High-performance microkernels and virtualisation on ARM and segmented architectures

Author:
van Schaik, Carl; Heiser, Gernot

Publication details:
1st international workshop on microkernels for embedded systems: MIKES 2007, Proceedings
pp. 11-21
1833-9646 (ISSN)

Event details:
1st international workshop on microkernels for embedded systems: MIKES 2007
Sydney, Australia

Publication Date:
2007

DOI:
https://doi.org/10.26190/unsworks/515

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 descendents 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.

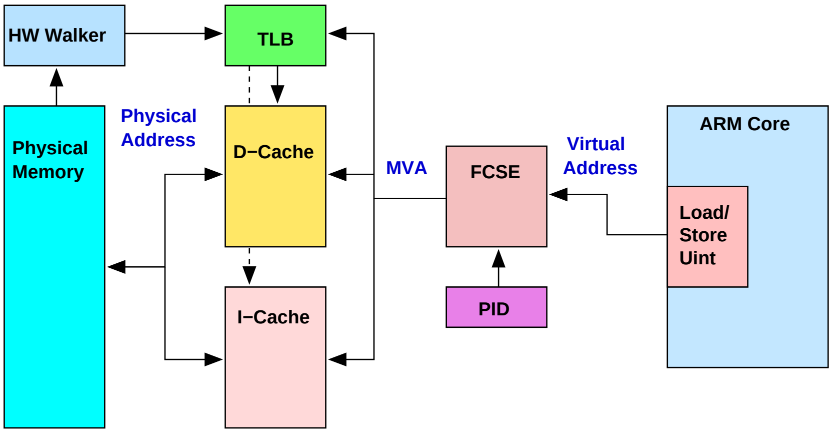

Figure 1: ARM MMU structure.

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 bandwidth, 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 remapped 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 access 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 section 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 UCTB 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 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, eliminating the indirect costs of flushing.

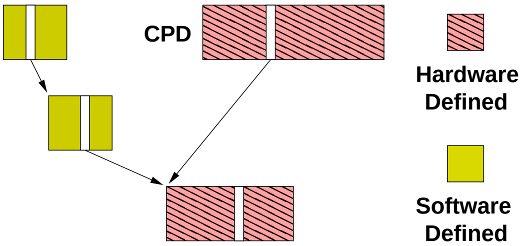

Figure 2: L4 software compressed page tables with the CPD.

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 underutilised 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

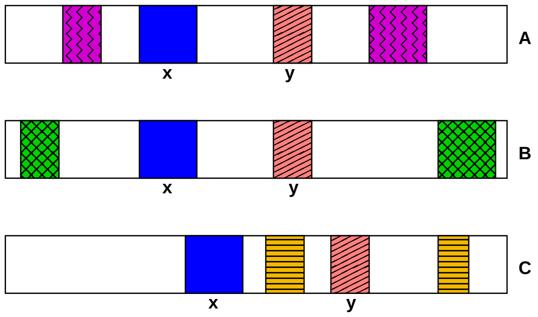

Figure 3: Three address spaces, with shared segments x and y.

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 poulated 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.

---

[1] This 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 *vspaces* 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. Componding 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 where 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-

---

[2] While 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 [μs]	before [μs]	after [μs]	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 benchmark 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 bandwith 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 aproach 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 is 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 Vochteloo, 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.