Fast Address-Space Switching on the StrongARM SA-1100 Processor

Adam Wiggins and Gernot Heiser
School of Computer Science & Engineering
University of New South Wales
Sydney 2052, Australia
{awiggins,gernot}@cse.unsw.edu.au, http://www.cse.unsw.edu.au/∼disy

Abstract

The StrongARM SA-1100 is a high-speed low-power processor aimed at embedded and portable applications. Its architecture features virtual caches and TLBs which are not tagged by an address-space identifier. Consequently, context switches on that processor are potentially very expensive, as they may require complete flushes of TLBs and caches.

This paper presents the design of an address-space management technique for the StrongARM which minimises TLB and cache flushes and thus context switching costs. The basic idea is to implement the top-level of the (hardware-walked) page-table as a cache for page directory entries for different address spaces. This allows switching address spaces with minimal overhead as long as the working sets do not overlap. For small ($\leq$ 32MB) address spaces further improvements are possible by making use of the StrongARM's re-mapping facility. Our technique is discussed in the context of the L4 microkernel in which it will be implemented.

1. Introduction

The StrongARM SA-1100 [5] is a high speed, low power processor based on the ARM architecture [6]. It is specifically designed for portable and embedded systems. The design is based around a first generation StrongARM core and has peripheral controllers (DRAM controller, serial ports, etc.) integrated into a single package.

To achieve a high clock rate of $\geq$200MHz, the core makes use of a Harvard architecture featuring separate translation-lookaside buffers (TLBs) and caches for data and instruction streams. Being targeted at applications which traditionally do not use multitasking operating systems, the design has minimised support for multiple address spaces. The caches in particular are virtually indexed and virtually tagged and the TLBs are not tagged with an address-space identifier.

A context switch between threads belonging to different tasks (and thus address-spaces) implies a change of virtual address mappings and thus a change of page tables. On an architecture which does not tag the TLBs with an address-space identifier, this normally implies that the TLBs must be flushed, as they would contain incorrect translations for the thread being scheduled. This not only implies some *direct* overhead for invalidating all TLB entries, it also implies significant *indirect* costs, as the thread, once it starts executing, will experience a number of TLB misses until its working set is mapped. Each TLB miss requires a costly page table lookup (could be 20–100 cycles per TLB miss on the StrongARM).

Similarly, virtual caches may contain stale data after a context switch, which would lead to incorrect execution of the threads. Unless the kernel is certain that no stale data exists in the caches, they must be flushed as well. Again, this has a direct cost of the flush operations, as well as an indirect cost, as the thread being scheduled starts with cold caches. The direct cost of cleaning (writing back) the data cache is particularly high since each line must be individually cleaned.

The purpose of this paper is to present an approach to providing fast address-space switches on the StrongARM, and discuss its proposed implementation in the L4 microkernel [1, 9, 10].

2. StrongARM Virtual Memory Architecture

In this section we summarise the StrongARM's virtual memory architecture as far as is relevant to the topic of this paper. We describe general ARM features and note which features are specific to the StrongARM.

2.1. ARM page table structure

The general ARM architecture specification leaves it up to the implementation as to whether to use a unified (instruction and data) TLB, split (i.e., separate instruction and data) TLBs or no TLB at all. As indicated earlier, the StrongARM uses split TLBs.

If available, the TLBs in the ARM architecture are hardware-loaded from a two-level page table. For clarity we will refer from now on to the top level of the page table as the *page directory* (PD), and to the second-level data structures as *leaf page tables* (LPTs).

The PD is an array of 4096 *section entries*. Each section entry can be in one of four states as follows:

Fault: The entry is invalid and will lead to a *translation fault* on access.

Page table: The entry contains a pointer to an LPT, as well as a *domain* identifier (see Section 2.3 for details) applying to the whole LPT.

Section: The entry maps a 1MB region of virtual address space to an aligned 1MB region of RAM. The entry contains a domain identifier and permission bits. The domain and permission bits apply to the whole 1MB region.

Reserved: This section entry state is reserved for future expansion and must not be used.

An LPT is an array of 256 *page table entries* (PTEs). Each PTE can be in one of four states as follows:

Fault: The entry is invalid and will lead to a translation fault on access.

