safely if domains are used as a poor-man's address-space tag for the TLB [WH00] —

the basic idea behind FASS. An implementation of this scheme in Linux has demonstrated context-switching costs reduced by as much as a factor of 50 [WTUH03].

3 Kernel Implementation

FASS has recently been implemented in the L4Ka::Pistachio [L4K] implementation of the V4 API. This kernel is the base for the NICTA versions of L4, called NICTA::Pistachio-embedded and OKL's version, called OKL4. The implementation of FASS in OKL4 is discussed here.

3.1 L4Ka::Pistachio

As indicated above, FASS is based on using ARM domains as address-space tags for TLB entries. The L4 implementation uses the same basic approach as described in [WTUH03]. At context-switch time, the DACR is reloaded by a mask disabling access to pages belonging to the address space that is being switched out, and enabling access to pages belonging to the address space that is being switched in. In this scheme, a *caching page directory* (CPD) is used as the global top-level page-table used by the hardware walker and 1MiB top-level entries are copied in and out from the per-address-space page-tables. The CPD thus points to leaf page tables of multiple address spaces concurrently.

Flushes are then only required if the new address space does not have a valid domain and no free domains are available. In this case, the kernel needs to free a domain to preempt. If two address spaces overlap, this is detected by hardware thanks to the access mask in the DACR, and the kernel then flushes TLB entries and caches selectively.

The kernel uses three data structures to keep track of domain usage: a bitfield of dirty domains, a bitfield of dirty *user TCBS* (UTCBS) and a bitfield of CPD domain ownership. In L4, each thread has a *UTCB*, a datastructure which is shared between the kernel and user which serves as an efficient means for threads to communicate with the kernel.

The dirty-domains bitfield is used to keep track of domains which may have data present in the cache. Whenever domain ownership of an ARM section changes, the kernel checks whether the domain of the original section is dirty and flushes the TLB and caches if that is the case. If the domain is clean, it is safe to leave the cache alone, and the kernel only flushes the TLB. Whenever the cache is flushed, all the domains are marked clean. A clean domain is marked dirty when switching to an address space which has access to pages in that domain.

The implementation minimises changes to the kernel API for the ARM architecture. The L4 V4 API specifies that the UTCB area of an address be user configurable. The kernel, however also needs to access the UTCB of all threads in the system. Typically, L4 accesses UTCBs directly in the kernel heap and users ac-

cess them through a mapping in their address space. On ARM, this presents a problem due to cache aliasing: Owing to the VIVT caches, virtual aliases present the same problem as overlapping address spaces described earlier in [Section 2.1](#). To avoid this, whenever L4 accesses a UTCB it performs a check to test whether the UTCB's user mapping is in the CPD and tagged with the user's domain. If this is true, the kernel accesses the user mapping, otherwise it accesses the address in its heap. Accessing the UTCB in the heap, however, may cause the user's UTCB mapping to present stale data when it is faulted in at a later stage. The kernel thus keeps a UTCB-dirty bitfield which it uses to indicate whether a UTCB of a domain has been accessed via a kernel mapping. Since this only happens when the domain's UTCB mapping was not in the CPD; a subsequent user access will cause a domain fault and the kernel knows to flush the dirty UTCB data from the cache.

Lastly, the kernel keeps a bitfield of domain ownership for the CPD. This is used during domain recycling to optimise flushing the CPD of entries belonging to a particular domain.

One compromise the ARM implementation had to make to the API was to map the *kernel information page* (KIP) to a fixed address common to all address spaces, rather than let user-level code determine the KIP address. This was required to allow the kernel to prevent cache aliases from occurring in the KIP.

With these features, L4Ka::Pistachio was able to provide a simple, essentially unmodified API to the user. Unmodified applications can run, with potential performance loss due to domain faults on conflicting virtual address ranges, but no loss of correctness. Furthermore, applications only need to adhere to a simple set of memory-layout guidelines in order to make full use of fast address-space switching. Iguana is a good example of such a system, since it uses a single address space (SAS) model where no conflicting virtual addresses are allowed, except when using shared data.

3.2 NICTA N2 API

While the original L4Ka::Pistachio implementation worked well for a some classes of systems, it had a number of limitations relevant to memory-constrained embedded systems. Some of those limitations could not be addressed without API changes. As NICTA and OKL are engaged in deploying L4 in a wide range of embedded applications, we needed an API that supported implementations optimised for such systems.

Commercial realities demanded a smooth and incremental migration path, and we therefore decided to evolve the existing API in several steps. This also allows us to provide a reasonable migration path towards the forthcoming seL4 API [EDE07]. The first step was the N1 API released in October 2005, which was followed by the N2 API (not yet released at the time of writing but available from the public source repository). This sec-

tion describes some of the changes provided by the N2 API relative to the X2 API on which L4Ka::Pistachio is based.

3.2.1 PID relocation

A simple extension to the API (and one that does not affect other architectures) is a provision for associating an ARM PID value with each address space, utilising the FCSE (or PID relocation) feature of the processor. If non-zero, this PID forces the lower 32MiB of the address space to be remapped as described in [Section 2.5](#).

This raises the issue that now within an address space two different virtual addresses, one smaller, the other larger than 32MiB, can reference the same data. In order to simplify the interface, the kernel treats all user addresses passed in or out of the kernel as MVAs (i.e. remapped virtual addresses). This specifically applies to addresses specifying mappings or fault addresses. However, the kernel will not modify user-visible thread state, such as the PC. This implies, for example, that a page fault triggered by an instruction fetch may show a fault address different from the faulting PC value (by 32MiB times the PID value).

3.2.2 UTCB addresses

One X2 API feature that is problematic on ARM processors is the user-determined mapping address of UTCBs. The kernel must prevent inconsistencies in the UTCB resulting from aliasing, and should also ensure that UTCB accesses do not result in performance degradation resulting from domain conflicts. It was therefore decided to allow the kernel to determine UTCB locations on some architectures, specifically ARM.

In our implementation of the N2 API on the ARM, the kernel reserves a 256MiB region of global virtual address space for use as UTCB areas. Each address space is allocated a 1MiB area corresponding to a single CPD entry for its UTCBs. This allows for up to 256 address spaces in the system, which is sufficient for most embedded systems (and this limit could be made a kernel configuration option). That way the kernel can guarantee that no cache aliases occur in the UTCB area and the kernel and user processes can access the same UTCB address safely. This can be achieved without having to keep track of "dirty" state.

L4 still needs to handle domain faults on UTCBs, as it frequently needs to access the UTCB of a thread other than the current thread (e.g., during IPC). Faults will be generated when the domain of the third-party UTCB has been recycled.

3.2.3 Shared pages

Another issue with the X2 API is that it does not support efficient sharing of memory between address spaces on the ARM. As each address-space's mappings are tagged with a (at any given time) unique domain ID, accesses

to shared memory would always result in domain mismatches and hence flushing of caches.

The obvious way of dealing with sharing on the ARM is to use a separate domain ID for shared pages, and configure the DACR to provide access to all sharers. Implementing this cleanly, without making the API too architecture-specific and implanting too much policy in the kernel, is tricky, however. For the N2 API, we therefore settled for a simpler approach that is almost as effective for the problems at hand, but is definitely seen as an interim solution. The idea is to let the system’s policy layer identify sharing environments, called *vspaces*.

A standard way of efficiently sharing data on the ARM is to introduce global address-space regions for sharing. Iguana, which is our policy and resource-management component that is the core of L4-based systems, provides such a single-address-space layout in a way similar to Opal [CLFL94], Mungi [HEV⁺98] or Nemesis [LMB⁺96]. In such a system, there exists a single, system-wide mapping from virtual to physical addresses, and as such no cache aliasing problems exist.

While a similar approach is also used by QNX and Windows CE, this is done at the expense of foregoing memory protection, a tradeoff we are not willing to make. Instead we allow address spaces with non-conflicting layouts to be tagged with a common *vspace ID*, which the kernel can use to avoid cache flushes.

Specifically, on a domain fault, the kernel compares the *vspace ID* of the faulting address space with that of the address space owning the domain that is used to tag the faulting page in the CPD. If the two *vspace IDs* match and are non-zero, the kernel assumes a non-conflicting address-space layout and does not flush the cache. The TLB is still flushed, ensuring that each address space can only access data explicitly mapped to it by its pager.

It is the responsibility of the policy layer to ensure that this is used securely, the kernel only provides the mechanisms. Incorrect use of the primitives by a pager can therefore lead to data corruption (not different from accidentally mapping the wrong page), but not to security violations beyond what the pager could cause by other misuse of mappings.

In our Iguana system, all processes running in the single address space use a *vspace ID* of one. Iguana also supports *external address spaces* (mostly used for legacy emulation and not intended to share memory), these all use a *vspace ID* of zero, and therefore require cache flushes on domain faults.

If shared memory regions used separate (shared) domain IDs, the TLB flush could also be avoided (and shared pages would be mapped by shared TLB entries), at the expense of a more complex implementation, and increased contention for domain IDs. This is planned for the future, as discussed below.

A complete TLB flush can be avoided where the conflicting CPD entry is a 1MiB superpage. In this case, a single mapping can be flushed from the TLB, eliminat-

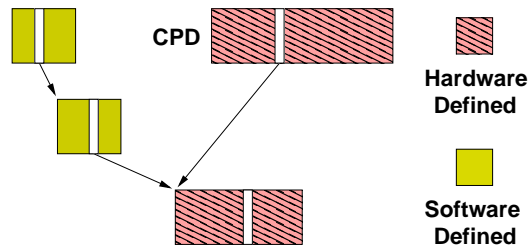


Figure 2: L4 software compressed page tables with the CPD.

ing the indirect costs of flushing.

3.2.4 Cache control

The L4Ka::Pistachio kernel does not provide a clean API for cache management. Cache management is important for applications that share data with explicit cache aliases. A typical example is a Unix-like operating system server, which maps pages to client address spaces and needs to copy data in from and out to the client.

For the N2 API, a new system call called *CacheControl* was created. This allows address-space pagers (threads with privileges to map and unmap pages in a client) and clients to perform various cache manipulation operations, such as cache-range flushing, as well as more complex control such as cache-line locking and cache setup/partitioning.

3.3 Planned Enhancements

A number of enhancements are proposed for the next versions of OKL4. These aim to provide better support on ARM for shared domains as part of a general model for improved address-space management, memory-usage optimisations and further performance improvements.

The ARM’s hardware-walked two-level page tables are quite expensive in terms of memory overhead, particularly in an embedded system with a significant number of small address spaces. Each address space has a 16KiB top-level page table, which for small processes will only contain a handful of valid entries — 16KiB of wasted space per process.

Owing to our implementation of FASS, the top level of an address-spaces page table is never walked by hardware, the page-table walker only accesses the CPD. This means that software is free to implement a different page-table structure, as long as the leaf page tables remain unchanged. Hence we can replace the page directory by a data structure more suitable for small address spaces, such as a simple linked list or a two-level page table with small fanout as shown in Figure 2. It is even possible to use different formats for different address spaces, as long as the root of the data structure indicates the format.

While this change is a pure implementation optimisation, other planned changes will be visible at the API level. This includes making address spaces first-class citizens (again); the present X2 approach of naming address spaces indirectly via threads allocated in them never felt quite right, and interferes with other improvements.

Note that the compressed page-table structure discussed above will still be useful for reducing kernel memory overhead on ARM v6 cores, even though the CPD will no longer be required for avoiding cache flushes.

A more drastic change aims at further improving the support for memory sharing, in a way that abstracts over mechanisms found in several different architectures. This is discussed in [Section 3.4](#), and will also be beneficial on ARM v6.

3.4 Segmentation API Proposal

While the unit of hardware-supported memory sharing is the page, in typical scenarios the logical unit of sharing is a more arbitrary region of contiguous address space. Examples are producer-consumer buffers and memory-mapped files.

Furthermore, several architectures (ARM, PowerPC [MSSW94] and Itanium [Int00]) provide hardware support for sharing, including the ability to share a single TLB entry for shared pages, an attractive way to reduce TLB pressure. While previous studies [WTUH03, CWH03] could not find a significant performance impact from TLB sharing, those were done in Linux. A microkernel-based system tends to have orders of magnitude higher context-switching rates than Linux. It also makes much more intense use of shared memory between user-level processes, as OS servers (such as Wombat) accessing client memory run at user level. Hence, the ability of the TLB to concurrently map the working sets of several processes is much more important in such a system. Similarly important is the ability to share page-table subtrees for shared memory regions in order to minimise kernel memory overheads.

The present API has no provisions that allow the kernel to utilise such hardware features or share page tables. An abstraction that identifies shared regions could achieve that, and at the same time significantly reduce the number of kernel entries required for setting up shared regions. An obvious abstraction, which maps directly on the hardware mechanisms in some architectures, is segmentation. Before presenting the model, we will first describe the relevant architectural features in PowerPC and Itanium.

3.4.1 PowerPC Segmentation

A number of commonly used PowerPC processors support segmentation, including the IBM's POWER processors and the newer embedded PPC603 cores. Traditional operating systems, as well as L4, have under-

utilised the segmentation architecture of these processors, essentially turning segmentation into address space identifiers. A memory-management model that supports PowerPC-style segmentation has been desired by the L4 community for some time.

In the case of the POWER processors, segmentation leads to a two-step address translation. The high-end bits of the CPU-issued *effective address* form an *effective segment ID*, which is used as an index into a per-process segment table to obtain a *virtual segment ID*. The latter is a system-wide unique identifier for a segment of up to 256MiB in size. It is combined with the remainder of the address to form the *virtual address*. The combined virtual segment ID and per-segment page number is translated into a physical address using a page table. That translation is cached in a TLB, while the segment translation is cached in a *segment lookaside buffer* (SLB). The latest generations of POWER processor also feature a device called an *ERAT* which caches the complete address translation.

On the PowerPC, addresses can efficiently share segments by using effective segment IDs that map to the same virtual segment ID. As the TLB is indexed by the virtual address, shared segments naturally share TLB entries. Since the segment table contains protection bits, this is possible even if the address spaces have different access rights to the segment (e.g., in a consumer-producer scenario).

3.4.2 Itanium Region Registers

The Itanium architecture also divides the virtual address space into a (much smaller) number of segments, called *regions*. The top three bits of the 64-bit virtual address form the *virtual region number*, which selects one of 8 region registers, containing a global 24-bit *region ID*. Regions serve as a generalised form of ASID tags on TLB entries, but can also be used for very coarse-granular sharing. For example, Linux [ME02] reserves one region for shared libraries, which means that they share TLB entries.

The Itanium region scheme only supports sharing with uniform access rights, and only at the same address for all participants. However, there is another feature, called *protection keys*, which allows the OS to further restrict access rights to pages on a per-process base.

3.4.3 Segmentation example

Consider the example in [Figure 3](#) which consists of three address spaces: *A*, *B* and *C*. These address spaces each have their private mappings, as well as the shared regions *x* and *y* in their address spaces.

In the current L4 API, constructing such a system is possible, however L4 does not take advantage of special hardware support for segmentation and TLB sharing. On ARM processors, the problem is compounded by significant performance overheads: the TLB needs to be flushed whenever a context switch occurs between

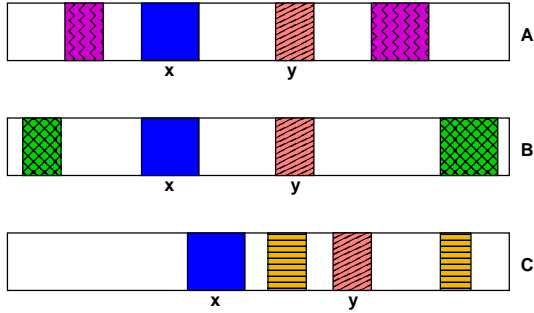


Figure 3: Three address spaces, with shared segments x and y .

these address spaces. Address spaces A and B could be placed in the same vspace since their private mappings do not conflict. The private mappings of address space C , however, conflict with those of the other spaces, and would require cache flushes.

If only address space A and B are considered on ARM, the preferred approach would be to tag the private mappings of address space A with a unique domain ID, and use a different domain ID for the private mappings of B . Assuming suitable alignment (to 1MiB ARM sections), a third domain ID could be used to tag the shared regions x and y ; that domain would be enabled in the DACR whenever A or B are running. On a context switch between the two address spaces, the DACR would be the only addressing/protection information that would need to be updated. The TLB entries mapping x and y would be valid for both address spaces. However, a switch to address space C , which maps the shared regions at different addresses, would require a cache flush.

On PowerPC, the same effect can be achieved by using separate segments for regions x and y (subject to appropriate alignment). In this case, TLB entry sharing is even possible between all three address spaces. Cache flushes and TLB flushes are never necessary on that architecture, irrespective of address-space layout.

On Itanium, sharing could be achieved by allocating x and y in separate regions. Given the coarseness of regions, this is not a very feasible approach. Alternatively, one region can be reserved for shared memory (as in Linux). This forces shared regions to use a fixed virtual address, hence rules out the layout of address space C (but without restrictions on usage of other regions, in particular no need for non-conflicting mappings outside the sharing region).

Protection keys provide additional flexibility, but in any case, TLB entry sharing is only possible if the shared region is mapped to a unique address [CWH03]. As on the PowerPC, TLB or caches never need to be flushed on context switches.

The three architectures are representative of hardware support for sharing. We can see that an abstraction of a contiguous segment of memory as a unit of sharing

could be used to exploit the hardware support mechanisms offered, as long as the usage is compatible with the requirements of the underlying hardware. The kernel needs to be able to detect the case where the use of the mechanism allows the use of the hardware support mechanisms, and otherwise needs to ensure correctness (at the expense of performance). It is then up to the policy layer to ensure that the hardware mechanisms are used.

3.4.4 API Design

We propose to extend the L4 API with a generic abstraction of the segmentation and TLB sharing capabilities of some modern processor architectures. We introduce the concept of a *segment*, and a system call called *SegmentControl* for their manipulation.

A segment is a contiguous page-aligned range of virtual memory which can be shared by (mapped into) one or more L4 address spaces at an arbitrary (page-aligned) address. A segment logically has its own page tables, and changes to a segment's mappings are visible in all address spaces sharing the segment. It is up to user-level code to create segments which are compatible with the limitations of the underlying hardware architecture in order to make full use of the API. Incompatible segment layout, or using segments on a processor without TLB sharing or segmentation support, will result in L4 emulating the API.

Segments are mapped and unmapped into address spaces as indivisible units; that is, they are either shared in their entirety or not at all. Mapping pages within segments uses the existing L4 API for mapping pages to address spaces. Pages are not mapped directly to segments, rather, segments are populated implicitly by mapping into the virtual-address range of a segment in a target address space. In addition to permissions of individual pages, there are per-segment permissions: An address space's access rights on a particular page is the intersection of the rights with which the page is mapped and the rights with which the segment is mapped.

In the N2 API, mapping segments is a privileged operation.¹ The handling of page faults is unchanged, they are delivered to the faulting thread's page fault handler.

SegmentControl provides four basic operations:

1. Segment creation.

A segment of a specified size is created, and assigned a unique, caller-specified, segment ID. The segment is initially empty (i.e. does not contain any mappings).
2. Segment deletion.

Deletion removes a segment and its page tables. The segment and its pages are unmapped from all address spaces to which it has been mapped.

¹This will change in the seL4 API, which supports the delegation of privileges.

3. Segment mapping.

This allows a segment to be mapped into an address space. The base address of the segment and the access rights are specified by the caller. If different pagers map the same segment (not possible in the N2 API as mapping is restricted to the privileged root task) then those pagers must communicate the segment ID once the segment has been created.

4. Segment unmapping.

Unmapping removes a segment from a particular address space, and implies unmapping all its pages from that address space.

3.4.5 ARM Implementation

On ARM, the kernel will allocate a new domain ID, different from any per-address-space domain ID, when a segment is first mapped into an address space. If the segment is subsequently mapped into other address spaces at the same base address, its domain is enabled (in the DACR) for all those address spaces. Provided that the segment is mapped with full access rights in all participating address spaces, TLB entries will be shared and no TLB or cache flushes are required on context switches.

In order to minimise bookkeeping and the amount of policy in the kernel, domains will only be allocated to segments that are aligned to MiB boundaries (i.e., CPD entries). Such segments, of course, are described by a range of leaf page tables, and can therefore naturally be represented as a small top-level page table. Instead of being tagged with an address-space ID, they are tagged with a segment ID. In terms of domain management (e.g., domain recycling), segments will be treated like address spaces.

For segments which do not conform to the above restrictions (alignment and permissions), the kernel can implement the same functionality as for the *vspace* abstraction in the present kernel.

Theoretically the kernel could detect cases where several segments are shared between the same address spaces, and use the same domain ID for all of them. While this could reduce pressure on domain IDs, this would put unnecessary policy into the kernel, as the same effect can be achieved by proper user-level management of segments.

PowerPC and Itanium implementations are left as an exercise for the reader.

4 Wombat Implementation

Wombat [LvSH05] is a port of the Linux 2.6 kernel to the L4/Iguana operating system and runs on ARM, i386 and MIPS64 processors. Since the port has been done such that L4/Iguana is treated as a new architecture, the portability to other L4-supported architectures is increased.

On the ARM platform, the original port of Wombat suffered from very poor performance. This was mostly due to inefficient implementation of Wombat and insufficient abstraction provided by the L4 API.

The main reason for the performance problems were that the Wombat server and its user processes reside in separate L4 address spaces, and thus Wombat cannot access the client's address spaces. Compounding the problem is the memory layout of Unix-style processes, which create address conflicts between clients, causing cache and TLB flushing to occur. Also, since Wombat, unlike native Linux, cannot directly access the clients' address spaces, it needs to access the base pages from which clients' pages are mapped. Wombat has mappings for all memory that is available to the Linux subsystem, including those in use by Linux client processes. However, such pages are mapped at different addresses in Wombat's address space than in the client's, resulting in cache-alias problems that need to be managed by Wombat.²

Two ARM-related changes have been made in Wombat to reduce these performance problems:

Firstly, Wombat was updated to use the *CacheControl* API introduced in the N2 API, which allows finer control of cache flushing. This reduces overheads by allowing Wombat to flush cache lines selectively when accessing user memory.