Large page: The entry maps a 64kB page. The entry contains a single domain identifier which applies to the whole page, and four sets of permission bits, each set defining accessibility of a 16kB subpage.

Small page: The entry maps a 4kB page. The entry contains a single domain identifier which applies to the whole page, and four sets of permission bits, each set defining accessibility of a 1kB subpage.

Reserved: This section entry state is reserved for future expansion and must not be used.

It is clear from the above that the ARM supports three different page sizes from 4kB to 1MB (in powers of 16), and five protection granularities from 1kB up to 1Mb. Obviously, TLB use is optimised by choosing the largest possible page sizes.

The StrongARM has separate data and instruction TLBs which are fully associative and hold 32 entries each. The implementation allows invalidation of a single data TLB entry or of a whole TLB.

2.2. Caches

ARMv4 supports the implementation of processors with unified or split instruction and data caches. A write buffer, pre-fetch buffer and branch cache may also be implemented. Caches may be physically or virtually tagged; the choice is left up to the implementor of any particular ARM core.

2.2.1. StrongARM cache architecture. The SA-1100 uses split instruction and data caches. These are complemented by a read buffer, a write buffer and a mini-data cache. The details of interest are as follows:

Instruction cache: 16kB in total, organised into 512 lines of 32 bytes each. It is virtually tagged and 32-way set associative. A cache control operation allows flushing of the complete instruction cache.

Data Cache: Consists of a *main data cache* and a *mini data cache*, with the following properties:

Main data cache: 8kB in total, organised into 256 lines of 32 bytes each. It is virtually tagged and 32-way set associative and uses a write-back policy.

Mini data cache: 512B in total, organised into 16 lines of 32 bytes each. It is virtually tagged and 2-way set associative and uses a write-back policy. The mini cache is meant to be used for processing large data structures which would unnecessarily flood the main cache.

The *cachable* and *bufferable* PTE attributes define in which, if any, of the two caches a data line may reside; the hardware guarantees that no line can be in both caches at the same time. Stores do not allocate in either cache.

Cache controls allow *invalidation* of the whole cache or a single line. There is also a *clean* operation to force write-back of a single line.

Write buffer: Holds up to eight blocks of data, each 1 to 16 bytes in size. The use of the write buffer is controlled by the *bufferable* bit in the PTE corresponding to a data line.

There is a *drain* operation to empty the whole write buffer.

Read buffer: Holds up to four blocks of data, each 1 to 32 bytes in size. The read buffer is used to pre-fetch data (not instructions) and is fully under software control.

For the purposes of this paper the difference between the main and mini data caches as well as the read buffer are irrelevant and will be ignored. We expect to reserve the use of the read buffer and the mini-cache to the kernel. Hence the coherency of the read buffer and mini-cache is independent of user address spaces.

2.3. ARM domains and protection bits

2.3.1. Domains. ARM domains are a generalisation of the access permission bits found in many TLBs. ARMv4 supports 16 domains; each domain forms a collection of section and page mappings defined in a page table (or TLBs), tagged by the domain identifier.

A coprocessor register, the *domain access control register* (DACR), defines the present accessibility of each domain, by specifying each domain's state as one of the following:

No access: Any access to sections or pages belonging to this domain will generate a *domain fault* exception.

Client: Accessibility is determined by the setting of the standard permissions bits of the page table entry (with sub-page granularity).

Master: Access is permitted independent of the setting of the standard permissions bits of the page table entry.

Reserved: This state is reserved for future expansion and must not be used.

2.3.2. Protection bits. The ARM page table protection bits, along with the *system* and *ROM* protection bits of the system coprocessor's control register, allow specification of memory access rights as follows:

- no access,
- kernel R/O, user no access,
- kernel R/O, user R/O,
- kernel R/W, user no access,
- kernel R/W, user R/O, and
- kernel R/W, user R/W.

2.4. The StrongARM SA-1100 PID register

The SA-1100 exhibits one more relevant feature: the *process identifier* (PID) register. Whenever an address less than 32MB is issued, that address is, prior to translation into a physical address, bitwise OR-ed with bits 30–25 of the PID register. This re-maps a small 32MB address space into any aligned 32MB region of the lower 2GB half of the virtual address space (a total of 64 "slots").