Secondly, Wombat has been modified to make use of the PID-relocation extensions to L4, also introduced in the N2 API. In contrast to FASS on native Linux, we opted to simplify the implementation and restrict user applications to a 32MiB address space (the system will presently refuse to load larger programs). Although seemingly small, this is more than adequate for most embedded applications. PID relocation is used by allocating a PID register value for each Linux user process. This PID subsequently remaps each user address space to a higher 32MiB slot. Since L4 handles domain allocation and reuse transparently, no notion of domains is needed in Wombat. For PID-relocation support, the only changes required were translating all user addresses to MVAs when dealing with L4.

4.1 Future Work

The proposed API needs to be inspected to test its suitability on a larger range of machine architectures including IA32. Once done and suitably revised, it will make up part of future NICTA L4 APIs.

We plan to modify Wombat to take advantage of the segmentation concept in order to match native Linux's ability to directly access user address spaces. On machines that support it, Wombat will additionally be able to share TLB entries with its clients. On ARM, TLB sharing is possible under the proposed API and perfor-

²While Wombat itself runs inside Iguana's single address space, binary-compatible Linux applications each run in their own *external address space*, using the standard Linux address-space layout.

Table 1: Lmbench performance of native Linux vs. Wombat *before* and *after* FASS optimisations. *Gain* shows the relative improvement due to FASS. *Relative* shows the performance of optimised Wombat (*after*) relative to native Linux.

Latency	native [μ s]	before [μ s]	after [μ s]	gain	relative
ctx 0k	190.8	207.9	6.48	32.1	29
ctx 1k	218.7	204.8	6.43	31.9	34
ctx 4k	257.7	209.3	7.15	29.3	36
fifo	377.0	1146	80.0	14.3	4.7
pipe	378.4	1146	81.6	14.0	4.6
unix	764.5	1440	107.5	13.4	7.1
syscall	0.82	5.27	4.0	1.32	0.21
fork	4334	28918	5706	5.07	0.76
exec	4600	29473	6400	4.61	0.72
Bandwidth	[MB/s]	[MB/s]	[MB/s]		
file IO	39.4	2.12	12.43	5.86	0.32
mmap IO	106.7	105.4	106.1	1.01	0.99
mem rd	416.0	412.8	416.1	1.01	1.00
pipe	10.15	6.59	15.3	2.32	1.51
unix	24.23	11.32	11.32	1.00	0.47

mance will be greatly improved due to the removal of cache alias problems. Furthermore, with PID relocation for fast context switching, Wombat on ARM L4 should outperform native Linux in many areas.

5 Evaluation

We evaluated the performance benefits of the implementation of FASS in NICTA::Pistachio-embedded by running Wombat and the lmbench suite [MS96].

All results were obtained on a PLEB2 [SPH05] machine which comprises an Intel PXA255 XScale processor running at 400MHz and with 64MiB of RAM. The XScale has an ITLB and DTLB, each fully associative with 32-entries. It has a 32KiB instruction cache and data cache, both VIVT and 32-way associative.

Lmbench system latency and bandwidth results are shown in Table 1. The first set of results in *latencies* shows context switching latency between user processes. The second set shows hot-potato latencies and the third shows raw system call overhead and process creation overheads. The final set of numbers shows the memory bandwidth of various Lmbench tests.

The context switching numbers show the dramatic effect that FASS has on address-space switching, with the para-virtualised Wombat outperforming native Linux by an average factor of 30. Even the hot-potato benchmarks, which copy data between processes, benefited significantly. This is particularly noteworthy, given that the Wombat implementation presently supports no shared domains, and thus needs to flush caches for all data copying operations into and out of Wombat. The high numbers for the Wombat *before* hot potato bench-

mark reflect the vast overhead of the previous cache flushing implementation in L4.

The system call overhead shows the overhead of the para-virtualisation implementation. L4 needs to switch address spaces from the user’s context to Wombat’s context, whereas native Linux simply enters kernel mode via a trap. Process creation times have been greatly improved in Wombat, however they still present a 31% to 39% overhead over native Linux. This is a result of sharing (where Wombat accesses user memory) leading to domain conflicts that result in cache flushes, an effect that will be eliminated by implementing the segment API.

In the bandwidth benchmarks, it is clear that file-IO performance is presently poor. Although the cache-API changes improved Wombat significantly, domain sharing is still needed to approach native Linux’s kernel-user copy performance. Memory accessed by user mode only (*mmap* and *mem rd*), displays identical performance to Linux.

Interestingly, pipe bandwidth on Wombat surpasses Linux. This is due to the benefits of fast context switching outweighing the cost of cache flushing, while in the unix benchmark, the cache flushing outweighed the fast context switching. Using shared domains should further boost these numbers in Wombat.

6 Related Work

QNX [Hil92] is a microkernel system that provides a message passing primitive which may involve a context switch between the message sender and receiver. Like L4, this results in a higher frequency of context switches compared to other kernels.

To alleviate the cost of context switching on ARM, QNX uses FCSE (PID relocation) with support for up to 63 concurrent processes that are limited to 32MiB of virtual memory. Any shared objects are mapped uncached, since the objects reside at different MVA’s. Hence, memory-access cost is traded against context-switching overheads.

QNX also uses memory above 2GiB as a global shared memory area that processes can use to map and shared objects which can be cached. It is unclear (but seems unlikely) that QNX uses domains for address space protection, as opposed to simply switching page-tables and flushing the TLB.

In contrast, L4 on ARM, as described in Section 3.2, supports a maximum of 256 address spaces. Furthermore, each address space supports over 400 threads. L4 does not restrict the address space like QNX. Applications may choose to use FCSE and are treated no differently to those not using it. Furthermore, each L4 address space can use up to 3.25GiB of virtual memory. Applications may use any address freely, however if domain conflicts occur due to address space conflicts in the CPD, L4 will flush the cache and TLB on each domain fault.

Windows CE uses FCSE [Hur, Mic], however not much information is available about its implementation. The latest version, Windows CE 5.0, supports 32 address spaces each limited to 32MiB in size which are located in the first 1GiB of virtual memory. The next 1GiB area is used for global objects and memory mapped files. The top 2GiB is kernel address space. Address spaces are protected, but it is not clear if domains are used or the TLB is flushed on context switches. TLB flushing is suspected since it is possible to disable memory protection in Windows CE for performance reasons.

FASS has been implemented in Linux [WH00, WTUH03] previously and reported vastly increased improved context switching times over standard Linux. However, the maintainers, who were offered the FASS patches several times, did not seem to consider the obtained performance improvements significant enough. This has resulted in the paradoxical situation of Wombat (virtualised Linux) on ARM outperforming native Linux.

EROS [SSF99] exposes the logical page-table structure to applications and allows mapping of complete subtrees. This inherently leads to efficient sharing of address-space regions (effectively superpages) and can naturally support segmentation hardware and share TLB entries on such architectures. On ARM v4/v5, it would still require a mechanism for associating subtrees with domains. The seL4 API [EDE07] will similarly expose a generalised mapping data structure, which will ease the implementation of the mechanisms discussed here.

7 Conclusions

This paper discussed the present implementation of fast context switching in L4 on ARM v4/v5 processors, and identified its limitations. The implementation produced the impressive result that context-switching overheads of a virtualised Linux system are 1–2 magnitudes less than in standard Linux. However, system calls that access client memory are still up to a factor of three more expensive in the virtualised system. We proposed implementation strategies which will eliminate this extra cost, and proposed a *segment* abstraction as an API mechanism that will map well those strategies. An additional benefit is that this will support the efficient use of segmentation hardware.

References

[CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *Trans. Comp. Systems*, 12:271–307, 1994.

[CWH03] Matthew Chapman, Ian Wienand, and Gernot Heiser. Itanium page tables and TLB. Technical Report UNSW-CSE-TR-0307, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, May 2003.

[EDE07] Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. A memory allocation model for an embedded microkernel. In *1st MIKES*, pages 28–34, Sydney, Australia, Jan 2007. NICTA.

[HEV⁺98] Gernot Heiser, Kevin Elphinstone, Jerry Vochtelloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Softw.: Pract. & Exp.*, 28(9):901–928, Jul 1998.

[Hil92] Dan Hildebrand. An architectural overview of QNX. In *USENIX WS Microkernels & other Kernel Arch.*, pages 113–126, Seattle, WA, USA, Apr 1992.

[Hur] Tim Hurman. Exploring Windows CE shellcode. http://www.pentest.co.uk/documents/exploringwce/exploring_wce_shellcode.html, last visited 25 January 2007.

[Int00] Intel Corp. *Itanium Architecture Software Developer's Manual*, Feb 2000. <http://developer.intel.com/design/itanium/family>.

[Jag95] Dave Jagger, editor. *Advanced RISC Machines Architecture Reference Manual*. Prentice Hall, Jul 1995.

[L4K] L4Ka Team. L4Ka::Pistachio kernel. <http://l4ka.org/projects/pistachio/>.

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

[LMB⁺96] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *J. Selected Areas Comm.*, 14:1280–1297, 1996.

[LvSH05] Ben Leslie, Carl van Schaik, and Gernot Heiser. Wombat: A portable user-mode Linux for embedded systems. In *6th Linux.Conf.Au*, Canberra, Apr 2005.

[ME02] David Mosberger and Stéphane Eranian. *IA-64 Linux Kernel: Design and Implementation*. Prentice Hall, 2002.

[Mic] Microsoft. Windows CE memory architecture. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcecoreos5/html/wce50conMemoryArchitecture.asp>, last visited 19 October 2006.

[MS96] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *1996 USENIX Techn. Conf.*, San Diego, CA, USA, Jan 1996.

[MSSW94] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann, 1994.

[Mur98] John Murray. *Inside Microsoft Windows CE*. Microsoft Press, 1998.

[NIC05] National ICT Australia. *NICTA L4-embedded Kernel Reference Manual Version N1*, Oct 2005. <http://ertos.nicta.com.au/Software/systems/kenge/pistachio/refman.pdf>.

- [SPH05] David C. Snowdon, Stefan M. Petters, and Gernot Heiser. Power measurement as the basis for power management. In *2005 WS Operat. Syst. Platforms for Embedded Real-Time applications*, Palma, Mallorca, Spain, Jul 2005.
- [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.
- [WH00] Adam Wiggins and Gernot Heiser. Fast address-space switching on the StrongARM SA-1100 processor. In *5th Aust. Comp. Arch. Conf*, pages 97–104, Canberra, Australia, Jan 2000. IEEE CS Press.
- [WTUH03] Adam Wiggins, Harvey Tuch, Volkmar Uhlig, and Gernot Heiser. Implementation of fast address-space switching and TLB sharing on the StrongARM processor. In *8th Asia-Pacific Comp. Syst. Arch. Conf*, Aizu-Wakamatsu City, Japan, Sep 2003. Springer Verlag.

Automated Object Layout Optimization in a Portable Microkernel

Uwe Dannowski

System Architecture Group, Universität Karlsruhe (TH), Germany

Abstract—In a portable microkernel, the increasing diversity of target configurations can lead to software complexity problems. Insufficiencies of current kernel programming techniques manifest in excessive preprocessor use for code selection, in code duplication, and in suboptimal performance. Object-oriented programming can solve the portability problems. However, the language implementation of inheritance often enforces a memory layout of objects that is governed by inheritance relations, not by access patterns, resulting in suboptimal cache usage on the kernel’s critical path.

In this paper we present an automated approach to eliminating inheritance-induced overheads in selected performance-critical data structures. We combine class flattening and profile-guided data member reordering and heavily rely on microkernel characteristics. Evaluation in the L4 microkernel indicates that we can use fine-grained class hierarchies in the kernel at no cost and still optimize for the target system, allowing for portable yet efficient microkernels.

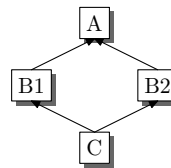
I. INTRODUCTION

Microkernels can and must be fast. A successful microkernel must have minimal cache footprint and execution time [11]. Any unnecessary overhead reduces the performance of the system on top of the microkernel. At the same time, microkernels, at core of the system, must be maintainable and portable — traditionally considered a contradiction to the first objective [10].

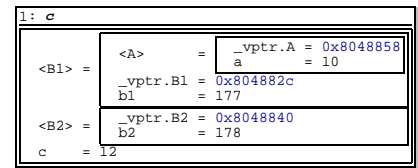
The diversity of target configurations is the root cause of portability problems. Modularity is the key to configurability and thus to portability: Code that is specific to one configuration or a group of configurations must be separated from generic code. Placed in different modules they are combined when building for the particular configuration. We followed this principle in our initial implementation of the L4Ka::Pistachio microkernel [17]. We identified configuration dimensions, such as the architecture, the processor family member, the platform, the amount of parallelism, or even the kernel API, and let the build system choose the appropriate fragments for the particular point in the configuration space. However, we found that insufficiencies of current kernel programming techniques lead to excessive preprocessor use for code selection, code duplication, or suboptimal performance.

Object-oriented programming strongly encourages modularity [4]. With inheritance, classes can be constructed from other classes, enabling fine-grained combination and stepwise refinement of functionality. In earlier work [6] we showed how inheritance can be used to construct classes for kernel objects from configuration-specific super-classes to manage the configuration diversity in the microkernel.

This approach solves problems regarding code duplication and code selection. However, the language implementation of inheritance imposes high run-time overheads that are unacceptable in a microkernel. Whereas in a simple class, the order of data members (or fields) in the class declaration determines the layout of the object [7], the object memory layout of classes using inheritance is governed by the inheritance relationship, and unnamed pointers (vtable pointers) may be added to the object — both in support of dynamic polymorphism. Figure 1 illustrates the object memory representation of a C++ class using inheritance.



(a) A class hierarchy with a virtual base class. Each class contains an `int` data member.



(b) The object layout of C. Addresses increase by four per line. Data members are located in inseparable subobjects. Three vtable pointers are added, inflating the object from 16 to 28 bytes.

Fig. 1. Object memory representation in C++ (gcc)

The superclasses `A`, `B1`, and `B2` form subobjects in the object of the inheriting class `C`, allowing to treat an object of class `C` as an object of a superclass. To call the appropriate version of a virtual function for an object without statically knowing its exact type, some sort of run-time type identifier is required. The compiler therefore adds a pointer to a table of function pointers, the virtual function table, to each object and invokes the function via a double indirection [16].

Both, the inheritance-dictated placement of data members and the introduction of vtable pointers take away control over the memory layout from the programmer, resulting in poor cache usage on the kernel’s critical path. The optimal layout of data structures accessed on the kernel’s critical path depends on many factors, such as the architecture [10], [13], the particular choice of algorithms in the kernel, and the workload atop the kernel.

Dynamic polymorphism is, however, not necessary when inheritance is solely used as a tool to efficiently compose classes. This is exactly the case for the way we proposed to construct kernel objects from a set of configuration-specific superclasses. Code using such a class makes no assumptions about how the class was constructed, and inheritance can be

safely removed without changing the interface to the class.

In this paper we present an automated approach to eliminating inheritance-induced overheads in selected performance-critical kernel data structures. We combine class flattening and profile-guided data member reordering, and heavily rely on microkernel characteristics to customize the optimization process. Class flattening removes inheritance, turns virtual functions into normal functions, and prepares the class for data member reordering. Member reordering arranges data members in the class declaration for optimal cache usage. Profiling determines data member access patterns on the kernel’s critical path under workload. We integrate flattening, profiling, and member reordering into the kernel build process as illustrated in Figure 2.

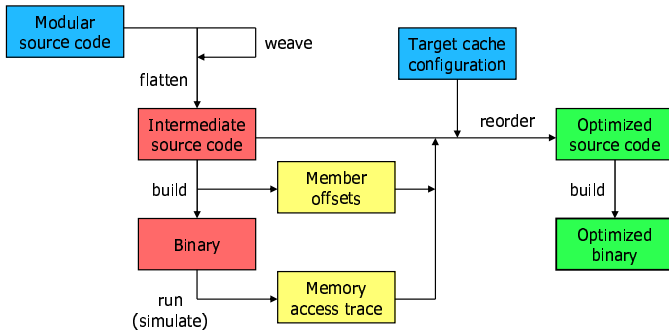


Fig. 2. The optimization embedded into the kernel build process. A kernel with flattened classes is built and profiled to collect access patterns on the critical path. In a second step, members in the flattened classes are reordered for optimal cache usage.

Class flattening and data member reordering are whole program transformations. We implement them as source-to-source transformations that replace the preprocessing stage in the usual per-file preprocess-compile-assemble build process. Not integrating both these transformations into the compiler has the benefit of compiler independence, and does not require a custom-built toolchain for using them.

In previous work [6], we have already demonstrated how class flattening can be successfully used to eliminate the overhead of virtual function calls in a microkernel. Due to space constraints, we will omit a detailed discussion of class flattening in this paper and focus on field reordering and profiling in the context of a portable microkernel.

The remainder of this paper is structured as follows: Section II briefly introduces class flattening. In Section III we discuss strategies for data member reordering in microkernel objects. Section IV presents a profiling approach tailored at extracting access patterns for the critical path from the microkernel. In Section V we evaluate our optimization approach. Section VI discusses related work and Section VII concludes.

II. TRANSPARENT CLASS FLATTENING

Class flattening produces a single flat class from a class hierarchy by copying all inherited members from base classes into the most derived class and removing the inheritance relationship. Following standard lookup, overriding functions and

shadowing data members in a derived class hide their inherited versions, so that shadowed data members and overridden functions become inaccessible and can simply be removed.

Class flattening can be applied as a transparent optimization, such that the code using the class does not need to be modified. The conditions that enable transparent class flattening are discussed in [14].

III. DATA MEMBER REORDERING

Data member reordering attempts to optimize the memory layout of compound objects (records, structs, classes) according to certain criteria by manipulating the location of data members inside the binary object representation. Type-safe languages abstract from the physical storage layout and leave placement of members to the compiler or run-time system. All layout optimizations are thus automatically valid with respect to program correctness. In contrast, type-unsafe languages expose the locations of members and allow (limited) control of member placement, for example by order of appearance in the compound type declaration. Compound types that are used to represent data structures with a predefined layout such as device registers, hardware-walked tables, API data types, or structured storage in files must not have their data members reordered. However, when code makes no assumption about the internal organization of a type, program correctness is not affected by reordering members. In such cases, data member reordering is transparent to the code using the type and can be applied automatically.

Reordering can even be applied automatically in the presence of programmer-written assembly code that references objects defined in the high-level language, as it is often found in kernels, for example in inline assembly fragments and entry and exit stubs of exception handlers and system calls. Such assembly code can automatically adapt to changing object layouts when it uses symbolic instead of literal offsets to address fields in objects. The respective symbols can be automatically derived from the high-level object definition at build time, appear as constant displacements in the assembly code, and thus will cause no run-time overhead.

Data member reordering maximizes spatial locality of compound data structures larger than a cache block in order to optimize cache behavior. Memory reference traces provide information about a program’s memory access behavior. Data members accessed contemporaneously are placed close together to minimize the number of cache blocks used.

The mapping of data members to cache blocks depends on the location of the member in the object and the location of the object relative to the cache block boundaries. Allocating objects at cache block boundaries or at a fixed offset to them allows to minimize the number of cache blocks used. For arbitrarily allocated objects, the worst case cache footprint after reordering is one cache block more than the minimum. The performance gained by aligning objects at cache block boundaries may well make up for the potential waste of memory due to fragmentation. Furthermore, alignment restrictions of data members may already dictate minimum alignment of a

compound object. Also, when an object is known to be aligned at its size (or the next higher power of two), an object’s base address can be derived by masking a pointer to an arbitrary location inside the object.

The remainder of this section describes strategies driving data member reordering that have not been considered by previous work. These strategies may lead to higher optimization potential or can simplify the reordering algorithm. Depending on hardware configuration and usage scenarios, not all strategies are necessarily applicable at the same time. Strategies may also, at least partially, contradict each other. It is left to the actual reordering algorithm to choose or prioritize them.

1) *Object Roles*: Based on the observation that objects of a class show similar access behavior [5], previous work does not distinguish objects of the same class when reordering fields. This certainly holds true for programs that operate on a large number of objects such as nodes in a tree. A microkernel, however, typically manipulates only very few objects during its short, performance-critical operations.

Objects of the same class that are referenced during an operation may actually expose very different access patterns for their fields. In the example shown in Figure 3, all fields of a class worth two cache lines are accessed in a first object, X, whereas only half of the fields (i.e., worth one cache line) are accessed in a second object, Y. Ignoring differences in access characteristics of different objects may result in four cache lines referenced for both objects (Figure 3(a)). The minimum of three cache lines can be achieved by clustering the fields accessed in the second object into one cache line within the cluster of fields accessed in the first object (Figure 3(b)).

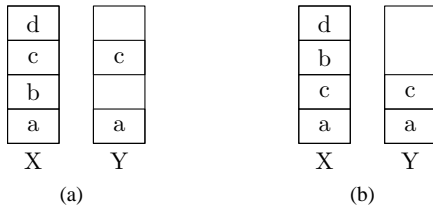


Fig. 3. Two objects, X and Y, of the same class are accessed with different characteristics (a). All fields a, b, c, and d are referenced in X, but only fields a and c are referenced in Y. The cache footprint for accessing both objects can be minimized by field reordering if the objects’ access patterns are considered separately (b).

Although all objects of a class share one internal layout, considering access patterns to different objects of the same class separately may yield a higher optimization potential.

2) *Field Access Mode*: The set of fields of a data structure that are referenced during an operation can be divided into the two subsets of fields that are only read and fields that are written. Fields that are read as well as written belong to the written set.

Assuming a write-back cache, the number of cache lines marked dirty by an operation has no direct influence on this operation’s execution time (assuming no self-interference occurs.) Instead, deferred write-back of dirty lines will penalize

completely unrelated code. While not beneficial for the current operation, minimizing the number of dirty cache lines may improve overall performance.

A minimum number of dirty lines can be achieved by packing the written fields closely together within the referenced fields and aligning them on a cache line boundary.

3) *Field Alignment*: Proper alignment of fields can be a matter of performance (penalties due to cache-line splits) or, worse, a matter of correctness (for example, unaligned accesses with LDR and STR instructions on ARM).

However, strict natural alignment of fields is unnecessary as long as all accesses are aligned. For example, a 64-bit integer can safely be 4-byte aligned when it is only ever accessed in 32-bit words. The requirement for natural alignment can be relaxed when the generated code is *known* to be safe, i.e. by configuring the compiler to not use so-called multimedia instructions. Operating systems kernels rarely and microkernels never contain such complex instructions because of the extremely expensive management of the associated hardware state.

Relaxed alignment requirements for large fields increase the flexibility in placing these fields and may simplify the placement algorithm or allow a higher level of optimization.

IV. DETERMINING FIELD ACCESS PATTERNS

Reordering fields for optimal cache usage requires precise information about field accesses. The actual code that accesses fields is not very interesting; the memory accesses it generates carry the required information. To drive optimization as described in the previous section, field access information must include the order in which fields are accessed, the access mode (read or write), and the access width.

Profiling the actual kernel with workload on top has a major advantage over analyzing the kernel source code: the programming language (or languages), compiler, optimization level, etc. determine the resulting kernel and its data structures, but they are of no concern for the process of gathering field access information. Also, the set of fields that are accessed on the critical path can be a rather small subset of the fields that can possibly be accessed by the kernel source.

The various methods for analyzing a running program are more or less suitable for recording memory references of kernel code on the critical path: Statistical or event-based sampling easily identifies hot paths, but requires instruction and register analysis to infer the target of a memory operation. Instrumentation provides exact information, but — like sampling — requires substantial infrastructure: Code for logging and extracting the data from the kernel reside in (and pollute) the space the target code runs in. In contrast, an extensible full system simulator can execute the unmodified target kernel and its workload without any infrastructure to the system to be profiled. A simulator extension that collects and exports profiling data is likely to be portable across various simulation targets.

A slowdown of the target system due to run-time overhead of profiling may result in false identification of critical paths.

For example, a network server as workload may experience massive packet losses and behave differently, marking other paths as critical. However, these problems can be side-stepped by replacing the actual workload with a workload simulator causing a representative mix of kernel activities.

A. Microkernel Specifics

Complete system address traces are huge and require significant amounts of time for postprocessing. Often only a few seconds of program execution result in gigabytes of trace data. Field access information can be extracted from a full address trace. However, customized tracing targeted at the specific problem of collecting field access information for field reordering in a microkernel can significantly reduce the amount of trace data and the required processing.

The key to reducing trace data is to aggressively customize the tracing process by incorporating knowledge about the target. Part of this knowledge is inherent in the way microkernels are designed and used, part is available in the kernel source and/or configuration information.

1) *Processor Mode*: Kernel objects store state information pertaining to API objects or kernel-internal resource management. Kernel objects are accessed by kernel code. Code that accesses kernel objects is executing in the processor's privileged mode. Consequently, for collecting access information to kernel object fields, the tracing facility needs to consider memory accesses only while the processor is executing in privileged mode.

2) *Path Length*: Microkernel invocations can be thought of as separate, short runs of the program "microkernel", interspersed with executions of user code. Performance-critical system call handlers in a microkernel are rather short, typically in the order of tens or a few hundreds of instructions. With such a limited code path length, a complete trace of one kernel invocation is limited in size, too. For example, all paths taken through the L4Ka:Pistachio microkernel during a run of the pingpong IPC benchmark perform between 2 and 85 accesses to kernel objects.

3) *Similarity*: Kernel invocations that perform the same operation on different kernel objects produce similar traces. For example, a trace of an IPC system call transferring three words between threads *A* and *B* will not differ from a trace for that IPC call between threads *C* and *D*, except for the thread identifiers and hence the respective kernel objects being referenced. Short traces with an expected high similarity can be efficiently processed and compressed online instead of generating a complete trace for offline analysis.

4) *Number of Objects*: Often-called and thus performance-critical microkernel invocations typically reference only very few kernel objects. For example, a simple IPC message transfer between two threads in the L4Ka:Pistachio microkernel involves two, at most three thread control blocks (TCBs). More complex operations involving many kernel objects, such as address space deletion, tend to be invoked less frequently.

5) *Address Ranges*: The target classes for field reordering are known in advance and so is the size of objects of

these classes. Addresses of statically allocated kernel objects are known at kernel build time. Addresses of dynamically allocated objects can only be determined at run-time, but may be easy to track in certain cases. For example, almost all L4 kernels store TCBs in a linear virtual array. At the time of writing, only one L4 kernel [13] allocates thread control blocks dynamically from the kernel heap. However, it then stores their addresses in a statically allocated table. Memory references can be filtered by address range immediately to further analyze only references to objects of target classes.

B. Precise Tracing for Field Reordering

Complete memory reference traces of programs are precise in the sense that they do not omit information. However, they often contain large amounts of useless information. In contrast, field affinity graphs [5] and member transition graphs [9] store only pairwise temporal information about field accesses. Prior research has shown that such pairwise information is theoretically insufficient for finding an optimal field placement [15], and has suggested to keep complete traces when the sequence of memory references is short.

The remainder of this section describes a tracing approach for collecting field access information to drive field reordering for selected target classes in a microkernel. The tracing facility performs aggressive online compression of memory reference trace data to customize tracing by exploiting the microkernel specifics described above. For static customization, the tracing facility uses information from various sources: definitions in the kernel source, addresses from the kernel binary's symbol table, and debug information from the kernel binary. This information is embedded when the tracing facility is built.

The tracing facility produces sequences of field references for different kernel invocations and their frequency of occurrence, whereby invocations that differ only in the addresses of referenced objects are considered identical. These sequences contain all the necessary information for field reordering.

1) *Address Filtering and Type Inference*: Memory references are filtered by processor privilege mode and address range as discussed in the previous section, so that the tracing facility receives only memory references to kernel objects that are objects of a target class for field reordering. From the address of the memory reference, the type of the object accessed can be inferred. Along with the information about the memory access, the address filter delivers the base address and the type of the referenced object.

Supporting large padding between objects in an array is necessary as this space is often abused. For example, most L4 kernels keep the kernel stack of a thread in the unused part of the memory block (usually 1KB or 2KB) that is allocated for each TCB in the linear virtual array of TCBs.

2) *Address Abstraction*: The actual addresses of referenced objects are not relevant for field reordering. However, accesses to fields of different objects still need to be tracked separately to allow optimizing for differing field usage patterns.

To distinguish between the kernel objects used during an invocation, the tracing facility assigns sequential object

numbers as different objects are encountered. Objects with different addresses that are used in the same place in similar invocations will be assigned the same object numbers: For example, the first TCB referenced during an IPC operation in the L4Ka::Pistachio microkernel belongs to the target thread of the send phase, while the second TCB referenced belongs to the source. Substituting object numbers for object addresses abstracts from the actual object in favor of an “object role.” The number of objects is small so that object addresses can be tracked efficiently.

Memory references are converted to quadruples (n, o, s, m) , with n being the number of the distinct object instance encountered since kernel entry (not the actual address of it), o the offset of the reference into that instance, s the access size, and m the access mode (read vs. write.)

3) *Per-class Sequences*: Using the type information from address filtering, quadruples are recorded in sequences of accesses since the kernel was entered. For every field reordering target class a sequence of references to objects of that class is built.

When exiting the kernel (or on the next entry), the sequence is compared with previously recorded sequences. On a match, a counter associated with the matching sequence is increased. Otherwise, the sequence is added to the list of known sequences with a counter value of one. Runaway sequences of long-running operations in the kernel (idle loop, kernel debugger, etc.) are cut off when reaching an unreasonable length.

4) *Sequence Weights*: The value of an access sequence’s counter in relation to the sum of all counters represents the weight of that access sequence in the profile. Without information about the actual code paths taken, the sequences describe precisely the access patterns to fields in the class and the probability of the pattern during the tracing session. The sequence with the highest weight should be used to determine a new field ordering.

A sequence with a lower weight may be a subset of a sequence with a higher weight in terms of field footprint. That is, optimization goals do not contradict, and optimizing for the latter also optimizes for the former, although potentially not as much as possible. By comparing only the footprint, not the sequence of accesses, inclusion signals a possibility for merging both sequences, thereby increasing the weight of the more frequent sequence.

V. EVALUATION

We evaluate data member reordering for kernel objects in the context of the L4Ka::Pistachio microkernel. Our workload is the standard L4 pingpong IPC benchmark which sends simple IPC messages back and forth between two threads. The measurement system is a 450MHz Intel Pentium III processor with a cache line size of 32 bytes. The kernel is configured to use the assembly implementation of the IPC path (the so-called “fastpath”) whenever it sees fit. To simulate cache pressure from user code, we inserted a WBINVD instruction at the

entry point of the IPC path. This instruction writes back dirty cache lines before invalidating all data caches.

We apply automatic field reordering to the `tcb_t` class that stores the kernel state of an L4 thread and is thus used heavily during an IPC operation. Member access patterns for the class are collected in the Simics extensible full system simulator [20] using a custom profiling extension as described in Section IV. The `reorder` tool, sharing its code base with the `collapse` class flattening tool [14], performs field reordering as a source-to-source transformation.

On the fastpath, a kernel with an optimized `tcb_t` class transfers IPC messages 21–25 cycles faster than the original kernel, about the cost of a cache miss on all levels without prior write-back. A cache analysis of the original kernel, shown in Figure 4, supports this: There is an outlier referenced data member in the fifth cache line of the destination TCB. Our field reordering algorithm moves all referenced members to the start of the class declaration and thereby reduces the number of cache lines for the destination TCB from three to two.

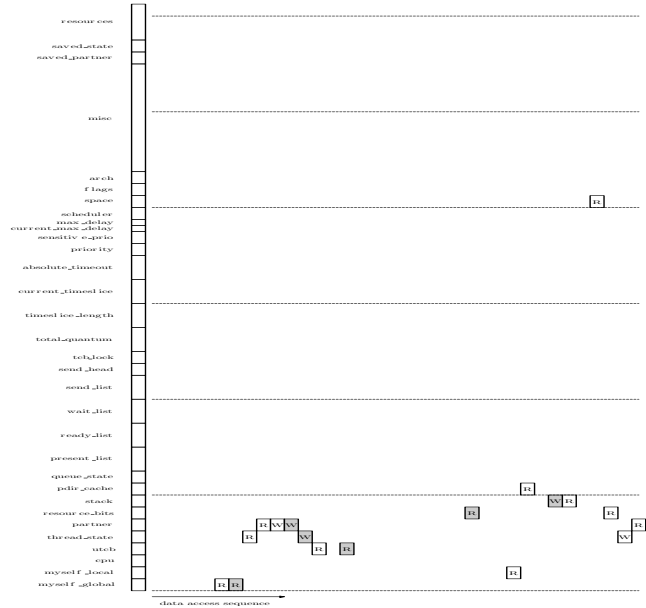


Fig. 4. Accesses to members of the `tcb_t` class on the IPC system call path. Time progresses in units of memory accesses from left to right. The object layout is shown vertically, with thin horizontal lines marking cache block boundaries. R=read, W=write, gray=source TCB, white=destination TCB.

The memory layout of the `tcb_t` class in the original kernel has been manually optimized by the kernel developers to use as few cache lines as possible. Yet, its layout was not optimal for the particular workload we happened to choose in our evaluation: IPC between threads in different address spaces — the most frequent kernel operation in well-structured microkernel-based systems.

Inheritance will result in a much less compact memory layout of the referenced members and thus in many more cache misses. However, we have not yet completed the transformation of the Pistachio source code to use fine-grained

inheritance to the full extent described in the introduction. For the purpose of evaluation we instead sort data members alphabetically by their name, assuming that we achieve a similarly suboptimal layout as inheritance would produce. For a kernel with alphabetically sorted `tc_b_t` data members, we measured IPC times 240 cycles worse than for the original kernel.

Our evaluation shows that field reordering optimizes the memory layout of kernel objects for minimum cache usage. By automatically optimizing for the particular workload, it is superior to manual optimization, which is precluded by using inheritance anyway. By enabling the fastpath in the kernel we showed that our optimization applies not only to C++ code but to assembly code as well.

VI. RELATED WORK

The *flat form* of a class was introduced by Meyer [12] in the context of Eiffel. Meyer sees two uses for the flat form: inspection of the full feature set of a class by a developer, and distribution of a class without its history. Bellur et al. [1] describe a class flattening tool for C++, targeted at eliminating virtual functions calls. An automated approach for variable flattening (replacing the type of a variable with the flattened version of the type) is suggested, but considered infeasible due to the high cost of a full data flow analysis. Bellur et al. also suggest use of class flattening in source code browsing to enhance program understanding, and as a debugging aid, because execution no longer jumps up and down in the class hierarchy. Beyer et al. [2] discuss the impact of inheritance on software metrics like size, coupling, and cohesion. Binder [3] applies class flattening to reduce complexity in the context of software testing. In previous work [6], we used class flattening to completely eliminate the run-time overhead of virtual functions calls in a portable microkernel. In this work, we apply class flattening to produce a flat version of a class whose memory layout can subsequently be optimized by member reordering.

Truong et al. [19] introduced *field reordering* as a technique to improve the cache behavior of dynamically allocated data structures in C. Truong leaves determining the optimal layout to the programmer, because “At present, the automatic detection of the most frequently used fields of a structure is beyond the possibility of current compiler technology.” In combination with a second optimization technique, instance interleaving, Truong reports speedups of 1.08–2.53. Chilimbi et al. [5] and Zatloukal et al. [21] describe algorithms for field reordering in C. Based on profiling input and static analysis, a tool produces recommendations for new field orderings that need to be verified and implemented manually. Chilimbi et al. constructs a field affinity graph for every structure type. Nodes represent fields and edge weights are proportional to the frequency of contemporaneous field accesses. Fields with high temporal affinity are placed near each other; no assumption is made about structure alignment on cache-line boundaries, as this “can only be determined at run time”. Zatloukal et al. use member transition graphs, where edges represent transition

probabilities and cache line survival probabilities. From the graph, cache hit probabilities can be determined for any member ordering. Based on the initial address trace, the new ordering is subjected to a cache simulation to report the reduced cache miss rates. Chilimbi et al. report performance improvements of 2–3% after reordering five of the most frequently used data structures in Microsoft’s SQL Server whereas Zatloukal et al. achieved only 1.3% with optimizing seven different structures. By focusing on type-safe programming languages such as Oberon, Kistler and Franz [9] can automate field reordering. They identify memory interleaving and cache line-fill buffer forwarding as source of different latencies for the words in a cache line after a cache miss. Kistler and Franz also discuss optimizing the layout of derived objects. They reorder only the fields introduced in a derived class, because encapsulation often restricts access to inherited fields and thus reduces temporal relations between fields from different levels. Kistler and Franz report combined speedups of 3%–96% for their layout optimizations.

Data packing for a given block size using pairwise information is NP-hard [8], [18]. However, for complete access traces, for example from extremely short sequences that are reused many times, an algorithm can find the optimal layout in exponential time [15]. Algorithms targeted at specific access patterns can be more efficient [22].

Except for type-safe languages, all field reordering approaches merely produce suggestions for a new ordering and require manual checking and implementation. We build on programming conventions, class use restrictions, and programmer’s knowledge to automate field reordering for C++, including rewriting the class declaration.

VII. CONCLUSION

In this paper we describe an automated approach to optimizing the object memory layout of performance-critical classes in a portable microkernel. Inheritance, used to improve kernel portability through modularity, dictates suboptimal object layouts unsuitable for a microkernel’s critical path. We combine transparent class flattening and profile-based field reordering to optimize classes composed from many small, configuration-specific classes. We rely on microkernel characteristics to automate and aggressively customize the optimization process. Evaluation indicates that we can eliminate inheritance-related overheads. We enable the use of fine-grained class hierarchies in the kernel at no cost and can automatically optimize for the target system, allowing for portable yet efficient microkernels.

REFERENCES

- [1] Umesh Bellur, Al Villarica, Kevin Shank, Imram Bashir, and Doug Lea. Flattening C++ classes. Technical Report TR-92-23, New York CASE Center, Syracuse NY 13244, August 21 1992.
- [2] Dirk Beyer, Claus Lewerentz, and Frank Simon. Impact of inheritance on metrics for size, coupling, and cohesion in object oriented systems. In R. Dumke and A. Abran, editors, *Proceedings of the 10th International Workshop on Software Measurement (IWSM 2000): New Approaches in Software Measurement*, LNCS 2006, pages 1–17. Springer-Verlag, Berlin, 2001.
- [3] Robert V. Binder. Testing object-oriented systems: a status report. *American Programmer*, 7(4):22–28, April 1994.

- [4] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 282–290, Washington, DC, 1992. IEEE Computer Society.
- [5] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI'99)*, pages 13–24, New York, NY, USA, 1999. ACM Press.
- [6] Uwe Dannowski. Managing code complexity in a portable microkernel. In *Proceedings of the ECOOP Workshop on Programming Languages and Operating Systems at ECOOP 2004 (ECOOP-PLOS'04)*, Oslo, Norway, June 2004.
- [7] International Organization for Standardization (ISO). *ISO/IEC 14882:1998(E) Programming Languages — C++*, September 1998.
- [8] Ken Kennedy and Ulrich Kremer. Automatic data layout for distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4):869–916, 1998.
- [9] Thomas Kistler and Michael Franz. The case for dynamic optimization: Improving memory-hierarchy performance by continuously adapting the internal storage layout of heap objects at run-time. Technical Report 99–21, University of California, Irvine, May 1999.
- [10] J. Liedtke. On μ -kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.
- [11] J. Liedtke. μ -kernels must and can be small. In *5th International Workshop on Object Orientation in Operating Systems (IWOOS)*, pages 152–155, Seattle, WA, October 1996.
- [12] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.
- [13] Abi Nourai. A physically-addressed L4 kernel. BE thesis, University of NSW, Sydney 2052, Australia, March 2005.
- [14] Jan Oberländer. Applying source code transformation to collapse class hierarchies in C++. Study Thesis, System Architecture Group, University of Karlsruhe, Germany, December 2003.
- [15] Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages (POPL'02)*, Portland, OR, January 2002. Extended abstract.
- [16] Bjarne Stroustrup. Multiple inheritance for C++. In *Proceeding of the Spring '87 European Unix Systems User's Group Conference*, pages 189–208, Helsinki, Finland, May 1987.
- [17] System Architecture Group. The L4Ka::Pistachio microkernel. White paper, Karlsruhe University (TH), May 1 2003.
- [18] Khalid Omar Thabit. *Cache management by the compiler*. PhD thesis, Dept. of Computer Science, Rice University, Houston, TX, 1981.
- [19] D. N. Truong, François Bodin, and André Sez nec. Improving cache behavior of dynamically allocated data structures. In *Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques*, pages 322+, October 1998.
- [20] Virtutech Inc. Simics — a full system simulator, 1998–2006.
- [21] K. Zatloukal, A. Corduneanu, R. E. Ladner, V. Grover, and S. Meacham. Improving cache performance by structure reordering. Extended Abstract, November 1998.
- [22] Chengliang Zhang, Yutao Zhong, Mitsunori Ogihara, and Chen Ding. Harness of modeling data locality and a sampling approximate approach. Technical Report TR 877, Computer Science Department, University of Rochester, September 2005.

A Memory Allocation Model For An Embedded Microkernel

Dharmika 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 inter-process 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 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 under-utilisation, 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 re-assigned 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 preemptible. 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

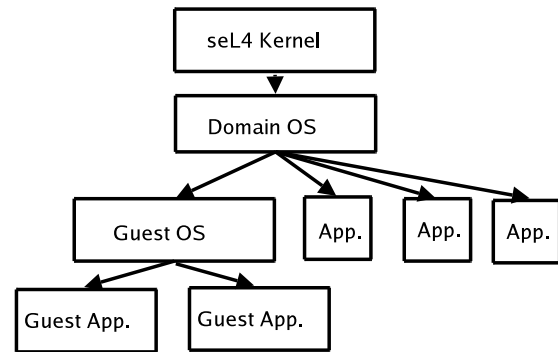


Figure 1: Sample system architecture.

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 were 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.

Haerberlen 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 provide 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 sharing¹. 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 Haerberlen 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/>.

¹A 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.

L4-Based Real Virtual Machines

– An API Proposal –

Sebastian Biemueller and Uwe Dannowski
System Architecture Group
Universität Karlsruhe (TH), Germany

Abstract—Virtual machines (VMs) recently regained attention as a solution to problems not only in high-performance computing, servers, and desktops, but in embedded systems as well. For example, network-enabled embedded systems use virtual machines to provide hardened subsystems for banking, encryption, and digital rights management.

Virtual machine systems and microkernels share a common set of goals such as reliability, security, isolation and, flexibility, so that integrating VMs and microkernels is a promising approach. In fact, modern microkernels already provide the abstractions and mechanisms necessary to cater for virtual machines.

In this paper we show how virtual machine concepts map to the concepts of a microkernel, the L4 microkernel. We identify shortcomings of the current kernel API with respect to virtual machine support and propose a minimalistic set of extensions.

I. INTRODUCTION

The microkernel approach is an ideal construction principle for the design of embedded operating systems. It reduces the amount of code that is executing in privileged mode and generally improves security, reliability, and verifiability of the system — aspects crucial for embedded systems [1]. Furthermore, the modular system structure on top of the small kernel contributes flexibility and diversity to the operating system design space and allows, for instance, an untrusted open-source web browsing component to run safely besides a closed-source banking module, or an entertainment console system to run simultaneously with a system component controlling mission-critical special-purpose hardware.

Application of the microkernel-based system construction principle to existing software requires porting. However, software reuse is an important aspect especially in embedded systems where the whole system is often designed for decades whereas rapid development of hardware often obsoletes the underlying platform after only a few years.

Full virtualization bridges this gap. By creating the illusion of a physical machine, a virtual machine monitor can provide a stable interface to software inside the virtual machine despite any changes in the underlying hardware. This is also its weak point: With isolation at the granularity of complete machines, flexible, modular, and efficient systems are hard to build. The virtual machine approach lacks design principles for the construction of the virtual machine monitor, the software inside the virtual machine, and the virtual machine system as a whole.

In this paper we propose the construction of a virtual machine system based on the principles of microkernels.