As any addresses outside the lowest 32MB are not remapped, this introduces a form of aliasing in the address space. If $0 \leq x < 2^{25}$ then addresses x and $PID|x$ refer to the same memory location (with the same access permissions).

2.5. Discussion

Domains mitigate, to a degree, the problem of TLB entries not being tagged with an address-space identifier. They allow the kernel to map in or invalidate large and non-contiguous regions of virtual memory efficiently. This supports fast address-space switches, *as long as the kernel can guarantee that there is no overlap in mapped address spaces.* If this guarantee can be made then an address-space switch can be performed simply by reloading the DACR with an appropriate mask.

A similar observation can be made about caches. The architecture observes domains and protection bits even for cached data and instructions, not only on cache misses. Hence, as long as there is no overlap in mapped address space between threads, it is guaranteed that a context switch does not leave stale data in the caches. Caches need not be flushed in that case.

We can therefore conclude that fast address-space switches are possible on the StrongARM as long as the kernel ensures the above condition of no overlap between mapped address spaces. The core idea of this paper is a technique which will ensure that this condition always holds. This is discussed in detail in the following section.

3. Fast address-space switches

We now present our proposal on how to implement fast address-space switches on the StrongARM. The basic idea bears some similarity to Liedtke's work on fast address-space switches on the Pentium processor [8].

3.1. General idea

Normally an address-space switch is performed by replacing the page table. In the case of the StrongARM, where the hardware reloads the TLB from a two-level page table, this would be implemented by reloading the *translation table base* register with a pointer to the new page directory. This, by itself, is fast. However, the two address spaces would overlap in general, and TLBs and caches would have to be flushed.

For this reason we *never* change the translation table base pointer. Instead we have the register point to a *caching page directory* (CPD) which contains entries from a number

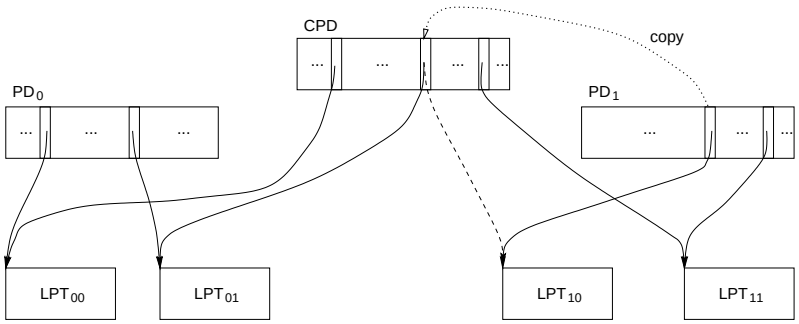

Figure 1. Caching page directory (CPD) and per-address-space page tables.

of different address spaces, each defined by its own page table.

Figure 1 shows the idea: On a TLB miss the hardware reloads the TLB by indexing into the CPD, which contains pointers to LPTs of various address spaces. Each of these address spaces is associated with a unique domain, and CPD entries are tagged with that domain. The DACR is set up to give *client* access rights to the domain associated with the address space of the presently executing thread, and *no access* to any other domains.

On a TLB reload, one of the following can happen when the hardware accesses a CPD entry:

Translation fault: The kernel's fault handler indexes the current thread's PD and, if it finds a valid entry, copies that into the CPD and restarts the thread. The hardware will again attempt to reload the TLB and this time find a *valid section* or *valid page table* entry. Otherwise the kernel treats the access as a page fault.

Valid section entry: The entry is a *section* entry and the DACR indicates a valid access. If the permission bits allow the attempted access, the hardware loads the entry into the TLB and proceeds. If there is a permission mismatch, the hardware raises a permission fault which is interpreted as an invalid access by the kernel.

Domain fault: The kernel handles this as in the case of a *translation fault*: The entry is reloaded from the current thread's PD, replacing the existing entry in the CPD. In addition, the cache must be flushed of any remaining entries corresponding to the replaced CPD entry. If the current thread's PD does not contain a valid entry, the kernel treats the access as a page fault.

Valid page table entry: The entry is a *page table* entry and the DACR indicates a valid access. The hardware follows the pointer to some address-space's LPT which it indexes to find a PTE. If this PTE contains a *large* or *small page* entry, the hardware loads the entry or raises an permission exception as in the case of a *valid section* entry in the CPD.