We separate the virtual machine monitor into a necessarily privileged part, the hypervisor, and an unprivileged part, the user-level monitor. The hypervisor and a microkernel are similar enough to justify integration of both. The resulting system comprises a microkernel that also provides for virtual machines and a microkernel-based system on top that includes components for maintaining the virtual machines, as illustrated in Figure 1.

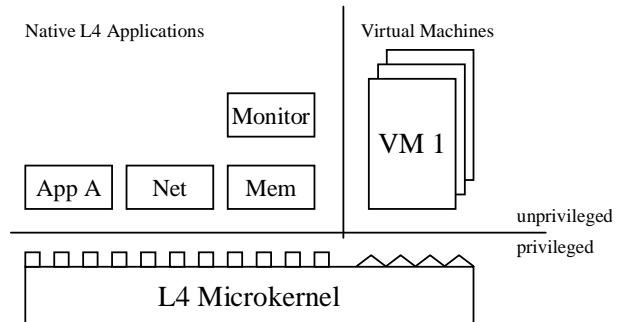


Fig. 1. Native microkernel applications including the unprivileged part of the virtual machine monitor run side-by-side with virtual machines.

Such a system combines the best of both worlds: Virtual machines provide the stable, compatible interface, the platform API, at the granularity of complete machines, whereas the microkernel approach enables construction of arbitrary, well-structured systems.

This paper is organized as follows: In Section II, we briefly describe the operation of a virtual machine, followed by a short introduction to the L4 microkernel. In Section III, we discuss how virtual machine concepts map to L4 concepts and identify shortcomings of the current kernel API. Section IV formulates the necessary changes to the L4 API to support virtual machines. We discuss related work in Section V and conclude the paper in Section VI.

II. BACKGROUND

A. Virtual Machines

A Virtual Machine (VM) as defined by Goldberg [2] is an “efficient hardware-software duplicate of a real machine”. The idea of virtual machines is to create the illusion of a physical machine at the lowest level, the platform API. At this level, the communication between the guest (operating system executing inside the VM) and the virtual machine

environment is completely defined by the behavior of the hardware interface.

The virtual machine monitor (VMM) provides the illusion of the VM's environment using the resources of the host system. It has full control of the virtual machine and can establish full isolation or controlled sharing of resources between the VM, itself, and the rest of the system. The environment of the VM includes all resources of a physical machine: the processor, memory, interrupt lines, IO ports and devices. These resources can be provided either as pass-through access to a physical resource or by software emulation.

Pass-through access of a physical resource provides the guest with direct access at the full performance of native operation. If the resource can not be securely multiplexed between virtual machines, it must be assigned exclusively to one VM.

Software emulation of a resource is needed for a physical resource which cannot be securely multiplexed, when strong monitoring of the virtual machine's interaction is wanted, or when no physical instance of the resource is available in the host machine. In the virtual machine, each instruction that accesses an emulated resource has to be prevented from execution [3]. Fully virtualizable architectures have the ability to generate traps on such instructions [4]. The VMM catches the trap, and emulates the effects of the instruction on the virtual machine's state and the software model of the resource.

To emulate active components, for instance, a network device, the VMM has to inject interrupts into the virtual machine. The VMM must respect the virtual machine's state which may require to delay the delivery, e.g., the guest has its interrupts disabled.

A virtual machine, while running on the physical processor, needs to be sheltered from events on the physical machine. For example, physical interrupts must not enter directly the virtual machine. Fully virtualizable architectures support this by strongly separating the virtual machine's context from the physical environment, e.g., by performing a world switch before delivering the event.

B. The L4 Microkernel

The L4 microkernel [5] is a second generation microkernel originally developed by Jochen Liedtke at GMD, IBM, and Karlsruhe University. Various versions exist at Karlsruhe University [6], UNSW/NICTA Sydney, and Dresden University of Technology. The kernel offers two abstractions and two major mechanisms.

Threads are the abstraction of an activity. CPU time is multiplexed between threads bound to the same processor. An L4 thread is represented by its register state (processor registers and virtual registers), a unique global identifier, and an associated address space.

Address spaces provide the abstraction for protection and isolation; resource permissions are bound to an address space. L4 address spaces are no first class object; they are indirectly identified via a thread associated to this particular space. All

threads in an address space have the same rights and can freely manipulate each other.

Inter process communication (IPC) is the mechanism for data transfer and controlled execution transfer between threads. Message transfers are synchronous and involve exactly two threads. Both sender and receiver have to agree on the format of the message.

Mapping is the mechanism for controlled transfer of resource permissions between address spaces. Access to a resource is granted by transferring a *map* or *grant* item identifying a region of the sender's (virtual) address space in an IPC message. Mapping requires mutual agreement of the sender and receiver thread and thus allows save user-level management of address spaces. Map duplicates the resource permissions from the sender's into the receiver's address space; grant moves the permission. The receiver's permissions can only be a subset of the sender's permissions. Mapping can be applied recursively. Revocation of resource rights is done asynchronously through the *unmap* primitive and does not require explicit consent from the receiver of the mapping. L4 implements the address space abstraction with whatever hardware mechanisms available, such as TLBs, page tables, and permission bitmaps. The minimum granularity of mapping operations on address spaces is subject to the hardware's capabilities.

L4 has an in-kernel round-robin *scheduler* that allocates time to threads according to their priority and time-slice length. If the time slice of a thread expires, L4 preempts the thread and schedules the next runnable thread.

Hardware generated events such as *exceptions* and *interrupts* are translated into kernel-generated IPC messages. On a hardware interrupt, the kernel synthesizes a message to a thread that is registered as the handler for that interrupt. The sender appears to be a thread with a special per-interrupt thread identifier. Hardware exceptions are transparent to the faulting thread; the kernel preserves the thread's context. The hardware exceptions are mapped onto an IPC based fault protocol. In the name of the faulting thread, the kernel synthesizes a message with information about the cause of the fault and sends it to the faulting thread's exception handler. The faulting thread is automatically set into a blocking IPC receive operation, waiting for a reply from its exception handler to resume execution. On a page fault exception, the fault message is sent to the pager of the thread, expecting a memory mapping in the reply. The special treatment of the page fault exception has historical reasons.

These protocols allow easy virtualization of physical resources, e.g., paged virtual memory: Under memory pressure, the provider of a page unmaps it from the address space it was mapped to. If a thread now accesses the removed page, a page-fault IPC is sent so that the pager can transparently re-establish the mapping and resume the faulting thread.

III. DESIGN

This section discusses how L4 concepts can be used to provide a virtual machine environment. We first present the general architecture and then discuss selected aspects of VM support in slightly more detail. In fact, L4 already provides the right abstractions and mechanisms to cater for virtual machines; where necessary, we propose minimal extensions or generalizations.

A. Architecture

The architecture is microkernel-based and consists of three major components: the virtual machine, its monitor, and the microkernel.

The *virtual machine* consists of an L4 address space, containing all directly accessible physical resources and one or more L4 threads representing the virtual machine's processors (VCPUs).

The *monitor* defines and maintains the virtual machine's environment. It guarantees isolation by securely controlling the VM's access to physical resources and implements all complex aspects of virtualization, such as emulating instructions accessing resources not directly available to the VM. As a normal L4 thread, the monitor can benefit from the services of other user-level components to build the environment of the virtual machine, for example disk storage or network connectivity.

The *microkernel* provides the execution environment for the VM, the monitor, and the rest of the system. It ensures controlled execution of the guest in the VM, using hardware-supported virtualization techniques where necessary.

B. Resources

To securely isolate a virtual machine, the VM must not have uncontrolled direct access to the physical hardware resources of the host machine. L4's resource mapping mechanism already provides a way to control the permissions of an address space. The monitor simply uses mappings to selectively grant a VM access to a physical resource. In the following, we illustrate this in the context of memory. Some architectures have additional resource spaces, e.g. x86's IO port space. In L4, they are part of the address space abstraction and thus subject to the same mapping mechanism.

Physical Memory: A virtual machine can not have direct access to physical memory as it would circumvent protection. Instead, virtual memory is used [7]. An L4 address space represents the virtual machine's physical memory address space. The monitor populates this space by mapping parts of its own address space into it.

For efficiency reasons, L4 does not offer the complete architecturally defined virtual address space to user level; the kernel keeps part of it for its own purposes. There is, however, no conceptual limitation in L4's mapping mechanism that would prevent managing the whole address space. A guest may require the complete physical address space of the virtual machine, and hardware support for virtualization makes it easy to provide the full address space.

The L4 API defines two mandatory objects in each address space: the kernel interface page and the UTCB area. Their location is determined by the creator of the address space. Being part of the L4 virtual address space, they will appear as objects in the VM's physical address space. The monitor can freely define their position and thereby effectively hide them from the guest. Removing these objects would create special cases for VM address spaces resulting in larger changes to the API, and it would preclude later optimizations.

Virtual Memory: We use L4's virtual memory management to provide the VM's physical address space. However, the guest operating system in the VM may want to use virtual memory itself. To maintain the illusion of direct access to memory, the virtual machine system must resolve a guest-virtual address into a host-physical address [7]. This translation consists of two stages. The first stage translates the guest-virtual address to a guest-physical address via page tables maintained by the guest operating system located in guest-physical memory. The second stage is determined by the monitor's mapping of guest-physical addresses to host-physical addresses. Current hardware lacks support for such a cascaded memory translation, called nested paging. Thus, both stages have to be merged into a single translation, the shadow page table, which directly translates a guest-virtual address into a host-physical address. The shadow page table can also be seen as a virtual TLB (vTLB) managed in software and located between the guest's page table and the hardware TLB.

One way to establish the guest-virtual to host-physical translation is to represent the VM's virtual address space as an L4 address space, containing the contents defined by the shadow page table. VM-internal translation faults are propagated to the monitor which then walks the guest operating system's page-table to find the guest-physical address. It then maps pages from its own address space directly into the virtual address space of the virtual machine. However, this approach has several drawbacks:

- L4's mapping mechanism abstracts from the underlying hardware page table. To allow user-level management of address spaces, it includes access rights such as read, write, and execute, but does not expose the distinction between user- and kernel-accessible memory. Extending L4's mapping mechanism accordingly would require to disable this feature for all but VM-address spaces.
- L4 threads are associated with exactly one L4 address space. As a result only the currently active guest virtual address space can be described by the L4 address space. An address space switch in the VM requires a complete flush and repopulation of the vTLB via mapping by the monitor. Since updates to the virtual TLB are very frequent operations, efficiency of the vTLB is paramount to the virtual machine's overall performance. We consider the cost of two address space switches and a map operation for every vTLB update too expensive.

These problems can be avoided by emulating guest-virtual address spaces transparently inside the kernel. The VM's physical address space is represented by the L4 address space

which is maintained by the monitor. Only faults caused by non-present guest-physical memory are propagated to the monitor. Of course, this approach has some disadvantages, too:

- The in-kernel shadow page-table management can perform only very limited optimizations. Without introducing awkward configuration protocols, optimizations based on the knowledge of a specific guest’s behavior are not possible.
- The complexity of shadow page-table management is rather high: The vTLB algorithm needs to walk the VM’s guest physical memory, which may cause in-kernel page faults (that can be handled like faults during an IPC though.) Furthermore, L4 uses one page table format while the guest operating system may use one of many, increasing complexity of the page table walker for the guest page table.

However, we expect upcoming hardware to natively support nested paging which will remove the need for shadow page tables altogether. Therefore, we prefer in-kernel vTLB management as a temporary, clean solution at the API level: If hardware support is present, L4 simply uses it without any further changes to the API.

C. Processor

The processor of a virtual machine is represented by an L4 thread with its associated state (identifier, priority, scheduler, pager, exception handler) extended by the complete architecturally defined processor state. The VCPU behaves like a native L4 thread; especially, the VCPU is scheduled like all other threads in L4.

To emulate the virtual machine’s resources such as privileged registers, the monitor needs to emulate the instructions accessing these resources. Fully virtualizable hardware allows to generate traps on these instructions, which cause an exit out of the virtual machine into the privileged part of the virtual machine monitor (here the L4 microkernel). L4 already provides an abstraction for these hardware events: the exception protocol. L4 synthesizes a fault message to the associated user-level handler thread in order to report the reason of the fault, including selected user-visible CPU state. The handler resolves the fault, i.e., by emulating the behavior of the faulting instruction, and sends a reply message back to resume the thread. This message contains the updated CPU register state to be established before resuming the thread.

For virtualization, this static protocol is too inflexible because the VCPU’s state is much larger and the relevant state heavily depends on the exact fault reason. Similar to the exception protocol, we propose a *virtualization fault protocol*. For each virtualization fault reason, a pre-defined part of the VCPU state is transferred in the fault message. In the rare case that the monitor requires more or different state than was sent in the fault message, the virtualization fault protocol provides a *no-resume* item. This item contains a request for additional VCPU state; it causes the VCPU to immediately generate another virtualization fault without resuming the guest. Thus

the monitor can iteratively access the complete VCPU register state.

D. Asynchronous Events

To emulate the behavior of active devices, e.g. to inject virtual device interrupts, the monitor has to asynchronously modify the VCPU’s state.

L4 already provides a way to asynchronously manipulate another thread through the EXCHANGEREGISTERS system call. EXCHANGEREGISTERS allows manipulation of the thread’s instruction, stack pointer and flags register, but only from within the same address space. Access to the complete register state has to be emulated by user-level protocols, for example, by inserting a helper thread into the destination address space, reachable via IPC, to do EXCHANGEREGISTERS locally.

Inserting an L4 thread transparently into the virtual machine’s address space is a major intrusion. The thread needs stub code for its protocol logic mapped into the VM’s virtual address space, it needs the ability to invoke IPC, and its presence must not induce any side-effects in the guest.

To avoid this model, the monitor can delay asynchronous events and piggyback them on the next virtualization fault reply. However, this is no general solution, because it may delay asynchronous events for too long. Yet, it is an efficient optimization for high workload situations. As a minimally invasive method to asynchronously access the VCPU state, we favor the extension of EXCHANGEREGISTERS across address space boundaries as already required by the L4Ka Virtual Machine Technology projects [8], yet with one further extension, a possibility to asynchronously force a virtualization event:

- An *immediate fault* causes the VCPU to immediately raise a virtualization fault. The monitor can use this to unconditionally inject events such as non-maskable interrupts or exceptions, or to inspect the VM’s state, e.g., for debugging purposes.
- A *delayed fault* causes the VCPU to raise a virtualization fault on a certain event, for example, the next time the guest is able to receive interrupts. The monitor can then inject pending virtual interrupts.

Allowing a thread’s pager and exception handler to invoke EXCHANGEREGISTERS does not introduce any (additional) security issues, as a pager is already trusted strongly.

Apart from the VCPU register state, the monitor may need to access the VM’s memory. Accessing guest physical memory is not problematic for the monitor since it provided the memory from its own address space or knows the providing component.

E. Memory Mapped Devices

Memory mapped-devices are located in the guest physical address space. They are accessed by normal load/store operations which cannot be trapped even by fully virtualizable hardware. Instead, access to memory-mapped devices can be tracked by page-faults that should not be satisfied with a mapping but trigger an emulation of the accessing instruction.

Therefore, page faults should also use the virtualization fault protocol. Unifying page faults and exception handling is already under discussion in the context of the L4Ka Virtual Machine Technology [8] projects for other reasons such as orthogonality.

IV. CHANGES TO THE L4 API

This section describes the changes to the L4 API that enable the design presented in the previous section. The goal is to allow for user-level management of virtual machines with minimal extensions to the L4 API.

The changes fit harmonically into L4's interface; the resulting API is fully backward compatible. No changes to the thread as the abstraction for a multiplexed CPU were necessary. The address space still is the abstraction for isolation and contains the accessible resources. All virtualization-related hardware events are abstracted as kernel-synthesized IPC messages. To handle a virtual machine no further fault handlers had to be added.

A. SPACECONTROL

The API now differentiates between three modes of an address space. Backwards compatibility is achieved by using two formerly should-be-zero bits to select the mode.

- **Native Mode** This is the normal address space for native L4 threads. No virtualization is used. All L4 services are available.
- **User-Level Virtualization Mode** This mode supports user-level virtualization such as para-virtualization and pre-virtualization. This type of space is not discussed here.
- **Full Virtualization Mode** In full virtualization mode, the physical processor supports virtualization in hardware, the mode addressed in this paper. The address space holds all physical resources of the virtual machine. Initially it is empty, only KIP and UTCB are mapped as parts of the physical address space.

B. EXCHANGEREGISTERS

EXCHANGEREGISTERS uses two additional flags to raise an asynchronous virtualization fault: One flag, when set, forces the VCPU to raise the fault. A second flag, when set, delays generation of the fault until the VCPU enters a state where interrupts can be accepted. The EXCHANGEREGISTERS system call returns, even though generation of the fault may be delayed.

This extension relies on the experimental feature that allows a thread to invoke EXCHANGEREGISTERS on all threads for which it is registered as the pager, even in a different address space.

C. Virtualization Fault Protocol

The virtualization fault protocol unifies the page fault and exception protocol for VCPUs. As such, it is based on IPC. The virtualization fault protocol is defined between the faulting VCPU and the thread registered as its pager. It allows the

pager to get notifications on virtualization-critical events and to access the VCPU register state.

A *virtualization fault message* is synthesized by the kernel when the VCPU raises a virtualization event. The message contains a word specifying the reason followed by a subset of the VCPU register state. The exact set of registers depends on the fault reason and the architecture.

The pager answers the virtualization fault with a *virtualization reply message*. This message can contain the already available typed items for resource mapping, for example, memory mappings. The reply message can also be used to modify or request more VCPU state. Similar to the general IPC protocol, several new items are defined:

- The *set item* changes exactly one register of the VCPU. The target register is encoded in the item, followed by the new value.
- A *group item* sets a predefined, fixed group of VCPU registers.
- The *set-multiple item* sets an arbitrary subset of VCPU registers. The target registers are identified by a dense encoding, such as a bit-field, followed by the values.
- The *no-resume item* requests additional VCPU register state; it uses the encoding of the set multiple item to identify registers to be sent. This item, if present, must be the last item in the reply message. It forces the VCPU to immediately send the requested state in another virtualization fault message without resuming the VCPU.

D. Thread-Startup-Protocol

The startup message of a VCPU thread requires a virtualization reply. This message initializes the VCPU register state and activates the VCPU.

V. RELATED WORK

Related work can be found in three areas: virtual machine monitors, microkernels, and integrations of both.

L4Linux [9], a para-virtualized Linux running on the L4 microkernel, was created to evaluate the microkernel's performance. Since then, L4Linux served various projects targeting efficient, secure subsystems using virtualization [10], [11].

LeVasseur et. al introduced pre-virtualization [12], an automated para-virtualization technique, to reduce the effort of porting new guests to their virtualization environment. The guest operating system is compiled with a pre-virtualization compiler that locates virtualization-sensitive instructions and inserts pad bytes for runtime instruction rewriting. The resulting binary can execute on raw hardware as well as in a pre-virtualization environment.

In contrast, binary translation, as used, for example, by VMware to fully virtualize the x86 architecture, transparently traps and patches virtualization-sensitive guest instructions. It achieves good performance but introduces high complexity into the privileged virtual machine monitor [13]. Recent VMware products can also use hardware virtualization extensions where available.

The Xen [14] system is a para-virtualizing VMM for the x86 architecture. Xen's virtual machines are managed by the Xen hypervisor, but all IO is performed on their behalf by a privileged VM. Currently, only a modified Linux operating system is supported as a guest for such a privileged VM. With Xen 3.0, the hypervisor includes mechanisms for memory sharing and efficient communication between virtual machines to allow running each device driver in its own privileged VM in order to achieve stronger isolation [15]. In recent releases, Xen also added support for full virtualization using the x86 hardware virtualization extensions.

The KVM project [16] is another VMM for full virtualization. It adds a kernel driver to the Linux operating system and exports the hardware virtualization features through a special device file. Each virtual machine is represented as a Linux process. A second, associated process maintains the platform environment for the virtual machine. With the driver being a part of the Linux kernel, the VM's trusted computing base includes the whole Linux kernel.

Fluke [17] is a software-based virtualizable architecture. It combines the concepts of microkernels and virtual machines to increase the operating system's extensibility. Operating system functionality is decomposed vertically into layers, called nesters, that allow the environment provided to the application to be stepwise refined. The application's execution environment consists of the hierarchy of nesters the application runs on and thus only includes the operating system services required.

The close relationship between microkernels and virtual machine monitors has been discussed by Hand et. al [18] and Heiser et. al [19]. The authors argue, that microkernels and virtual machines, although their definitions are quite different, follow similar goals. The work presented in [20] reasons towards microkernel-based virtual machine systems for high performance computing.

The work of Hohmuth et. al [21] explores the design space of hybrid virtual machine/microkernel systems with the goal to minimize the trusted computing base (TCB) required to implement the virtual machine environment. It is found that a VMM extended by mechanisms for memory sharing and communication can reach a very small TCB.

Our work implements the same point in that design space, yet we reach it by including support for hardware-assisted full virtualization into a microkernel. The microkernel design allows for fine-grained, component-based systems with a small trusted computing base whereas support for full virtualization enables reuse of legacy (operating) systems only known from purely virtual machine systems.

VI. CONCLUSION & FUTURE WORK

In this paper we presented a minimalistic set of extensions to the API of the L4 microkernel. They enable user-level management of virtual machines based on L4 abstractions and mechanisms and are fully transparent to threads which do not require these virtualization features. This is achieved by defining minimal, backwards-compatible extensions. This

work shows that a microkernel and the privileged part of a virtual machine monitor can be integrated, providing the advantages of both worlds in one system without affecting the microkernel's overall performance. Especially, there are no changes to the carefully crafted IPC path.

Currently, a prototype implementation of the proposed API is under development. The target architecture is IA-32 using Intel's virtualization extensions (VT-x) [22].

Since this paper is an API proposal, we encourage any type of constructive feedback.

REFERENCES

- [1] G. Heiser, "Secure embedded systems need microkernels," *The USENIX Magazine*, vol. 30, no. 6, pp. 9–13, Dec. 2005.
- [2] R. P. Goldberg, "Survey of virtual machine research," *IEEE Computer Magazine*, vol. 7, no. 6, June 1974.
- [3] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, June 2001, pp. 1–14.
- [4] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, July 1974.
- [5] J. Liedtke, "Toward real microkernels," *Communications of the ACM*, vol. 39, no. 9, pp. 70–77, Sept. 1996.
- [6] University of Karlsruhe, System Architecture Group, "L4 experimental kernel reference manual," Feb. 2006.
- [7] C. A. Waldspurger, "Memory resource management in VMware ESX server," in *Proceedings of the 5th Symposium on Operating System Design and Implementation*. Boston, MA, USA: USENIX Association, Dec. 2002, pp. 181–194.
- [8] The L4Ka Team, "The l4ka virtual machine technology." <http://www.l4ka.org/projects/virtualization/>
- [9] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter, "The performance of microkernel-based systems," in *Proceedings of the 16th Symposium on Operating System Principles (SOSP)*, St. Malo, France. ACM Press, Oct. 5–8 1997. [Online]. Available: <http://l4ka.org/publications/>
- [10] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter, "The Nizza secure-system architecture," in *Proceedings of the The First International Conference on Collaborative Computing: Networking, Applications and Worksharing, December 19–21, 2005, San Jose, CA, USA*. San Jose, CA, USA: IEEE Press, Dec. 2005.
- [11] C. Helmuth, A. Warg, and N. Freske, "Micro-sina – hands-on experiences with the Nizza security architecture," in *Proceedings of the D.A.CH Security 2005, Darmstadt, Germany*, Mar. 2005.
- [12] J. LeVasseur, V. Uhlig, M. Chapman, P. Chubb, B. Leslie, and G. Heiser, "Pre-virtualization: Slashing the cost of virtualization," Fakultät für Informatik, Universität Karlsruhe (TH), Tech. Rep. 2005-30, Nov. 2005.
- [13] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLoS)*, San Jose, CA, USA, Oct. 21–25 2006.
- [14] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, "Xen and the art of virtualization," in *Proceedings of the 19th Symposium on Operating System Principles*. Bolton Landing, New York, USA: ACM Press, Oct. 2003, pp. 164–177.
- [15] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallik, "Xen 3.0 and the art of virtualization," in *Proceedings of the Linux Symposium, Ottawa, Ontario, Canada*, July 20–23 2005, pp. 65–77.
- [16] Qumranet, "KVM: Kernel-based virtual machine for linux." <http://kvm.sourceforge.net/>
- [17] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson, "Microkernels meet recursive virtual machines," in *Proceedings of the 2nd Symposium on Operating System Design and Implementation (OSDI)*, Seattle, Washington, USA. USENIX Association, Oct. 1996, pp. 137–151.

- [18] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. Magenheimer, "Are virtual machine monitors microkernels done right?" in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS X)*, Santa Fe, NM, USA. USENIX Association, June 2005.
- [19] G. Heiser, V. Uhlig, and J. LeVasseur, "Are virtual-machine monitors microkernels done right?" National ITC Australia and University of New South Wales, Tech. Rep. PA0005103, Oct. 2005.
- [20] M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis, "Virtualization for high-performance computing," *ACM Sigops Operating System Review*, vol. 40, no. 2, pp. 8–11, Apr. 2006.
- [21] M. Hohmuth, M. Peter, H. Hartig, and J. S. Shapiro, "Reducing tcb size by using untrusted components – small kernels versus virtual-machine monitors," in *Proceedings of the 11th ACM SIGOPS European Workshop*. Leuven, Belgium: ACM Press, Aug. 2004.
- [22] Intel Corporation, *Intel IA-32 Architecture Software Developer's Manual: Volume 3B: System Programming Guide, Part 2*, Santa Clara, CA, USA, Jan. 2006, order number 253669.

A declarative approach to extensible interface compilation

Nicholas FitzRoy-Dale `nfd@cse.unsw.edu.au`

Abstract— In microkernel-based operating systems, source-to-source compilers generate code to ease the process of marshaling data for communication via message passing. However, the rule of thumb for these *interface compilers* seems to be “simple, extensible, efficient output – pick any one”. I argue that the major cause of extensibility-limiting complexity in interface compilers comes from the source-to-source transformation code itself, and this complexity is primarily a result of the difficulties inherent in supporting multiple targets. I describe a specification-based approach for generating interface compilers, discuss the advantages of such an approach over a procedural approach, and outline a proposed implementation, with particular reference to the advantages of a specification-based approach to interface compilation in terms of flexibility and extensibility.

I. INTRODUCTION

With any systems programming task come situations involving code that could easily be machine generated. A particularly good example of this in microkernel-based systems is in the code required for communication of messages across an interface boundary. Such code is not strictly boilerplate, because it varies with the data layout of the message being sent, but the interface-specific differences can easily be described programmatically. Using a code generator is not necessarily the right choice: adding another layer of abstraction to the communication process makes debugging more difficult and encourages new programmers to ignore the intricacies of the underlying interface (the so-called *leaky abstraction* problem[1]). However, code generators make up for these shortcomings in many ways. For example, several classes of bugs are not possible in generated code, and all code making use of the same interface will communicate across it in the same way.

Automating the process requires a tool capable of producing code from a specification, usually an *interface compiler*. This specialised tool parses an interface specification, usually in a form of Interface Definition Language (IDL), and produces output in the language used to implement the rest of the system (the *target language*), typically in the form of *stub* functions. Code on the caller side of the interface uses these stubs like any other function: the work of packing function parameters into an appropriate location in memory (*marshaling*), communicating using an appropriate operating system primitive, and extracting the results from memory and returning them to the calling function (*unmarshaling*) is performed by the stub. Interface compilers produce appropriate *mirror* stub functions for the service side of the interface, and may also produce skeletal code for a server loop implementing the interface.

Interface compilers are used in many areas of comput-

ing. Servers and distributed systems may make use of a *component system* of which stub compilation is only a very small part, such as an implementation of CORBA[2], COM[3], or JavaBeans[4]. All these component systems support distribution across a network in addition to other “enterprise” features such as load balancing and quality of service. In microkernel-based systems, by contrast, space and efficiency constraints tend to result in the need for safe cross-domain communication without the overheads of a traditional component system. In this situation, most of the work is performed in stubs produced by an interface compiler.

Interface compilers for microkernel-based systems are the focus of this paper. In addition to missing many of the features described above, typical interface compilers in this category differ from their enterprise equivalents in two major ways: they generally produce simpler code, because the microkernel performs the tasks of message-passing and queuing and thus acts as a (partial) object request broker; and they are more likely to be required to produce multiple types of output, even for the same microkernel, because the primary concern of microkernel-focused interface compilers is to produce fast, as opposed to feature-complete, code.

Historically, these interface compilers have been heavily tied to their attendant microkernel. For example, the Mach Interface Generator (MIG)[5] accepts a Mach-specific interface specification, produces highly Mach-specific code, and could not reasonably be used with another kernel. More recently, interface compilers capable of accepting various IDLs and supporting multiple target languages have emerged. These include the Flexible IDL Compiler Kit (Flick)[6], which supports IDLs such as CORBA IDL and ONC RPC at the frontend, and targets Mach[7] and various incarnations of L4[8] at the backend. Flick achieves this modularity through a series of programmer-visible intermediate languages (five in total) which can be operated on independently.

Changes in the design of an interface compiler are typically made either to improve performance or improve generality. However, interface compilers for microkernels remain remarkably difficult to adapt to changing interface requirements. There are three major reasons for the difficulty: firstly, they tend to remain tightly-coupled to a small selection of kernels; secondly, the core interface-generation routines tend to be difficult to modify; and, finally, representation of target code is primitive compared with that offered by source-to-binary compilers.

In the rest of this paper I describe extensibility issues and their various solutions in current interface compilers. I then propose an alternative interface compiler design, mak-

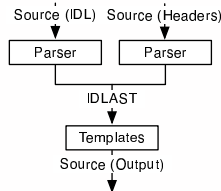


Fig. 1. Stages in the Magpie interface compiler

ing use of specification-directed transformations, and discuss the advantages and disadvantages of this approach. Although I refer to several popular interface compilers, the focus is on adding a declarative layer to Magpie, an interface compiler used with NICTA L4.[9]

II. INTERFACE COMPILERS

Like source-to-binary compilers, interface compilers are all composed of at least a *frontend*, containing the parser for the specification language, and a *backend*, containing the code generator for the target system. It is common to make use of at least one intermediate representation. Figure 1 shows the major stages of the Magpie interface compiler.

Source-to-binary compilers may support multiple platforms, but the targets (platform-specific assembly languages) are all very similar; intermediate stages therefore output some variety of three-address code[10], which is by design quite homogeneous. The intermediate stage of an interface compiler, however, must target some higher-level language, such as C, working with a kernel or library API. Unlike processor assembly languages, kernel or library APIs are very dissimilar, even when designed for the same task (i.e., communication between threads). Designing an effective intermediate stage for an interface compiler is thus simply a harder task, because of higher degree of cross-platform variation. A common response in interface compiler design has been to blur the separation between intermediate stages and final code generation, resulting in intermediate stages which are complicated, inflexible, or both.

Further complicating the task of the interface compiler are the frequently-changing requirements of applications and the kernel, a particularly severe problem for research systems. For example, NICTA L4 has undergone several internal API changes, each one necessitating an equivalent change to the interface compiler. The difficulty of implementing these changes in the popular IDL4 interface compiler has led to the development of Magpie, an interface compiler based on a flexible template system. The partial success of Magpie in this regard is the prime motivation for the work described in this paper.

A. Code transformation

The real work of an interface compiler is performed by the intermediate stage known as the *generator*, which analyses the internal representation of the interface definition

and produces output (either in the target language, or in an intermediate format). In modern interface compilers, the generator forms a third stage between the parser and code output stages, allowing for some degree of flexibility with regard to input and output formats. Nonetheless, making sizeable changes to the format of the output requires modifying the generator, and here choice of language and generator design become important: in many common interface compilers for microkernels, such as Flick, IDL4, and DICE, making even simple changes requires changing a nontrivial amount of code.

B. Code output

Code output is widely regarded to be an easy problem and is thus given little attention during interface compiler development. Indeed, it is easy (in the sense that little design is required) to create a simple interface generator that generates appropriate code for a fixed interface, tied to a single microkernel. However, there are several problems with this approach that both limit the scope of interface compilers and increase their maintenance overhead.

Typically, an interface compiler backend is composed of a series of `printf` statements which together produce the entire stub function or module. This approach is simple to implement, and the intent of the code is obvious, but it is difficult to extend and to test. Alternatively, the backend may assemble a syntax tree in the target language, which is then *walked* as a final step to produce source code. This approach at least provides some reassurance that the generated code will be syntactically correct, but the usual method of assembling the tree in a procedural language, calling class constructors, is difficult to follow, and difficult to extend.

Before considering parameterised templates it is important to consider boilerplate code, that is, code which does not change between invocations of the compiler. Generated code typically contains a lot of boilerplate in the form of header comments, error-handling code, helper routines, etc. A typical procedural approach to generating this sort of code is simply to insert it at appropriate points when producing output. This looks ugly for a system that uses `printf`-style output, and practically unreadable (if it occurs in large chunks) in a system that generates a concrete syntax tree on-the-fly. In either case, boilerplate generation is simply noise for a maintainer attempting to understand what the interface compiler does. The situation gets worse if the programmer wishes to extend the system, perhaps with a new output mode. She is then presented with two options: to copy the code, creating the opportunity for unwanted divergence in the future; or to refactor the code to be more fine-grained, making flow less obvious and providing no guarantee that additional changes will not necessitate further refactoring in the future.

The difficulties associated with code generated piecemeal are magnified for sections of parameterised code, i.e., the sections of code in which the generator actually performs work. Using a syntax tree approach, a single line of (relatively-simple) generated code consumes approximately

```

1  addTo(result, new CASTDeclarationStatement(
2      new CASTDeclaration(
3          new CASTTypeSpecifier(
4              new CASTIdentifier("L4_ThreadId_t")),
5          new CASTDeclarator(
6              new CASTIdentifier("_dummy")))
7  );

```

Fig. 2. Variable declaration, IDL4

```

1 L4_MsgTag_t _result;
2 /*-run(templates.get(
3     'client_function_body_pre_ipc_defs'))-*/
4 ...
5 /*-run(templates.get(
6     'client_function_body_pre_ipc'))-*/
7 _result = L4_Call(_service);
8 /*-run(templates.get(
9     'client_function_body_post_ipc'))-*/

```

Fig. 3. An example of the Magpie templating language

six lines of generated AST (abstract syntax tree) in IDL4. Figure 2, extracted from IDL4, creates a new type instance using multiple classes to construct an abstract syntax tree, which is later walked to produce output. The example in the figure is the equivalent of the C code `L4_ThreadId_t _dummy;`.

C. Magpie

The Magpie interface compiler was developed in recognition of these code-generation problems. Magpie uses a simple templating system, interspersing control code and the target language. The design goals were to keep the backend simple to maintain for developers who wished to make use of alternative interfacing techniques but did not wish to become intimately familiar with Magpie’s code base, and to maintain code flow of the target language as much as possible, thus making the generator and backend easy to understand and, in turn, easy to extend. Magpie was not the first interface compiler to make use of templates. Flick supports its own templating system with the apparent design goal of reducing the amount of non-parameterised code present in the existing interface compiler. Because they are relatively new, Flick’s templates are not supported by most backends.

A small example of the Magpie templating language is shown in Figure 3. The templating command language is embedded inside comments within the target language (Python within C, in this case). The code shown executes the `L4.Call()` function, which performs synchronous IPC in L4. Before and after the call, additional templates are executed using the `run()` function (lines 2, 3, and 5). These templates may generate code that performs additional work to support the IPC operation. For example, in NICTA L4, asynchronous IPC notifications must be disabled on the client side prior to a synchronous IPC call. Code generated by the `client_function_body_*` functions

```

int notifymask;

notifymask = set_notifymask(0);
L4_Call(...);
set_notifymask(notifymask);

```

Fig. 4. Code to manage async IPC in NICTA L4

thus saves the old state of asynchronous IPC notification, disables asynchronous IPC notification, and re-enables it after `L4.Call()` completes. The C code for this functionality is show in Figure 4.

The `run()` command is a convenient extensibility mechanism, but convolutes the flow of the template, contrary to the stated design goals (specifically, understandability of the backend). In the example given above, four separate source files are required to properly disable and re-enable asynchronous IPC.

Magpie’s templating approach is at best only partially successful. Code flow is preserved (the above example notwithstanding), but the templated code is difficult to read, and does not get significantly easier as one becomes more familiar with the system. The templates are not stand-alone: to understand the system one must also be familiar with the procedural-code generator with which the templates communicate and, in some cases, the abstract syntax tree containing the parsed data. Perhaps the best proof of both the success and failure of Magpie is the fact that after more than a year of use, many different templates were successfully developed to accommodate changing requirements, but the sole developer of new templates was the implementor of Magpie.

III. A SPECIFICATION-BASED APPROACH

The interface compilers discussed in the previous sections all use a procedural approach to perform code generation. That is, a program written in a procedural language (commonly C or C++) examines an AST created by the frontend and generates code in the target language, if the compiler does not include a separate generator, or an intermediate representation, if it does. This approach manages to be both too general-purpose and too inflexible: a programmer wishing to modify the behaviour of the stub code must become very familiar with interface-compiler internals, and must modify a nontrivial amount of code to extend the system.

The use of a procedural language to write the generator section of the interface compiler is an obvious choice, because the rest of the interface compiler is typically implemented in a procedural language. However, a procedural approach results in complicated, verbose generator code. The core of an interface compiler is essentially a tree-to-tree transformation. In this section, I demonstrate that using a declarative language confers several benefits over the standard approach, particularly in the areas of code size, understandability, and extensibility. The design comprises a set of modifications and extensions to the Magpie interface compiler, resulting in a new interface compiler

Magpie	Currawong
Parser	Parser
AST gen	AST gen
Generator	Declarations
Templates	Tree walk

Fig. 5. Magpie and Currawong

named Currawong.

A. Currawong and Magpie

Currawong replaces most of Magpie’s generator and templating sections, as shown in Figure 5. Magpie’s frontend parser and generators are preserved. The intermediate and backend layers are combined to form a declarative processing layer. The output from this layer is essentially an AST, so a final small stage walks the tree to produce output.

B. Procedures and declarations

As discussed above, interface compilers are complicated at least partly because of the strong relationship between the target language and compilers for that language – such as C and the gcc compiler. In short, the specification of any language mapping must necessarily conflate interface-level details, such as the layout of data in memory, and language-level details, such as the appropriate casts, bitwise shifting, and logical operations required to create this layout. This problem cannot be avoided completely – at some point, we must speak the target language – but its impact on generator extensibility can be minimised by separating the levels of specification. Referring to the previous example: if a given platform requires that parameters be marshaled into a certain location in memory, it is plausible that one may desire to change the marshal location of a given 64-bit integer in an interface-defined function, but less likely that one would wish to change the fact that, in C, the low-order 32 bits of this word may be accessed using “`theword & 0xFFFFFFFF`”.

A common response to the specification requirement has been to perform exactly the separation described above. The Flick interface compiler, for example, uses five separate abstract representations for language-level and interface-level requirements. However, these five abstract representations have been determined by the implementors of Flick. They can, therefore, not be easily modified to accommodate level-spanning problems, and are thus inflexible: to again refer to the parameter-passing example above, one may make arbitrary changes at the level of determining the names and types of parameters to be marshaled, but the layout and organisation of parameters is only accessible at the lowest level, CAST, which essentially encodes a C++ abstract syntax tree. Other interface compilers tackle the same problem using a highly-stratified generator and rely on the language features of their interface compiler’s language to supply extension. For example, the IDL4 and DICE compilers, written in C++, stratify their generators

```
typedef unsigned int counter_t;

type counter_t (1, example.h:1)
  (meta_type) = ['alias']
  (indirection) = ['']
  target (2, ?:?)
  type unsigned int (backref)
```

Fig. 6. A simple C typedef and its AST representation

```
type(alias(counter_t, name(unsigned int)))
```

Fig. 7. A simple C typedef, Prolog syntax

into the generator proper and a smaller section that separates messages into *channels*, allowing message order to be changed through interface inheritance.

In the Currawong system, the generator applies a series of transformations to a *code template* written in the target language in order to transform the template into an appropriate stub function. The “meat” of the transformation system is specified declaratively in Prolog and, due to the template, only the transformations necessary to customise the template to a particular interface are specified. These two characteristics combine to make the customisable aspects of the transformation system very compact and understandable.

C. Type management

In order to compile an interface definition, it must first be parsed and converted to an easy-to-manipulate form. In the Magpie interface compiler, the primary data structure is an abstract syntax tree produced by the frontend. An example of this structure is shown in Figure 6, with the C code that it represents. Magpie can make use of any type declaration presented in its frontend and, therefore, types defined in the interface definition file, and types defined in any included C files, are interchangeable.

Representing Magpie’s AST in Currawong is a simple matter of transforming the AST, producing idiomatic Prolog nested structures. An example is shown in Figure 7. Some simplification may take place: AST nodes representing alias types (such as those defined by `typedef` in C) include a direct reference to the target type of the alias (strictly speaking, the “AST” is actually a directed acyclic graph) but the converted representation does not. Node *attributes* (`meta_type` and `indirection` in this case) are discarded or converted, and line number information is not preserved. Nonetheless, this form of specification is very powerful, because it allows one to use Prolog’s unification facility in the expected way to perform pattern-matching. For example, all C `typedef` types may be matched using the Prolog term `type(alias(Name, Type))`.

D. Specification language

The Currawong specification language is a declarative representation of the generator present in other interface compilers, such as Flick, IDL4, and Magpie. A generator

```

parameters(P) :: list_head(P,
    parameter('L4_Word_t', '_service').

list_head([H|T], H).
make_list_head(List, H, Result)
    :- Result = [H|List].

```

Fig. 8. An example of Currawong rule language

in the Currawong syntax is a series of declarations, each one representing a single code transformation.

In general, a code transformation may be described as consisting of three parts: a *match rule* that locates the desired portion of code, a set of *requirements* which serves to determine whether the code is correct with regards to the transformation, and a *transformation rule* which may be followed to transform the matched code and make it satisfy the requirements. In Currawong, these three parts are separated using a minor extension to Prolog syntax, as per the example shown in Figure 8: a syntax marker consisting of two colons separates the match rule from the requirements. In other regards, the syntax is Prolog. In the current experimental prototype, Currawong includes its own interpreter for this language, which is essentially a custom Prolog implementation with the above extension. A future, more complete, version of Currawong will support one or more open-source Prolog implementations, such as SWI-Prolog or GNU Prolog, instead.

Matching of rules and requirements is performed using the Prolog unification mechanism. A transformation is considered complete when, for each item matched by a match rule, the corresponding set of requirements also matches. Referring to the example in Figure 8, any set of parameters is matched, and the parameter list is bound to the Prolog variable `P`. Then `list_head` is called, which ensures that the first element of `P` is the parameter `L4_Word_t _service`.

If a match rule is encountered for which the corresponding requirements do not match, *code transformation* is performed. Code transformation proceeds as follows:

1. Take the first requirement from the list of requirements.
2. Find the corresponding transformation rule, which is the rule whose functor starts with `match_` and ends with the functor of the requirement under consideration. The corresponding transformation rule must have an arity one greater than that of the requirement under consideration – the final parameter is taken to be the transformed rule, and all other parameters correspond.
3. Unify any free variables in the transformation rule. Replace the matched portion with the result generated by the transformation rule.
4. Remove this requirement from the list.
5. Repeat this procedure until there are no more requirements.

Referring again to 8, if the parameter list does not contain `L4_Word_t _service` as a first parameter, the list is mutated by a call to `make_list_head`, and the AST is updated.

Note that the match rule and requirements reside in the

```

context(expression(call(L4_Call, _)),
    Before, After) ::

list_contains(Before, expression(
    equals(Var, call(set_notifymask(0))))),
list_contains(After, expression(
    call(set_notifymask(Var))))).