Remember that domains and permission bits are checked on every access, not only on a TLB load. Hence most of these faults can also occur when accessing cached data. This complicates handling somewhat, as in some circumstances the TLB must also be flushed, which wouldn't be necessary if the fault arose as the result of a TLB miss.

The CPD therefore caches PD entries from several different address spaces, similar to a direct-mapped software TLB [2]. As long as there are enough domains for all threads wishing to execute, no flushes are required. Caches and TLBs must be flushed whenever a valid CPD entry is replaced.

However, the ARM only supports 16 domains, which is much less than the number of concurrently running tasks a general-purpose operating system is expected to support. We therefore need to recycle domains.

Domain recycling is required when a thread whose address space is not yet associated with a domain becomes runnable, and there are no free domains left. A domain must be recycled by preempting an address space. Before allocating a recycled domain to the new address space, the following must be done:

- TLBs must be flushed of all entries belonging to the recycled domain;
- the caches must be flushed of all lines corresponding to any of the just invalidated TLB entries;
- all of the domain's entries in the CPD must be invalidated.

This is obviously costly. A detailed analysis of cost will be required to determine whether the TLBs and caches should be flushed completely or selectively.

3.2. Domain preemption

In order to implement domain preemption we require a strategy for selecting a victim. There are two rather different scenarios in which domain preemption becomes necessary:

1. the number of available domains is greater than or equal to the working set of address spaces but less than the number of active address spaces. In this case preemption is an infrequent event triggered by changes in the address-space working set;

2. the number of available domains is less than the working set of address spaces. In this case we experience domain thrashing and preemption occurs frequently.

The third possibility, the number of available domains being greater than or equal to the number of active address spaces, is irrelevant as no domain preemption occurs in that regime.

If we assume that *address-space switching only occurs as a result of preemptive scheduling, and that address spaces are scheduled round-robin*,[1] it is clear that the optimal domain-preemption strategies are:

- *least-recently used* (LRU) for case 1 above, and

- *most-recently used* (MRU) for the case of domain thrashing.

To find the least or most recently used address space we would, in theory, keep time stamps on domains. Under the above assumptions we can then preempt optimally if we can determine in which regime (thrashing or non-thrashing) we are.

Note that in the non-thrashing case the preemption strategy is of relatively minor importance, as domain recycling is rare. If the "wrong" address space is preempted, this will lead to some extra preemptions, until the working set is established. In the thrashing case, the wrong preemption strategy will lead to frequent costly preemptions. In particular LRU is the worst possible strategy in the thrashing case, given the above assumption on how address spaces are scheduled.

However, the above scheduling assumption is a gross oversimplification, particularly for L4. L4 scheduling uses 256 hard priorities with threads being scheduled round-robin within each priority. If several of the runnable threads belong to the same address space, address spaces will not be scheduled round-robin, even if threads are. Furthermore, there are two circumstances where L4 dispatches threads without scheduler invocation:

- as is typical for microkernels, L4 relies heavily on message passing as the basic inter-process communication (IPC) mechanism. L4 IPC is blocking, and every successful IPC operation forces a context switch between the two communicating threads. The thread which was blocked waiting for the IPC to take place gets dispatched with the remainder of the partner's time slice;

- a thread can explicitly donate the remainder of its time slice to an arbitrary runnable thread. This feature is designed to be used for user-level scheduling. In an extreme case there would be a single highest-priority thread (the user-level scheduler) which schedules other threads by donating its time slices. That scheduler would be run once per time slice, about as often as all the other runnable threads combined.

Without prior knowledge of the communication and time-slice donation patterns of the client threads, no optimal strategy can be devised under these circumstances. The best that can be said is that an address space which has been used frequently (like the one of the scheduler thread, or a server which communicates with many clients) is most likely a bad choice for preemption, as it will most likely be used again in the immediate future. This argues in favour of a *least-frequently-used* (LFU) strategy. Note that least-recently used is no longer the worst possible algorithm in this case.

Note also that MRU is a dangerous algorithm to use, as it may fail to recycle domains which completely drop out of the working set, leading to thrashing of a small subset of domains.