```

Fig. 9. A currawong declaration: managing asynchronous IPC

same clause, on either side of the double-colon separator – collectively named the match/requirements clause. However, the transformation rule is a separate clause. This is because it is appropriate that the transformation rule, being far more general-purpose than the match/requirements clause, reside in a separate system-wide library of manipulation rules.

Given that Currawong parses a template and applies its transformations to the template, there is a great potential for reduction of code size via elimination of boilerplate code. In effect, Currawong statements act as a sort of aspect language for aspect-oriented programming (AOP)[11], specifying transformations through matching. Prolog is an ideal choice for an aspect language, because it is well-suited to the types of tasks, specifically searching, that form a large part of any aspect language.

IV. EXAMPLES

This section describes some examples for which a declarative code-transformation approach is well-suited.

A. L4 notification mask

Many superficial changes to non-parameterised code can be made simply by changing the code itself. For example, asynchronous notification was recently added to NICTA L4. It is undesirable for asynchronous notifications to arrive during a synchronous IPC, so they should be disabled prior to the IPC and re-enabled subsequently, as discussed in Section II-C. An example Currawong implementation is shown in Figure 9. Although this declaration effectively generates code to manage asynchronous IPC, it is essentially a correctness check, and can be used as such.

It is worthwhile to compare the declarative specification with the implementation of the same functionality in Magpie (Figure 3). In Magpie the code is distributed over several templates, including “hooks” (the `run()` command) in the base template, but the Currawong implementation has the form of an aspect in the AOP sense: both the base code and the aspect are well-separated, and significant changes may be made to the base code without any knowledge of the asynchronous support. In fact, the sorts of changes that would require knowledge of the asynchronous IPC support are those that would require rewriting the support anyway, such as a decision not to use `L4_Call()` in the base code.

B. Interface-specific optimisations

The previous examples demonstrated flexible creation of generic interface compilers. However, not all interfaces are

```

1 function(iguana_pd_pypd, _, Body) ::
2     % Ensure that we declare the static.
3     list_contains(Body, expression(typeinst
4         ('static uintptr_t', 'mypd', 0))),
5
6     % Return early if the static is
7     % initialised.
8     context(expression(call(L4_Call),
9         Before, After),
10    list_contains(Before, if(expression(
11        notequals(mypd, 0),
12        expression(return(mypd))))),
13    % Initialise the static after the call.
14    list_contains(After, expression(
15        equals(mypd, __retval))

```

Fig. 10. An interface-specific optimisation rule

the same, and some may benefit from interface-specific customisations – optimisations in particular.

A simple example occurs when caching. The Iguana L4 OS personality separates threads into *protection domains*, where two threads in disjoint protection domains may not share data. A thread’s protection domain identifier is represented by a nonzero integer which threads may obtain using the `iguana_pd_mypd()` function. This integer does not change over the thread’s lifetime. It might make sense, then, to include a declaration to perform caching on the `iguana_pd_mypd` interface function, which returns a thread’s protection domain identifier. Such an optimisation rule is shown in Figure 10. This rule declares a static variable to cache the result of an IPC call (lines 3 and 4), checks to see if it can return the static before making the call (lines 8 to 12), and sets the static to the result of the call afterwards (lines 14 and 15). The Prolog representation of the C code is somewhat verbose – a possible solution to this aesthetic problem is discussed in Section V.

V. RELATED WORK

The description of Currawong above includes an example of interface-specific optimisation in C. Approaches for non-microkernel-based systems have focused less on per-interface optimisation and more on per-language optimisation. The Concert Signature Representation[12] does not make use of an explicit interface definition language, but allows definition of an interface using the target language’s own definition sub-language (for example, function prototypes in C) with extensions. It then relies on a declarative mini-language to specify the type and location of each parameter, in a similar fashion to Currawong’s declarative language. The experimental interface generator Mockingbird[13] arrives at a similar solution in a different way. Recognising that the ideal presentation of an interface in C++ is very different to an ideal presentation of the same interface in Java, Mockingbird allows programmers to hand-define the ideal interface in both languages, and ensures correctness by compiling both interfaces to its own unambiguous internal interface definition language and en-

suring that the internal representations for both interfaces are the same.

Microkernel-focused systems concentrate more on per-interface customisation. Ford et al. [14] discuss the importance of separating interface *presentation* (the programmer-facing side of the interface; essentially the stub code signature) from the interface *contract* (the data that must be transferred) and describe a small specification language to customise the presentation for an interface generator for the Mach microkernel.

Currawong is not the first design to make use of declarations to perform program modification, known as *logic metaprogramming*. The TyRuBa system[15] is a tool for aspect-oriented programming in Java (a *weaver*) in which aspects are specified in a variant of Prolog. The authors claim that this approach allows easy extensions of the aspect language. An interesting feature of TyRuBa not currently incorporated into the Currawong design is the ability to embed portions of the target language directly into aspect declarations. Judicious use of such a feature could simplify blocks of code with few external dependencies, such as the optimisation example given above. If the target code were parsed by a Currawong implementation, it could also simplify examples such as the first one, allowing all rules to be written in plain C.

VI. LIMITATIONS AND FUTURE WORK

As mentioned above, information is lost when type information is converted from the AST representation supplied by the parser to Prolog-style declarations. This information, which includes line number information, may be useful when reconstructing modified files. This information may be restored in several ways. Perhaps the most obvious is simply to store the information as additional attributes in the data structure representing a Prolog atom or node. The extra information could then be accessed using built-in predicates.

Haerberlen et al.[16] implemented a number of optimisations when designing the L4-specific interface compiler IDL4. While some of these (such as the direct-stack transfer) are architecture-specific, it would be informative to implement these optimisations in the framework of Currawong, to determine which are feasible in that context. Although I suspect the completed Currawong implementation to run relatively slowly in its first incarnation, the runtime speed of the generated stub code should be entirely dependent on the input passed to Currawong – transformations performed inside Currawong should not have an impact on stub performance.

Although most of the sections necessary to implement the Currawong extensions to Magpie are complete, some implementation work is still required to produce a working, testable interface compiler.

VII. CONCLUSION

Interface compilers, particularly interface compilers for microkernel-based systems, have some unusual requirements for which traditional compiler techniques are not

particularly well-suited. The requirement to support a large array of targets, and to do so using a relatively high-level language, creates unique problems and has produced a variety of novel solutions. The use of a declarative language, combined with aspect-oriented techniques, has the potential to reduce the complexity and overall size of an interface compiler generator, and is worthy of further exploration and implementation.

REFERENCES

- [1] Joel Spolsky, "The law of leaky abstractions," <http://www.joelonsoftware.com/printerFriendly/articles/LeakyAbstractions.html>, 2002.
- [2] Object Management Group, "CORBA 3.0.3, Common Object Request Broker Architecture (Core Specification), 2004-03-01," 2004.
- [3] Microsoft Corporation and Digital Equipment Corporation, *The Component Object Model Specification*, 1995.
- [4] Sun Microsystems, "Java beans: A component architecture for Java," 1996.
- [5] Open Software Foundation and Carnegie Mellon University, *Mach 3 Server Writer's Guide*, Jul 1992.
- [6] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom, "Flick: A flexible, optimizing IDL compiler," in *SIGPLAN Conference on Programming Language Design and Implementation*, 1997, pp. 44–56.
- [7] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, "MACH: A new kernel foundation for UNIX development," Tech. Rep., Carnegie Mellon University, Computer Science Dept., Pittsburgh, PA, USA, 1986.
- [8] Jochen Liedtke, "On micro-kernel construction," in *SOSP 1995*, Copper Mountain, CO, USA, Dec. 1995, pp. 237–250.
- [9] Gernot Heiser, "Secure embedded systems need microkernels," in *login: USENIX*, Dec. 1995.
- [10] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: principles, techniques, and tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
- [11] Gregor Kiczales, John Lamping, Anurang Mendhekar, Chris Maeda, Christina Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect-oriented programming," *ACM Comput. Surv.*, vol. 28, no. 4es, pp. 154, 1996.
- [12] Joshua S. Auerbach and James R. Russell, "The Concert Signature Representation: IDL as intermediate language," *ACM SIGPLAN Notices*, vol. 29, no. 8, pp. 1–12, 1994.
- [13] Joshua S. Auerbach, Charles Barton, Mark Chu-Carroll, and Mukund Raghavachari, "Mockingbird: Flexible stub compilation from pairs of declarations," in *International Conference on Distributed Computing Systems*, 1999, pp. 393–402.
- [14] Bryan Ford, Mike Hibler, and Jay Lepreau, "Using annotated interface definitions to optimize RPC," in *Symposium on Operating Systems Principles*, 1995, p. 232.
- [15] Kris De Volder, "Aspect-oriented logic meta programming," in *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, Pierre Cointe, Ed. 1999, vol. 1616 of *Lecture Notes in Computer Science*, pp. 250–272, Springer Verlag.
- [16] Andreas Haeberlen, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig, "Stub-code performance is becoming important," in *Proceedings of the 1st Workshop on Industrial Experiences with Systems Software (WIESS)*, Berkeley, CA, 2000, USENIX Association.

Evolution of the PikeOS Microkernel

Robert Kaiser

SYSGO AG, Klein-Winternheim, and
Distributed Systems Lab,
University of Applied Sciences, Wiesbaden

Stephan Wagner

SYSGO AG, Klein-Winternheim

E-mail: {rob, swa}@sysgo.com

Abstract

The PikeOS microkernel is targeted at real-time embedded systems. Its main goal is to provide a partitioned environment for multiple operating systems with different design goals to coexist in a single machine. It was initially modelled after the L4 microkernel and has gradually evolved over the years of its application to the real-time, embedded systems space. This paper describes the concepts that were added or removed during this evolution and it provides the rationale behind these design decisions.

1 Introduction

Microkernels have been receiving new attention during the recent years. After being discarded in the mid 1990s on the grounds of causing too much performance impact, the approach seems to be the answer to today's computer problems: Today, computers generally do not suffer from lack of performance, but they often have severe reliability problems. This is especially relevant in the field of embedded systems: While the modern PC user may have (grudgingly) come to accept the occasional system crash as a fact of life, crashing cellular phones or video recorders are embarrassing for their manufacturers and a potential cause for loss of reputation and market share. More critically though, a malfunction in an electronic control unit of, e.g. a car or an airplane can be a severe threat to the life of humans.

Software complexity is the core problem here and microkernels offer the possibility to tackle it with a "divide and conquer" approach: a microkernel only provides basic functionality which can be used to divide the system's resources (memory, I/O devices, CPU time) into separate subsets. Each of these subsets, which we will further refer to as *partitions*, can be regarded as a virtual machine¹ and

¹We consider virtual machine monitors such as Xen [2] or VMware ESX Server [22] to be specialised microkernels.

as such it can host an operating system along with its world of application programs (see Figure 1). Since partitions operate on separate sets of resources, they are completely isolated and there is no way for a program in one partition to affect another partition². In this way, multiple "guest" operating systems are able to coexist in a single machine and their individual functionalities can be tailored to match the requirements of their application programs. Thus, applications are no longer forced to unconditionally trust a huge monolithic kernel containing a lot of complex functionalities that the application may or may not need. Instead, each subsystem can choose the amount of code that it wants to trust: It can trade complexity for trust.

In complex embedded systems, there frequently exist applications with very different temporal requirements: some must guarantee timely service under all circumstances while others have no such constraint, but are instead expected to work "as fast as possible" by making good use of all resources they can possibly get. The differences between such real-time and non-real-time programs are reflected in the functionalities they need their underlying operating system to provide: There are distinct real-time and non-real-time operating system functions. This presents a problem in monolithic systems because there exists only one operating system interface which has to incorporate all the real-time and non-real-time functionalities. In contrast, a microkernel can host multiple operating systems, so it is possible to have distinct real-time or non-real-time interfaces coexisting in separate partitions of a single machine. However, in such a scenario, the microkernel must guarantee timely availability of sufficient computational resources to its real-time guests so they can in turn fulfil their requirements. Not all microkernels are suitable in this respect.

Since the late 1990s, Sysgo have been developing their own microkernel. Initially, it was called "P4" and it was a faithful re-implementation of the L4 version 2.0 microkernel API as specified by Jochen Liedtke in [17]. It was

²Unless, of course, both sides explicitly agree on a transaction

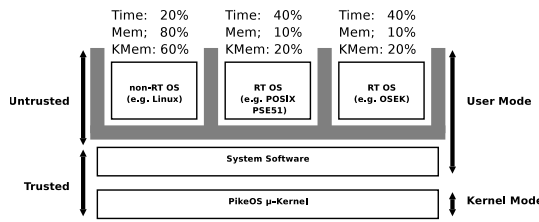


Figure 1. Partitioned system

targeted at embedded systems, therefore, unlike the other then-current L4 implementations, it was written almost entirely in C to facilitate porting, and it was designed to be fully preemptive to support real-time applications.

Over the years of using this kernel in the embedded space, we identified a number of issues with the original L4 version 2.0 interface. These prompted changes of the programming interface and thus, although it is still based on the principles laid out in [16], the PikeOS microkernel's interface today resembles none of the other existing L4 API versions. In the following sections, we will discuss the problems we encountered, and we will outline some of the solutions we chose to apply.

2 Issues

Scheduler Functionality

As already mentioned, there is often a need in embedded systems for programs and operating systems with different temporal requirements to coexist. Real-time programs need to have guaranteed amounts of time at their disposal so they can meet their deadlines. Non-real-time programs need to use all the resources they can get in order to deliver optimal performance. Time allocations for real-time programs must be dimensioned according to worst-case assumptions which are conceivable, but are generally very unlikely to apply. The discrepancy between such a worst case and the average case can easily span multiple orders of magnitude, mainly due to the caches found in every modern computer architecture. Therefore, in the average case, real-time programs usually complete well before their deadlines. In order to put the remaining, excess time to good use, the system should be able to dynamically re-allocate it for use by non-real-time programs.

The classical priority-driven scheduling used by L4 supports this: By assigning sufficiently high priorities to all threads which have temporal requirements, these threads obtain access to the CPU first, and any time not consumed by them is then dynamically made available to the lower-priority, non-real-time parts of the system.

However, there is a general problem with this: Any program with a sufficiently high priority has the ability to block

all other programs with priorities below its own indefinitely. So, conversely, every program with a given priority is forced to trust in the cooperation of all programs that have a priority higher than its own. There is an implicit dependency between the priority level and the level of trust attributed to a program. But these two attributes do not necessarily coincide: A high priority level may need to be granted to a program because it has a requirement regarding timely execution, whereas a high level of trust should only follow from a thorough inspection of the program code. If a high-priority program exists in one partition and a lower-priority one exists in another, the low priority program must still trust the high-priority one, so this defeats the secure isolation between partitions.

A method is needed here to guarantee (and to enforce) *sufficient* time quanta for partitions hosting real-time programs, thus enabling them to on the one hand provide timely service, while on the other hand keeping them from affecting other partitions by consuming more CPU time than they are entitled to.

Memory Requirements and Memory Accountability

Embedded systems are designed for a specific purpose and they are expected to perform their job at the lowest possible cost. Therefore, an embedded system is usually equipped with just enough memory to do what it is made for, and it is difficult to justify a new concept in the embedded market if it implies a need for more memory.

However, experience using a microkernel like L4 shows that it does increase a system's overall memory requirements. One reason for this is related to the need (or tendency) of using threads to implement certain concepts: Interrupt handlers, for example, are implemented as threads. In L⁴Linux, every user process needs *two* L4 threads for its implementation [10]. User-level synchronisation objects (e.g. counting semaphores) are implemented as threads. In result, systems based on L4 tend to employ a rather large number of threads. Each of them requires at least one page of kernel memory for kernel stack and data structures and another page of user memory for the user stack. With a typical load of more than a hundred threads, the resulting memory consumption (8 KB per thread) can be prohibitive for many embedded applications.

Another reason is the kernel's mapping database which is used to store the mapping trees of all of the system's physical pages. It grows and shrinks dynamically as mappings are created or deleted. There is no conceptual limit to the amount of memory that the mapping database might consume. In [18], Liedtke et al describe a problem which has been a topic of ongoing work in the L4 community until today: Malicious (or faulty) programs are able to exhaust

the kernel's memory resources, for example, by requesting a sufficiently large number of mappings to be made. In this way, programs running in one partition can adversely affect programs in other partitions, thereby again defeating the secure isolation between partitions. Furthermore, the mapping database requires a kernel heap allocator³ to deliver memory blocks which are used to store mapping entries. These entries are made on behalf of different user programs. So, while programs can be charged individually for the kernel memory they consume to store page tables, there is no straightforward way to do this also for the amounts of kernel memory that the mapping database consumes on their behalf.

Code Complexity

The PikeOS kernel is targeted for use in safety-critical applications. Thus it must be prepared for a comprehensive validation according to safety standards such as [20]. Since the kernel runs in privileged mode, all of its code contributes to the trusted code base of every application that might run on top of it. Therefore, the amount of kernel code must be kept minimal. L4 implementations have traditionally been very good in this respect. However, even in L4 there is a kernel mapping database which, besides its problematic consumption of kernel memory, is also a complex piece of code, and so is the underlying slab allocator. This makes it hard (read: costly) to validate.

On the other hand, some L4 concepts tend to force unnecessary complexity on user level code. An example for this would be the construction of address spaces by means of IPC: The creator of an address space has to provide a client thread to run in the new address space for the sole purpose of accepting its mappings. Creation of an address space is an operation that all user-level programs (including safety-critical ones) need to do at some point, so it does not help to reduce kernel complexity at the cost of making these operations overly complex at the user level: either way, the complex code will be part of the trusted code base.

Access Privileges

Application programs running under a guest operating system need not (and, according to the principle of least privilege, should not) be able to access the microkernel's system call interface directly. In fact, they should not even be aware of the microkernel's presence. Otherwise, for example, a Linux user process could change its address space layout without the Linux kernel knowing about it. However, the L4 version 2.0 interface does not provide any means to restrict access to its system call interface. Thus, any thread

³The PikeOS kernel originally employed a "slab" allocator ([4]) for this purpose.

is able to consume kernel resources (e.g. by installing mappings), to manipulate system settings or to delay a given schedule, etc.

Furthermore, L4 lacks a flexible but powerful IPC control mechanism to manage the information flow of a complex system easily and effectively. The "clans and chiefs" method introduced in [15] is generally regarded to be a simple, but too inflexible solution.

3 Approaches

The scope of this paper does not allow for an exhaustive discussion of the details of all the changes that were made. Therefore, we will concentrate mainly on the two changes that will probably be considered the most radical ones by the L4 community, namely the addition of partition scheduling and the removal of the mapping database. Subsequently, we will briefly discuss some more modifications, ordered by relevance.

Partition Scheduling

The goal of PikeOS is to provide partitions (or virtual machines) that comprise a subset of the system's resources. Processing time is one of those resources. We expect the partitions to host a variety of guest operating systems with different requirements regarding timely execution. There will typically be real-time as well as non-real-time systems, and the real-time systems will generally fall into one of two categories:

- Time-triggered: Threads are executed according to a static schedule which activates each thread at predetermined points in time and for a predetermined amount of time.
- Event-triggered: Threads are activated in response to external events. Scheduling priorities are used to decide which thread is activated first in case multiple events occur at the same time.

Both approaches have their specific advantages and disadvantages[7]. They are mutually exclusive: Allowing for events to interrupt a time-triggered schedule affects its determinism (it increases jitter). On the other hand, reserving a time slot in the schedule for processing any pending events leads to poor worst case response times and –again– jitter of event-triggered threads. So, whenever the two approaches are combined in a system, one of them has to be given precedence and as a consequence, the other is destined to perform poorly.

We can not expect guest operating systems to trust each other. Therefore, a scheduling technique had to be devised

that on one hand allows real-time and non-real-time systems to coexist efficiently, but that on the other hand avoids the implicit dependency between priority and trust we described earlier. This method had to be efficient and simple in order to not add to the complexity of trusted code. To our knowledge, the scheduling method used in the PikeOS kernel is both unique and new ([13]). It is a superset to the method described in the ARINC 653 standard [1], which is commonly applied in the field of avionics:

Like L4, the PikeOS microkernel uses threads with static priority levels to represent activities. But unlike L4, they are grouped into sets which we refer to as "time partitions", τ_i . The microkernel supports a configurable number of such time partitions. Each of them is represented in the microkernel as an array of linked lists (one list per priority level). Threads that have the same priority level are linked into the same list in a first in/first out manner, and the thread at the head of the list is the first to be executed. When a thread blocks, it is appended to the end of the list. So, within each time partition, there is the same scheduling method that L4 uses, i.e. a classical, priority-driven scheduling with round robin scheduling between threads at the same priority. However, unlike L4, the PikeOS microkernel supports multiple time partitions instead of just one. Threads can only execute while their corresponding time partition is active, regardless of their priority. If we cycled through the time partitions, activating each one at a time for a fixed duration, we would obtain the behaviour of an ARINC 653 scheduler. But in contrast to this, the PikeOS kernel allows *two* of the time partitions to be active at the same time:

- The first time partition, τ_0 plays a special role in that it is always active. It is referred to as the "background" partition.
- Of all other partitions $\tau_i (i \neq 0)$, only one can be active at a time. The microkernel provides a (privileged) system call to select the currently active time partition. It is referred to as the "foreground" partition. Switching happens cyclically, according to a pre-configured, static schedule.

The microkernel scheduler always selects for execution the thread with the highest priority from the set union of τ_0 and τ_i . Figure 2 shows the principle.

The threads have different semantics, depending on their priorities and on their time partitions:

- $\tau_i (i \neq 0)$: The system cyclically activates each of the possible τ_i in turn, giving each of them a configurable portion of the cycle time. So, the totality of all threads in any of these time partitions receives a fixed amount of time at fixed points in time. During their active time slice, the threads compete for the CPU according to

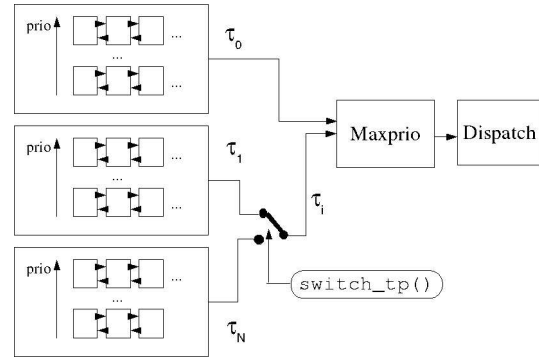


Figure 2. PikeOS partition scheduler: principle of operation.

their priorities. Generally, these threads will be configured to use a mid-level range of priorities (though this is not technically necessary).