What can be implemented efficiently? To answer that question it must be understood that two costs are involved:

1. the cost of the algorithm for selecting the victim domain. This cost is to be paid whenever domain preemption becomes necessary. As domain preemption is inherently expensive (due to the need to flush caches and TLBs) this cost is of minor importance;

2. the cost of maintaining the data structures required for victim selection. This cost must be paid at potentially each domain switch. As domain switches occur frequently, this cost is critical. In particular, a domain switch occurs on each cross-address-space IPC and we want to keep the critical IPC path as short as possible.

In keeping with the general L4 philosophy, the kernel should be optimised for the fast cases. In the StrongARM's target application areas of embedded and portable systems, 16 domains should in most cases be sufficient to hold the address-space working set. In this case the overhead for domain recycling should be as close to zero as possible. In

[1] The entity that gets scheduled is, of course, a thread, not an address-space. For simplicity we use the terminology of "scheduling an address space", which is understood to mean "scheduling a thread running in that address space."

particular, any slowdown of basic IPC operations should, as long as the number of domains suffices, be kept to a minimum.

3.3. Domain preemption strategies

The obvious approach to implementing a domain preemption algorithm is to use a reference counting scheme on address spaces associated with domains, and then use an LRU approximation algorithm such as the well-known clock scheme for selecting a victim domain.

The above requirement of minimising IPC overheads strongly impacts on the question whether it is reasonable to keep accurate reference counts. We will discuss the reference counting approach, as well as an alternative approach based on *approximate reference bit maintenance*.

3.3.1. Using proper reference counts. Maintenance of accurate reference counts requires that each domain be associated with a reference bit, and that this reference bit is set each time a switch to this domain is performed (that is, potentially at each IPC).

We keep the reference bit in the *task table*, an array of one or two 32-bit words for each of the 1024 tasks (address spaces) supported. That word has to be loaded on each address-space switch. Maintaining the reference bit requires two extra cycles: for setting the bit and for storing the word back.

The advantage of this approach is that, at no extra cost, the reference bit can be changed to a reference count. Up to 8 bits can be allocated to the reference count in the task table, and adding to that field (modulo 2^8) is not slower than setting it to one.

We then have the freedom to implement either a clock algorithm (which would only check the reference count for being non-zero), or an LFU algorithm, which appears particularly attractive in the L4 context. Which one is used in the end is a small implementation detail, and the final decision will be made once we can benchmark the system.

In either case we occasionally preempt a busy domain because its reference count has wrapped around to zero. This will only occur infrequently, so the cost of occasionally picking the wrong domain is irrelevant.

The advantage of accurate reference counts is that it gives us, at no extra cost, a good measure of whether or not we are in a domain-thrashing regime. The total number of address-space switches can be obtained by summing all domains' reference counts. Monitoring the number of address-space switches per domain preemption seems to be the best indicator of thrashing. That figure, normalised to the number of domains, would be one in the case of maximum thrashing (e.g. n domains, $n+1$ address spaces scheduled round-robin) and would grow very high if the working set fits into the available domains. This *thrashing criterion* would be very useful when taking special measures to reduce thrashing, such as using the PID register (see Section 3.4).

3.3.2. Maintaining approximate reference bits. There is an alternative way of approximately maintaining a reference bit which has zero overhead as long as the number of domains is sufficient, at the expense of increased domain-switching costs once preemption becomes necessary. The idea is based on the method used to implement page replacement algorithms on systems without a hardware-maintained reference bit in the TLB [7].

The method for maintaining the bit is tightly integrated with its use in the clock algorithm for selecting a victim domain. In addition to the reference bit we also store the *DACR value* in the task table. On each domain switch, the DACR is loaded from that task table entry.

When a domain is initially assigned to an address space, the reference bit is turned on and the DACR value is set to giving *client* rights to the domain.

When a domain is needed but no free one is available, the clock algorithm is applied to the reference bits in the usual fashion: All reference bits are examined round-robin, and if one is found unset, that domain is preempted. Whenever we encounter a reference bit which is on, we turn it off, and at the same time set the DACR value of the associated address space to *no access* (for all domains).

Turning off access rights for unreferenced domains will force a *domain fault* next time that domain's associated address space is activated. The domain fault handler can recognise this special case as the fault will occur on a CPD entry which is tagged with the correct domain (and should therefore allow access). The handler turns on the reference bit, changes the domain's DACR value back to the original *client* setting, and loads that value into the DACR.

This algorithm introduces domain management overheads only when domain preemption becomes necessary (and for a short time after, until all the domains in the working set have been accessed.)

3.3.3. Discussion. The latter approach saves two cycles on each address-space switch (including each IPC operation). The costs are:

- Increased frequency, and thus overhead, of domain fault handling. However, this overhead only occurs once domain preemption becomes necessary, and is small compared to the cost of preempting a domain.

- Less flexibility in the choice of preemption strategy, essentially only the basic clock algorithm is available. As there could be performance advantages from using LFU, this cost could be significant.

- No real reference counts, and therefore no accurate criterion for the onset of domain thrashing. (The best that can be done is to count the number of domain preemptions per unit of time, which is probably not unreasonable.) Whether or not this cost is relevant will depend on how effectively the PID register can be used to reduce domain thrashing, see next section.

In the end, the decision will have to be made on the basis of measured performance. We therefore plan to implement both variants so we can benchmark them.

In addition to either method, we can keep track on how many of an address space's PD entries are presently cached in the CPD for each domain. Before running one of the above algorithms we check whether there is any domain with currently no entries in the CPD. This would be the first candidate for preemption since recycling it would require no flushes.

3.4. Using the PID register

The above strategy will work on any ARM processor with a TLB. On the SA-1100 we can use the PID register to improve the performance of the above strategy. The PID register can either be used to effectively increase the number of available domains (at some not insignificant cost) or to reduce the conflicts in the CPD. Either method if successful would reduce the amount of flushing required.

3.4.1. Sharing a domain for small address spaces. The PID register allows transparent allocation of small address spaces at non-overlapping virtual memory regions. However, such a re-mapping only affects addresses issued in the lowest 32MB. Any larger address will be translated normally, without use of the PID register. This means that small address spaces cannot easily share a domain.

However, we can use the PID register to implement a less expensive version of domain preemption. We keep a *small bit* for each address space, indicating whether or not that address space has mappings above the 32MB limit. If we need to preempt a domain and there are at least two small address spaces currently associated with a domain, we get these address spaces to share a domain.

In order to share a domain between several small address spaces, we need to allocate a 32MB slot (and a corresponding PID) for each in the lower 2GB of the address space. This slot can, in principle, be shared with any of the large address spaces, as the domain tag ensures invalid data accesses or cache incoherence are prevented. However, this would mean that two (or more) address spaces compete for the same CPD entries, which would defeat the purpose of the exercise. Hence a slot should be chosen which is not presently in use by any active address space.

If a small address space, different from the last active small one, is to be scheduled, we have to invalidate all mappings left from the previous address space using the same domain. We do this by invalidating all CPD entries corresponding to the previous small address space's slot. We then flush both TLBs. Caches, however, do not need to be flushed, as the management of the domains ensures that no aliases exist.

Cleaning the data cache has been identified as being significantly more expensive than flushing the TLBs. Domain sharing of small address spaces should therefore incur less overhead than full domain preemption, although it is certainly not cheap. A thorough performance evaluation of the implementation will be required to determine whether it is worthwhile. In any case it should only be used if the system is definitively in domain thrashing mode.

Whether the system is thrashing domains can, in principle, be decided by counting the *number of address-space switches per domain preemption event*, Section 3.3.1 discussed how this figure can be determined if reference counts are being kept. Should the strategy presented in Section 3.3.2 be chosen in the end, domain thrashing must be identified (somewhat less reliably) by monitoring the domain preemption rate.

Note that this optimisation will only be beneficial if the operating system personality running on top of the microkernel cooperates by allocating user stacks within the first 32MB of virtual address space whenever possible.

3.4.2. Removing conflicts in the CPD. The PID remapping facility could alternately be used to minimise contention for the 32 CPD entries covering the first 32MB of the virtual address space. When replacing one of these first 32 CPD entries we first check if the new entry can be relocated (using the PID register) to a CPD entry in an empty 32MB slot.