- τ_0 : The semantics of the threads assigned to the background partition, τ_0 , depend on their respective priorities.
 - Low-priority threads within τ_0 will receive the processing time that was assigned to, but not used by the mid-priority threads in τ_i . All threads that do not have any real-time requirements are therefore assigned to τ_0 , and they are all given the same, low priority level. Since their priorities are equal, they run under a round robin scheduler, sharing their amount of computation time evenly.
 - Mid- or high-priority threads in τ_0 compete with the currently active foreground partition. If their priorities are higher, they can preempt any low- or mid-priority threads at any time. This is used to implement event-driven real-time threads.

The relation between the priorities of the foreground and the background partition decides whether time-driven threads take precedence over event-driven ones or vice versa: If an event-driven thread's priority is higher, it can preempt time-driven threads at any time, effectively "stealing" its execution time from them. Therefore, the points in time when time-driven threads are activated become undeterministic to some extent (i.e. they "jitter"). The maximum jitter is increased by the cumulative worst case execution times of all high-priority threads (which has to be bounded). It should be noted that in this case, the high-priority threads have to be trusted to not exceed their execution time.

In the opposite case, i.e. the time-triggered threads having a higher priority, the situation is reversed: the event-triggered threads need to trust in the time-triggered threads

to leave over sufficient time for processing events. This can simply be done by leaving a part of the schedule unallocated.

The PikeOS partition scheduler can not completely solve the dilemma between time-triggered and event-triggered real-time systems, but it does offer good flexibility in this respect: The decision whether to give precedence to time-triggered or event-triggered threads can be made individually for every time-triggered partition, by selecting the priorities of its threads to be either above or below those of the event-triggered threads.

The microkernel implements only the mechanism⁴ to select one of the time partitions as active foreground partition. The policy part of deciding which of the time partitions is activated when is left to the user level. This can be done by an interrupt handler thread which runs at a high priority in τ_0 . In a typical configuration, this thread gets activated by an external one-shot timer whenever the timeslice of the currently active foreground partition has expired. It then activates the next time partition, re-programs the one-shot timer to trigger an interrupt when the next partition's time allocation expires and then waits for this interrupt. However, this is only one possible scenario: Since it is implemented at the user level, the partition switching policy can easily be replaced without any changes to the microkernel.

In practice, for a safe coexistence of multiple real-time systems, each of them should have its own time slot, i.e. its threads should be assigned to one of the foreground partitions $\tau_i (i \neq 0)$. In this way, each of them can trust to receive a guaranteed amount of time at cyclically repeated (i.e. fixed) points in time. This is a necessary requirement when applying scheduling analysis to a real-time subsystem (see, for example [3]). The possibility to override this strict time-driven scheduling scheme by high-priority background partition threads is reserved for selected activities which can be trusted to consume only negligible amounts of CPU time⁵. With all real-time systems being confined to their individual time slots, their worst case response time and jitter directly depend on the cycle frequency at which their time slots are repeated (see [12]). It would be desirable to make this frequency as high as possible but this is limited by the cost of switching between time partitions. Therefore, the scheduling algorithm that is executed as part of these switches has to be both fast (i.e. non-complex) and bounded. The PikeOS partition scheduler meets these requirements: since it has to consider only two partitions for each switch, its execution time is constant (i.e. $O(1)$). Practical experience with a PowerPC platform⁶ has shown worst case partition switch times to be in the range of $25\mu s$.

⁴Called `switch_tp()` in Figure 2

⁵The microkernel maintains a "maximum controlled priority" as defined in the L4 version 2.0 API [17] to ensure that untrusted activities can not assume priorities above their limit.

⁶Motorola MPC5200 at 400 MHz.

Therefore, if an application can live –for example– with a context switch overhead of 10%, minimum per partition time allocations can be made as low as $250\mu s$.

Recently, a new scheduling method called "adaptive partitioning" [5] has been announced by QNX software systems. Like the PikeOS scheduler, this approach pursues the goal of combining deterministic time allocation with good processor utilisation. It also represents groups of threads as partitions and uses a combination of priority-driven and time-driven scheduling, but no per-partition time slots are defined, i.e. all partitions can compete for the CPU at all times, based on their thread's priorities. This would suggest an $O(n)$ complexity of the scheduler, however sufficient details about the implementation are not available. The guaranteed time allocations of partitions are specified as relative values (percentages of the total CPU processing capacity), but it is not clear how these could be mapped to time slots, i.e. points in time when a partition receives the CPU and durations for how long it will be able to retain it. The approach uses an "averaging window"⁷ during which a partition is entitled to receive its percentage of CPU capacity, but it also allows special "critical partitions" to exceed their allocations during an averaging window, as long as they later repay the "debt" which they accumulate in this way. In summary, this approach seems to be more complex than the PikeOS partition scheduler, which would imply a more complex trusted code base. It appears to be targeted mainly towards soft real-time applications: while it is able to guarantee a deterministic average distribution of CPU resources to its partitions, the exact time slot during which a particular partition has access to the CPU is not very well defined.

Abandonment of the Mapping Database

The method of creating address spaces recursively on top of other address spaces as described by Liedtke in [16] is an intriguingly elegant mechanism. However, the concept calls for a recursive unmap operation which in turn necessitates a mapping database to keep track of the mapping history of all pages. This mapping database is the source of many problems, most of which have already been introduced (i.e. potentially unlimited kernel memory consumption, difficulty to attribute the memory to individual kernel resource partitions, general code complexity). An additional problem which is relevant especially for real-time systems is the difficulty in estimating the worst case execution time needed for the recursive termination of task trees. The sum of all these problems provided a strong motivation to question the need for a mapping database in general, and an analysis of the practical use cases in the context of PikeOS revealed that recursive unmap operations normally do not occur:

⁷The size of this window, according to [5], is typically in the range of 100 milliseconds.

The address spaces in which guest operating systems exist are constructed and destroyed by so-called "head tasks" (see figure 3). The only time when these head tasks unmap pages is when they destroy their child's address space. In this special case, recursive unmapping is not needed since all address spaces that have been constructed on top of the one being destroyed are also destroyed anyway. Technically, a head task could revoke mappings without destroying the child's address space, but the PikeOS head tasks can be trusted to not do this. Since they are the parents of all user level processes and since a task must fully trust its parent anyway, this is not a new requirement.

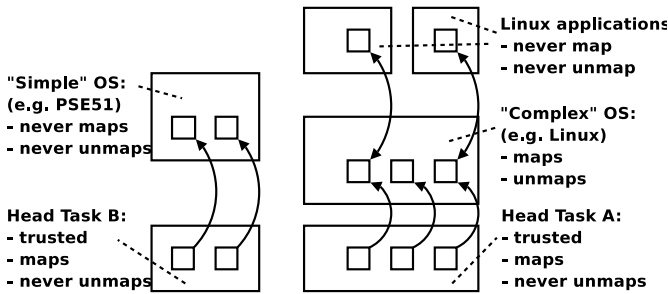


Figure 3. Simplified mappings in PikeOS

The range of guest operating systems that PikeOS can host in its partitions goes from relatively "simple", typically real-time and/or safety-critical systems, up to "complex" operating systems with virtual memory support (e.g. Linux). The former usually exist in a single, static address space. They do not require any mapping functionality themselves. In fact, these subsystems could do just as well with memory protection only. The latter implement their own memory management. These create tasks to whom they map pages, but those tasks do not create any subtasks themselves⁸ to whom they could map pages. Thus, all requests to revoke a mapping are either made from within a task itself or from its direct parent. These non-recursive unmap operations, however, can be implemented without a mapping database. Therefore, it was possible to remove it without substitution.

Recent research has shown formal verification of an L4 mapping database to be feasible ([23]). Thus, the code complexity argument made above against the mapping database may have lost some of its weight in the meantime. Nevertheless, the fact remains that validation/verification of a mapping database is a major task and, since its functionality is not required in the context of the PikeOS system, its removal still remains justified.

⁸This is usually enforced by not giving the tasks the "ability" (see below) to access the microkernel interface. And even without such an enforcement, the effects of then-possible violations would still remain confined to their partitions.

There exists at least one other L4 variant which, according to [19], also works without a mapping database [21]. Like PikeOS, this L4 variant explicitly addresses real-time, embedded systems, which suggests a similar motivation for this modification (although it has not been discussed in any paper we are aware of).

Kernel Resource Partitioning

To facilitate the accounting of kernel resources and thereby to avoid the possibility of denial-of-service attacks as described in [18], the concept of *kernel resource partitions* was introduced into the PikeOS microkernel. The approach is straightforward: The global kernel memory pool of L4 was split into a configurable subset of pools, which we refer to as *kernel resource partitions*. Every task is assigned to one of these partitions at creation time. Whenever the kernel allocates memory on behalf of a user thread, that memory is taken from the corresponding kernel resource partition. If the memory resources of a partition are exhausted, no further allocation for this partition is possible. Other resource partitions, however, are not affected. Thus, denial-of-service attacks are only possible among threads which belong to the same resource partition. The configuration of the resource partitions is done by a specific system call. The initial task, σ_0 , has the "ability" (see below) to make this call. Since all other tasks are ultimately children of σ_0 , and since σ_0 can be trusted to not pass this ability on to any of them, it is effectively the only task that is able to use this system call.

Clearly, this approach is a simplistic solution to the problem of kernel resource allocation: it is quite inflexible since all kernel memory resources have to be statically defined at boot time. Nevertheless, for the PikeOS system, it serves the purpose of enabling safe isolation between partitions while keeping the trusted code base small. Its inflexibility is not as big a problem in practice as it may seem: For real-time systems, it is usually not difficult to determine their worst-case kernel memory requirements beforehand and – unlike their time requirements – the discrepancy between worst case and average case is generally not as big. For general-purpose systems like Linux, kernel memory consumption is definitely not easy to predict. However, these systems can implement strategies for graceful degradation: The PikeOS Linux server, for example, flushes its client's mappings when its kernel resource partition runs out of memory, thus freeing up the kernel memory used for their page tables. This leads to a performance degradation akin to TLB thrashing, but, at least, the system remains stable.

The problem of managing kernel memory requirements has been a topic of active research in the L4 community for several years, and it still is. Several approaches have been proposed (e.g. [8, 6], but apparently, the final solution is

yet to be determined. These approaches aim to solve the far more challenging problem of dynamically changing kernel memory requirements, while PikeOS can live with a static, per partition allocation.

Mapping system call

To enable fast and simple creation of entire address spaces, the PikeOS kernel provides a special mapping system call. This system call is restricted to either the caller's address space, or to its child's address space. The receiver of such a mapping does not have to explicitly agree to receive the mapping. At first glance, this looks like a violation of Liedtke's principle that any transactions between address spaces must be explicitly agreed upon by all parties [16]. However, since the source of the mapping in this case is either the task itself (i.e. a trusted party), or its parent, which could just as well create a thread in the target address space under its own control that would then accept the mappings, this is not really a security risk. Although not strictly necessary, this facility greatly simplifies the process of constructing address spaces at the user-level side, while the added complexity at the kernel side is almost negligible.

Events

Many of the services that were implemented on top of the PikeOS microkernel have a requirement for a basic, asynchronous communication primitive (e.g. a counting semaphore). The L4 API only supports synchronous IPC. A counting semaphore can easily be implemented in user space based on the IPC call, so, according to the principle of minimality, this is how it should be done. However, this would require a user thread managing the semaphore object. The resulting memory overhead⁹ was considered high enough to justify a slight violation of the minimality principle by introducing a new concept, the *event*, into the kernel interface.

Events are counting semaphores associated to threads (i.e. every thread implicitly has one). A thread can wait on its event by making an appropriate system call. This decrements the event counter and it blocks the caller if the resulting counter is less than zero. The counter is incremented by other threads signalling events to the thread. If the counter is incremented to zero, its associated thread is unblocked again. The rules for sending events are similar to those for IPC, i.e. the recipient must explicitly allow any potential sender threads to signal events.

⁹I.e. 8 kB for a thread that implements a counter – a 32-bit object!

Access Privileges

The PikeOS kernel provides functionality to restrict access to the microkernel's system call interface on a per task basis. To implement this, the concept of *abilities* was added to the kernel: Each ability enables access to a set of system calls. The kernel checks a task's abilities with each system call. Depending on the settings, attempting a system call without sufficient abilities either results in an error, or an exception message being sent to the caller's exception handler. The latter can be used to virtualise system calls.

Abilities are assigned to a task during its creation by the creating *parent* task. They are thus a task property and are stored in the corresponding task descriptor data structure which is maintained by the microkernel. They can not be changed during the lifetime of the task. The kernel ensures that a parent can not give its child any abilities that it does not have itself, i.e. a parent can further restrict its child's abilities, but it can not extend them. It is possible to either disallow certain groups of system calls based on their specific functionality (like system-calls manipulating threads, etc.), or to disallow any system calls at all.

The concept of abilities –although developed independently– is similar to the fine-grained kernel access privileges of MINIX 3 ([9]). The L4::Pistachio kernel variant supports the notion of "privileged threads", which are entitled (by virtue of being members of an elite group of tasks) to make certain system calls. This also bears some similarity with PikeOS's abilities, however the selection of accessible system calls is pre-set and the right to access them can not be passed to tasks outside of the elite group.

To control IPC communication, the PikeOS kernel assigns *communication rights* to each task. A thread is only allowed to send an IPC or to signal an event to another thread if its task has the right of communication with the destination task. By default, every task is only allowed to communicate with its parent, but a parent is able to both grant the communication right between any of its children (i.e. siblings) and to grant communication rights with arbitrary, unrelated tasks.

4 Conclusion

The PikeOS microkernel is an early spin-off from the line of L4 kernel development, which has been adapted to the specific needs of safety-critical real-time embedded systems. It features partitioning of both temporal as well as spatial resources. Compared to other L4 variants that have been developed in parallel, it focuses on real-time computing and minimising trusted code, sacrificing flexibility in some cases.

It currently runs on various PowerPC, MIPS and ia-32

machines. A number of different real-time as well as non real-time operating system interfaces have been ported to run in the microkernel's partitions. Among them are Linux, POSIX PSE51, ITRON, OSEK OS, an Ada runtime system, a JVM and a Soft-PLC runtime system. Some performance comparisons between the PikeOS Linux server and native Linux on x86 platforms have been published in [14]. The average performance impact of 22% is slightly higher than that of other microkernel-based Linux implementations, however, optimizing the performance of Linux has always been a secondary concern for this system.

5 Outlook

Current development of PikeOS goes in several directions:

- Single threaded kernel: The PikeOS microkernel –like Fiasco ([11])– was designed to be fully preemptive. However, a fully preemptive kernel is always more complex, opening up many possibilities for, e.g., race conditions which are hard to detect. It also requires more resources, and the transient time spent in the kernel is very short anyway, so, a preemptive kernel may not really be worth the effort. Thus, in hindsight, this idea should be reconsidered.
- Multiprocessor: With the advent of multicore-CPU's, embedded multiprocessor systems are becoming feasible. The PikeOS microkernel was not designed for multiprocessors, so this must be remedied. Besides being a necessary step to support future processor generations, this also opens up the interesting new prospect of finally solving the conflict between time-driven and event-driven threads by binding them to separate processor cores.
- Certification: An effort to certify an appliance using PikeOS to DO-178B ([20]) is currently underway.

References

- [1] ARINC. Avionics Application Software Standard Interface. Technical Report ARINC Specification 653, Aeronautical Radio, Inc., 1997.
- [2] P. Barham, B. Dragovic, K. Fraser, S. H. T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization, 2003.
- [3] G. Bollella. *Slotted priorities: supporting real-time computing within general-purpose operating systems*. PhD thesis, University of North Carolina at Chapel Hill, 1997. Advisor: Kevin Jeffay.
- [4] J. Bonwick and Sun Microsystems. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1997.
- [5] D. Dodge, A. Dank, S. Marineau-Mes, P. van der Veen, C. Burgess, T. Fletcher, and B. Stecher. Process scheduler employing adaptive partitioning of process threads. Canadian patent application CA000002538503A1, March 2006.
- [6] D. Elkaduwe, P. Derrin, and K. Elphinstone. Kernel data - first class citizens of the system. Technical Report PA005847, National ICT Australia and University of New South Wales, December 2005.
- [7] G. Fohler. Flexible Reliable Timing - Real-Time vs. Reliability. In *Keynote Address, 10th European Workshop on Dependable Computing*, 1999.
- [8] A. Haeberlen. User-level Management of Kernel Memory. In Proc. of the 8th Asia-Pacific Computer Systems Architecture Conference, Aizu-Wakamatsu City, Japan, September 2003.
- [9] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Modular System Programming in MINIX 3. *j-LOGIN*, 31(2):19–28, April 2006.
- [10] M. Hohmuth. Linux-Emulation auf einem Mikrokern. Diploma Thesis, TU Dresden, August 1996.
- [11] M. Hohmuth. The Fiasco kernel: Requirements definition, 1998.
- [12] R. Kaiser. Scheduling Virtual Machines in Real-time Embedded Systems. In S. A. Brandt, editor, *OSPERT 2006 Workshop on Operating Systems for Embedded Real-Time applications*, pages 7–15, July 2006.
- [13] R. Kaiser and R. Fuchsen. Verfahren zur Verteilung von Rechenzeit in einem Rechnersystem. German patent DE102004054571A1, November 2004.
- [14] R. Kaiser, S. Wagner, and A. Zuepke. Safe and Cooperative Coexistence of a SoftPLC and Linux. 8th Real-Time Linux Workshop, Lanzhou, China, September 2006.
- [15] J. Liedtke. Clans & chiefs. In *Architektur von Rechensystemen, 12. GI/ITG-Fachtagung*, pages 294–305, London, UK, 1992. Springer-Verlag.
- [16] J. Liedtke. On μ -Kernel Construction. In *SOSP*, pages 237–250, 1995.
- [17] J. Liedtke. L4 Reference Manual - 486, Pentium, Pentium Pro, 1996.
- [18] J. Liedtke, N. Islam, and T. Jaeger. Preventing Denial-of-Service Attacks on a μ -Kernel for WebOSes, 1999.
- [19] NICTA – National ICT Australia. Embedded - ERTOS. Online: <http://ertos.nicta.com.au/research/14/embedded.pml>, 2006.
- [20] RTCA. Software Considerations in Airborne Systems and Equipment Certification. Guideline DO-178A, Radio Technical Commission for Aeronautics, One McPherson Square, 1425 K Street N.W., Suite 500, Washington DC 20005, USA, March 1985.
- [21] S. Ruocco. Real-Time Programming and L4 Microkernels. National ICT Australia, 2006.
- [22] J. Sugeran, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14. USENIX Association, 2001.
- [23] H. Tuch, G. Klein, and G. Heiser. Os verification — now! In M. Seltzer, editor, *Proc. 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, 2005.

Issues in Analysing a L4 for its WCET

Mohit Singal[†] Stefan M. Petters^{‡◇}

[†] Computer Science and
Engineering, IIT Guwahati
Assam, India

[◇]National ICT Australia*
Sydney, Australia

[‡] School of Computer Science
and Engineering, UNSW
Sydney, Australia

Abstract

Real-time analysis of a system requires knowledge of the worst-case execution time of all code in the system. This requirement covers not only application code, but also operating system and kernel code. In this paper we discuss the issues specific to kernel code and how we aim to address these in our work towards analysing the *L4* microkernel for the worst-case execution times of all system-call primitives. The main focus in that process is to maximise the degree of automation of the analysis, as the analysis needs to be repeated for any subsequent version of the kernel.

1 Introduction

Embedded real-time systems are becoming in general increasingly complex. While simple systems remain numerous, the number of more complex high end embedded systems is steadily growing and their complexity makes the use of a realtime operating system (RTOS) more or less mandatory. Even more, robustness requirements and the integration of functionality formerly implemented on a set of loosely coupled CPUs onto a single chip both require memory protection similar to a desktop or server system. The assumptions of the work in worst-case execution time (WCET) analysis have mostly been focused on the application domain. This involves, for example, the assumption of planar code or requirements of universal knowledge of memory accesses. The *Potoroo* project we have embarked in aims to analyse the NICTA developed version of L4 microkernel N1 embedded API. To avoid any confusion we will denote the L4 kernel in NICTA N1 embedded API *L4 NI* throughout the paper.

*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.

In embedded systems, this kernel is mainly targeted for mission critical and consumer electronics systems. This paper discusses the issues, that need to be solved for this specific kernel. It also examines if and how these issues might be addressed by related work.

While *Potoroo* aims to be largely target independent, the analysis is currently performed on an ARM [1] processor based platform and some of the issues reported are specific to the ARM architecture. The choice of ARM is mainly driven by the use of *L4 NI*. While ports to other architectures exist, the ARM port of this kernel is the most progressively developed. However, the processors implementing the ARM architecture do exhibit many of the problems experienced on other architectures and thus is a good reference point for this and future work.

To our knowledge Colin and Puaut [2] published the only other work addressing the WCET analysis of an operating system kernel, in their case the RTEMS kernel. However, RTEMS does not provide the memory protection offered by *L4 NI* and their work did not cover the entire kernel. Our work aims to cover all runtime relevant parts of the *L4 NI* kernel and aims to provide an environment which allows the WCET analysis for a given hardware platform outside the academic lab environment. Mehnert et al. [3] have looked into the cost of address spaces, but have not fundamentally addressed the issue of WCET analysis of the kernel. Commercially available kernels are sometimes delivered with representative sample execution times for some kernel primitives on a give platform. However, these execution times are always expressively exempted from being in any way guaranteed.

The next section will briefly introduce our approach to WCET analysis. Section 3 digs into a detailed analysis of the issues we have encountered during our effort to analyse *L4 NI*. Finally Sections 4 and 5 conclude the paper with an outlook into future work and a summary of the papers findings.

2 WCET Approach

Many issues will be explained in the context of our WCET approach. As such we consider it helpful to briefly introduce our approach and toolset prior to actually discussing the issues experienced during analysis so far.

Our approach is measurement based, but instead of using the end-to-end measurements and safety factors common in industry, we use measurements obtained on the basic-block level together with a tree and a timing schema to compute the WCET of the kernel primitives. Using the tree allows us to implicitly cover any possible path through a kernel primitive, thus ensuring that WCET is not underestimated based on the path executed. Opposed to end-to-end measurements basic blocks exhibit their respective WCET much more easily, as the numbers of different execution times for a basic block are usually much smaller due to a limited number of variation causing input states and cache misses. However, in order to provide guarantees that the WCET has been observed on the basic-block level we are also working on a static analysis approach [4]. Within this work we are aiming to establish the number of cache misses which could and should be observed during the measurements performed and compare the results obtained by static analysis with the measurements. The base line is to avoid detailed hardware modelling, but rather to stick to first order effects like cache misses in order to keep the static analysis light weight.

Our toolset is depicted in Figure 1. In terms of building blocks it is very similar to the pWCET toolset used by Colin et al. [5, 6]. The major difference is a shift in associating the computational weight in terms of WCET onto the edges of the control flow graph (CFG) rather than the nodes used in the previous work. This increases accuracy of the analysis by separating effects caused for example by a branch prediction unit into separate entities. However, more important for our work are the changes in the subsequent tree representation and timing schema, which allow us to deal efficiently with non well-structured code.

The kernel binary image is the base for our analysis. By analysing primarily from the compiled and linked executable, we avoid second guessing the effects of the compiler. The generation of traces from the executable may either be intrusive via instrumentation code added to the executable or non intrusive, via hardware supported tracing mechanisms or cycle accurate simulations. Whenever available hardware supported tracing will be used, as it is non-intrusive and is not subject to the question of whether the simulator matches 100 % the hardware.

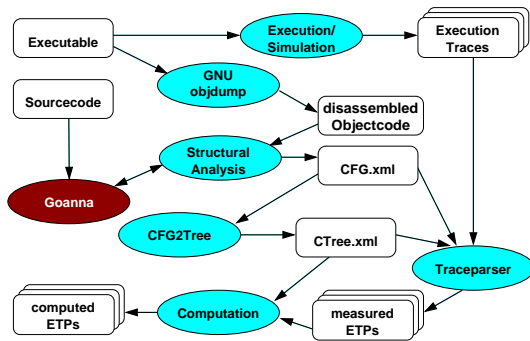


Figure 1: Toolchain Overview

The extraction of a control-flow graph (CFG) from the kernel binary image is split into three steps. In a first step objdump from GNU binutils is deployed to disassemble the code. This circumvents the problem of dealing with different binary formats. As such it minimises the hardware dependent part of the toolset. The second step of translating the code into a base CFG is left to a comparably simple program. Currently this program only deals with ARM code but can be ported to other CPUs with moderate effort. Besides analysing the output of objdump it also queries the *Goanna* [7] tool (which we have used as source code analyser/parser) to obtain additional information to fill in information missing in the object code analysis. The details of this interaction will be discussed in later sections of this paper. In a third and architecture independent step the CFG is augmented with various metadata which can be obtained with some effort from the base CFG. The metadata consists, for example, of loop-nesting levels, backward edges etc. This step has been separated from the second step to keep the architecture dependent part modular. In Figure 1, the second and third step are for simplicity of the representation joined into the process of *structural analysis*.

The CFG generated by the previous step is used to generate the tree representation with *CFG2Tree*. The parent nodes may either be of type sequence, alternative, or loop while the leaf nodes of the tree represent the transitions in the CFG. The leaf nodes may contain a call to another function. The traceparser uses the CFG, which actually describes already all the possible transitions which may occur, as well as the tree representation to convert the execution traces into measured ETPs. The traceparser does not only produce ETPs for all the leaf nodes of the tree, but also of parent nodes. This is useful when trying to track down where and why overestimations are introduced in the later computation process. For this the computed and the measured ETPs

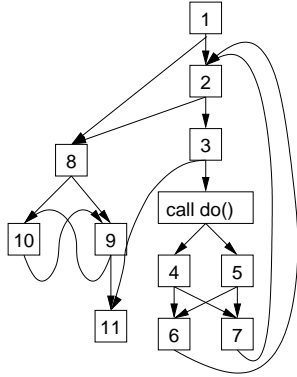


Figure 2: Sample Control-Flow Graph of Structures Found in the *L4 NIMicrokernel*

can be directly compared.

Finally the computation stage takes the schema rules to produce ETPs for the parent nodes of the tree. Sequences form simple additions, alternatives use the max operator and loops use multiplication with the number of loop iteration and add the loop entry and loop exit.

3 WCET Analysis of an RTOS Kernel

In this section we discuss the main issues encountered when analysing *L4 NI*. However, similar problems can be expected when analysing any other kernel. The constructs and issues listed below are often used in operating system kernels or are caused by compiler optimisations. *L4 NI* source exhibits a reasonable number of these. As removal of these would heavily affect the performance of the kernel, we deem that we have to work around the problem, rather than avoiding it by imposing strict coding rules and switching compiler optimisations off. Figure 2 depicts a number of constructs in the CFG for illustration purposes. In reality the constructs are much larger and span up to several dozens of nodes in the control-flow graph.

3.1 Non Well-Structured Code

In an RTOS kernel there is deliberate deviation from structured coding, in particulara with regards to the use of `goto` statements. The current implementation of *L4 NI* for ARM processors contains more than 20 `goto` statements. This number is not including non well-structured code written in assembly. Use of such coding technique to optimise the kernel leads to deviation from properly defined structures and thus, introduces

more complexity to resolve. For example, consider the edge between node-4 and node-6 (edge 4-6) in Figure 2. Three other transitions namely, 4-7, 5-6, 5-7 provide alternate paths through which control can flow. Such non planar code can be resolved by duplicating some of the nodes in the tree.

Syntax-tree based approaches, like the one used by Colin and Puaut [2] as well as the approach by Theiling et al. [8], which uses an integer linear programming approach, should technically be able to deal with such code. However, well-structured code is a typical restriction of many static WCET analysis approaches.

A similar problem exists with irreducible loops as formed by nodes 9 and 10 in Figure 2. Again this can be resolved by duplicating nodes. Currently our toolset has not yet implemented an algorithm to identify irreducible loops like the one presented by Sreedhar et al. [9] and as such this is manually resolved.

3.2 Multiple Loop Exits

The use of `break` or `return` statements in loops leads to multiple points of exit out of these loops. *L4 NI* for ARM currently contains 18 `break` statements in loops. `return` statements within loops are translated by the gcc ARM compiler into branches to the end of the function. This leads to code being *virtually* shared by the loop exit which results in the main function body bypassing the loop if the loop is part of a conditional. The transition 3-11 in Figure 2 demonstrates such a situation. It is an issue, since the loop exit notionally stretches to the nearest common node outside the loop, which in this case is the return node at the end of the function. Such code exists in various locations like, for example, the IPC slowpath implementation. Within our approach this is solved by duplicating the code shared between the loop exit and the main sequence bypassing the loop.

3.3 Inline Assembly

The introduction of inline assembly text into the source code in turn introduces difficulties when querying a source code analysis tool like *Goanna*. This typically occurs in sections of kernel which are expected to be executed several more times than others. Assembly text is inserted in 23 places in the current *L4 NI* implementation.

3.4 Assembly Files

Besides inline assembly, the kernel also has a considerable code written in assembly. This covers, in partic-

```

pistachio/kernel/include/arch/arm/ptab.h:51
f0008170: e3520007  cmp    r2, #7 ; 0x7
;; Switch statement indirect jump
f0008174: 979ff102  ldrls  pc, [pc, r2, lsl #2]
f0008178: ea0000c2  b      f0008488
;; Jump table begins after an instruction
f000817c: f0008470  andnv  r8, r0, r0, ror r4
f0008180: f000847c  andnv  r8, r0, ip, ror r4
f0008184: f0008488  andnv  r8, r0, r8, lsl #9
f0008188: f000847c  andnv  r8, r0, ip, ror r4
f000818c: f0008494  mulnv  r0, r4, r4
f0008190: f000847c  andnv  r8, r0, ip, ror r4
f0008194: f0008488  andnv  r8, r0, r8, lsl #9
f0008198: f000841c  andnv  r8, r0, ip, lsl r4

```

Figure 3: Switch statement from *L4 NI* kernel objdump

ular, the trap code resolving interrupt handling and the performance critical IPC fastpath. Being highly optimised code, it adheres to little convention in terms of standard C or C++ compiled code. In particular it introduces irreducible loops such as the transitions 9–10 and 10–9 in Figure 2. As mentioned earlier such loops need duplication of nodes to be represented within the tree. Additionally, the detection of these loops and translation into tree is non-trivial. In particular, the irreducible loops within the kernel span more than 10 CFG nodes.

3.5 Indexed Jumping

Indexed jumping occurs when there are multiple cases in a switch statement. The compiler creates a hash table of all the case addresses and makes an indirect jump to these while optimising the code. Control flows to the respective case depending upon the address stored in a register or by directly indexing the hash table. Ultimately, we need to obtain edges to all possible branch targets contained in the *jump table*

As a first approach, line references in source code seem to be an answer to the location (address) of each case body. But this is not true since an optimising compiler distorts the resulting object code (in many cases even merges several cases, extracts common statements, etc).

Knowing the typical anatomy of switch statements, we can reconstruct the possible control flow with moderate effort by parsing the jump table itself. Parsing a jump table involves the main issue of identifying it correctly. This task becomes more difficult when the table is embedded in the code segment by the compiler. We have observed that the jump table always lies one instruction after the switch statement indirect jump. This behavior being constant in all our cases, including

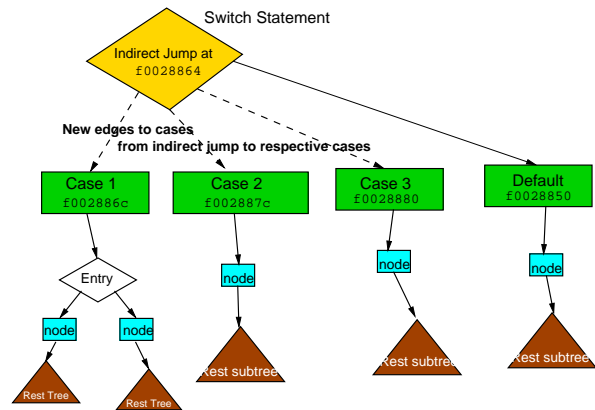


Figure 4: Corresponding graph after patching switch statement

the *L4 NI* kernel, we chose to use this as an identifying criteria. (see Figure 3, code modified for best view)

We can patch the switch statement by extracting the addresses of each case statement and creating an edge to it. This is illustrated in Figure 4 where we have drawn three more edges from switch indirect jump node to the respective case.

3.6 Returns

The kernel version investigated contained a large number of register indirect jumps. Technically closely related to the above, the problem is that there is no jump table which can easily be identified. In most cases these are compiler generated return statements. In ARM the register *lr* contains the return address for a function call. For recursive functions *lr* is pushed onto the stack which can be easily identified. However, in some cases the function is so small that it can make use of another local register which is used to store *lr* prior to a call to another function and later moved onto the *pc* to implement a return statement. This requires tracking of registers within the tool evaluating the output of objdump to distinguish return statements from genuine register indirect branches or function calls. For the time being it is not planned to implement full tracking of all register content, but rather tracking of where the return address of a function is stored. Alternatively we might deploy lookups in the source code using *Goanna* to solve this problem.

3.7 Function call targets

This issue refers to a set of locations created by the programmer himself, which are not easily retrievable from

```

for (int i = 7; i < IRQS; ++i) /* 0..6 are reserved */
{
  if (status & (1ul << i)) {
    void (*irq_handler)(int, arm_irq_context_t *) =
      (void (*)(int, arm_irq_context_t *))interrupt_handlers[i];
    irq_handler(i, context);
  }
  return;
}
}

```

Figure 5: *L4NI* kernel interrupt vector array indexed in a loop

the object code. A good example of this type of coding would be an *interrupt vector table*, which distributes incoming hardware interrupts to registered handlers. In the *L4NI* kernel code, a jump to these routines is made through an array of function pointers. Since the code is accessing a global array, ascertaining where this array is initialised is quite tedious in the sense that it may be initialised anywhere in the source code spanning considerable number of files. In addition, the initialisation may be obscure or actually happening dynamically at runtime. Since the source code analysis by *Goanna* has so far been unable to identify the content of global variables, manual analysis is the only way to describe these constructs. However, there are only few locations in the kernel which make use of this kind of function calling and thus the required manual intervention is limited.

Besides knowing the targets of called functions, there is also the issue of encoding it appropriately in the CFG and tree. Since our toolset allows only one `call` per CFG node, we need to circumvent the problem. We have done that by allowing stand alone function call nodes, which have no measured execution time themselves. This is useful for encoding alternative functions to be called and is used, for example, in the *L4NI* kernel debugger where depending upon an environment variable either one or the other function is called. Although the kernel debugger is irrelevant for our analysis, there are some constructs in this part of the code which are interesting for analysis. Furthermore, constructs similar to these may be included later during development of kernel. A special case is where the function array is indexed by a loop control variable or any descendant of that as has been used in the *L4NI* interrupt vector table (see Figure 5, extract taken from `irq.cc`). In this case only one of the target functions is executed for an interrupt. However, the latency is different since the code checks each bit of the interrupt mask with each loop iteration until it hits the correct one and calls the handler function.

We can apply this strategy in conjunction with unrolling of loops to solve the problem of multiple targets where an interrupt vector array has been used in

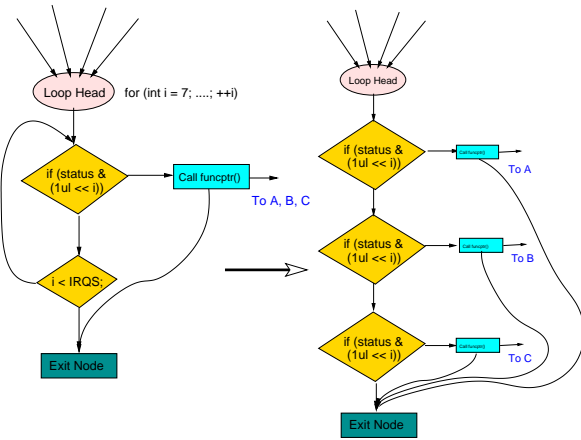


Figure 6: Stand alone nodes with loop unrolling technique for interrupt vector array

source code to address different interrupt handlers (see Figure 6).

3.8 Context Switch Jumps

Context switches are subject to three problems.

1. Context switches are nontrivial to positively identify without creating false positives. Thus it is considered inevitable to manually identify these. However, again there are only very few places in the kernel that actually perform a context switch, keeping the intervention at this stage very small. A result of the context switch is that the execution may transfer from any of the context switch nodes to any other node containing a context switch, requiring our tool to provide appropriate transitions in the automata performing the trace parsing.
2. After a context switch, an *asynchronous notification* may be delivered. This happens because a notification is issued when the sending thread is running and the receiving thread is not (this assumes a single threaded, single processor CPU). The current implementation of an *asynchronous notification* pushes a notification stack frame onto the stack of the receiving thread, thus executing the receive function prior to resuming the execution when the context switch passes control to the receiving thread. This is resolved by notionally adding a context switch to the start and end of the notification routine, allowing the *Traceparser* (which is part of the toolset) to switch to the notification and back from it. The time of the asyn-

chronous notification needs to be dealt with, depending on what the result of the analysis is to be used for. In the case of latency analysis, the time for the *asynchronous notification* needs to be added to the called function while for schedulability analysis, this needs to be considered separately as part of the communication cost.

3. Finally from a trace parsing point of view, performing a context switch means a `return` after a context switch no longer corresponds to the `call` performed before doing the context switch. As such again, the *Traceparser* needs to allow for `return` statements to connect to any possible location than only from where the returning function was called. Unfortunately this takes away some of the sanity checking available when doing the analysis such as checking that a `return` is returning to the place the function was called from. However, the likelihood of a trace which is corrupted in such a particular way is very low.

3.9 Portability

As opposed to application code, which is built on top of hardware abstractions (standard libraries, and kernel), the kernel is supposed to be deployed on a variety of different systems. Having multiple target architectures supported by the kernel requires that the WCET estimation technique be portable, since it needs to be available for all target architectures. This is a major challenge as the work over the years has shown that this is not a trivial task. The support for multiple architectures is often reflected by many `#defines`. The `#define` is efficient and easy to use, but makes the code harder to read. Since manual intervention in the analysis is almost inevitable, the analysis of a kernel requires detailed knowledge of the kernel as well as fundamental understanding of WCET analysis.

3.10 Memory Management

A problem reported by Colin and Puaut [2] for the RTEMS kernel is code in the memory allocation can produce extremely long execution times that exceed the average execution time by a large margin. *L4 NI* suffers in the same manner during the unmapping of memory regions. However, this is only of theoretical relevance, as we would expect real-time applications to only make use of this system call when shutting down or doing exception handling. Obviously this still leaves the issue of non real-time applications effectively blocking the kernel while performing such a call as they shut down. It

is expected that the issue will be addressed on the kernel side by changing the way memory management is handled.

3.11 Parametric WCET

Run-time parameter dependent worst case execution times have also been experienced by Colin and Puaut [2]. This can either be caused by system parameters (e.g., the number of threads sending messages to a specific thread in the system) or caused by structural parameters (e.g., what kind of inter-process communication (IPC) is used for a specific system call in *L4 NI*). The IPC example is caused by the fact that all IPC functions in *L4 NI* use the same system call, but with different parameters.

As kernel code is quite complex to understand, the analysis of this code needs expertise in WCET analysis and OS construction. Otherwise the intrinsic interaction between different parts of the kernel may be misinterpreted or overlooked completely. In order to separate out different invocations of the same primitive (e.g., receive only IPC, send only IPC, IPC payload size, etc.) we currently need to manually remove irrelevant parts of the respective CFG. Future versions will use code annotations to identify different parts of the primitives.

3.12 Rapid Evolution

L4 NI suffers from a very specific problem. The code is not fixed, but evolves rapidly over time, while at the same time the approach to analyse it is being developed and refined. Opposed to applications which are written and then deployed, different snapshots of the kernel will be deployed. Thus the analysis has to be performed repeatedly on slightly different versions of the kernel. This offers problems and opportunities. On one hand it requires the WCET approach used to require minimal user interaction, on the other hand it enables the use of annotations in the code.

Furthermore *L4 NI* uses memory protection and virtual addressing, which distinguishes it from most of the real-time operating systems around and is motivated by the fact that partitioning and fault isolation is a highly desirable feature in complex embedded systems. Static analysis requires the modelling of translation-lookaside buffers (TLB), which adds to the state space. For measurement-based approaches, this adds to the variability of the code, depending on the number of TLB misses. However, the kernel itself currently makes little use of the virtual memory, but applications analysed on top do suffer from this.

L4 NI is coded in C++. While only a very limited subset of C++ is chosen, it nevertheless creates an additional engineering effort in the analysis approach. However, on the other hand, *L4 NI* as a microkernel is small compared to monolithic kernels which in turn makes the analysis much more tractable.

4 What's Next

Future work can be split into two categories. One which is related to development of the kernel itself and the other which looks at future tool enhancements.

So far the work has been carried out on a working snapshot of the kernel. Due to the experimental nature of the work it does not seem practical to track all changes to the kernel as they are made. The most substantial change of *L4 NI* in the last half year was the move to a single stack kernel. This implies the dissolution of the call/return relationship and subsequently a substantial change in the context switch modelling.

The next step is to look into a multi-processing version of the kernel. This includes looking at more fundamental issues of real-time in multi-processing environments and is in itself a large project.

On the tool-set and approach side of things, further automation and support for other architectures is on the agenda. This specifically covers the areas of

- register tracking, to automatically resolve more control flow instructions;
- irreducible loop identification and resolution;
- allowing for source code annotations to be taken into account.

The source code annotations are particularly relevant to provide separate WCETs for different but closely related kernel primitives. Besides these automation issues, we also want to continue working on the static analysis support for the approach [4].

5 Conclusion

In this paper we have listed a number of issues we have encountered in our effort to analyse the *L4 NI* microkernel for the WCET of all kernel primitives and how we resolved these. While we have mainly looked at *L4 NI* the insights should translate to a number of other kernels. The small footprint of *L4 NI* compared to monolithic kernels has certainly been helpful in keeping complexity of the analysis within manageable levels. Besides

the worst-case analysis, the approach can support kernel development in a number of ways. Hot-spot analysis can identify code portions which account for larger parts of the execution time both in terms of execution frequency as well as execution time, and thus help directing optimisation efforts. Furthermore our approach can also be applied to detect dead code or code not covered in the regression tests.

References

- [1] *ARM 7TDMI Data Sheet*, August 1995. ARM DDI 0029E.
- [2] A. Colin and I. Puaut, "Worst case execution time analysis of the RTEMS real-time operating system," (Delft, Netherlands), pp. 191–198, June 13–15 2001.
- [3] F. Mehnert, M. Hohmuth, and H. Härtig, "Cost and benefit of separate address spaces in real-time operating systems," (Austin, TX, USA), 2002.
- [4] S. Schaefer, B. Scholz, S. M. Petters, and G. Heiser, "Static analysis support for measurement-based WCET analysis," in *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Work-in-Progress Session*, (Sydney, Australia), Aug. 2006.
- [5] A. Colin and S. M. Petters, "Experimental evaluation of code properties for WCET analysis," in *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, (Cancun, Mexico), Dec. 3–5 2003.
- [6] G. Bernat, A. Colin, and S. M. Petters, "WCET analysis of probabilistic hard real-time systems," (Austin, Texas, USA), pp. 279–288, Dec. 3–5 2002.
- [7] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch, "Goanna — A Static Model Checker," (Bonn, Germany), Aug. 2006.
- [8] H. Theiling, C. Ferdinand, and R. Wilhelm, "Fast and precise WCET prediction by separated cache and path analysis," vol. 18, pp. 157–179, 2000.
- [9] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee, "Identifying loops using dj graphs," vol. 18, no. 6, pp. 649–658, 1996.