By turning a CPD replacement into a PID register load we save flushing the TLBs and caches. The first 32MB of that address space is then relocated using the allocated PID until either the domain or the PID is recycled. While this method could be used on any user address space, the aliasing problem mentioned in Section 3.4 complicates matters. To keep the design simple we restrict the use of the PID to address spaces which do not contain any mappings in the relocation slots. This is a different style of *small address space* in which mappings are excluded from the region between the first 32MB and the upper 2GB. The operating system personality running on top of the microkernel can enhance the performance of the scheme by allocating user stacks at non-overlapping regions in the upper 2GB of the virtual address space.

3.4.3. Discussion. The above methods of utilising the PID register are both restricted to benefiting a form of small address space. Both also potentially complicate the original ARM generic design, which limits the potential benefits. Hence a full evaluation of both methods requires an implementation to explore their utility in a number of different application environments.

Both methods still require more thought before a detailed design can be given. The shared domain requires some method of reloading the CPD entries of an inactive small address spaces when a cache line of that address space is replaced. To further complicate matters, the effect of the PID register on abort exceptions is not sufficiently documented in [5] and its behaviour must be determined from experimentation. As a result, the work on PID optimisation is somewhat speculative at this stage.

4. Conclusions

The techniques presented in Section 3 are functionally transparent above the microkernel interface, the resulting reduction in context switching overhead will never lead to incorrect execution. However, microkernel clients can minimise overheads by reducing address-space conflicts as much as possible.

Reduction of overhead by smart address-space allocation is especially important for the StrongARM-specific optimisations for small address spaces (should they turn out to be beneficial). Standard address-space layout in Unix systems maps code and data segments at the low end of the address space, and the stack at the high end, thus making all address spaces "large".

To make efficient use of the hardware, an operating system personality running on top of the microkernel should allocate the stack at the top of the 32MB "small" address space whenever possible (for the approach discussed in Section 3.4.1) or allocate them at non-overlapping addresses in the upper 2GB of the address space (for the approach discussed in Section 3.4.2).

The algorithms discussed in Section 3 are geared specifically towards the L4 microkernel. However, they should be mostly applicable to other systems, such as monolithic Linux. In monolithic systems address-space switching behaviour typically be better approximated by round-robin. This should make an approximate LRU preemption algorithm, like clock, perform better in the non-thrashing case (where it is somewhat less important) but possibly perform worse in the thrashing case. Other algorithms could be implemented, although their cost would almost certainly be higher.

5. Future Work

An implementation of L4 on the StrongARM is presently underway, we plan to have a basic kernel running at about the time this paper goes to print. This will constitute a basis on which various embedded applications can be build. We also plan to port L^4Linux [3] and the Mungi [4] single-address-space operating system. These will provide multitasking environments in which the context switching performance of L4/StrongARM can be assessed.

A single-address-space operating system will provide a particularly interesting test of the kernel. On the one hand, the sparse address-space use of such a system makes it highly unlikely that the small address-space optimisation introduced in Section 3.4 would give any benefit. On the other hand a single address space does prevent aliasing by construction, and thus keeps threads' "address-spaces" disjoint, unless they explicitly share data. Sharing could be supported in the kernel by allocating extra domains for mappings of shared data. We plan to revisit this issue once we have gathered some experience with running a single-address-space system on the architecture.

References

[1] A. Au and G. Heiser. *L4 User Manual*. SCS&E, UNSW, Sydney 2052, Australia, Jan. 1998. UNSW-CSE-TR-9801. Available from http://www.cse.unsw.edu.au/∼disy/L4/.

[2] K. Bala, M. F. Kaashoek, and W. E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proc. 1st OSDI*, pages 243–253, Monterey, CA, USA, 1994. USENIX/ACM/IEEE.

[3] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proc. 16th SOSP*, pages 66–77, St. Malo, France, Oct. 1997.

[4] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, July 1998.

[5] Intel Corp. *SA-1100 Microprocessor Technical Reference Manual*, Sept. 1998. Order no: 278088-001.

[6] D. Jagger, editor. *Advanced RISC Machines Architecture Reference Manual*. Prentice Hall, July 1995.

[7] H. M. Levy and P. H. Lipman. Virtual memory management in the VAX/VMS operating system. *IEEE Computer*, 15(3):35–41, Mar. 1982.

[8] J. Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Technical Report 933, GMD SET-RS, Schloß Birlinghoven, 53754 Sankt Augustin, Germany, Nov. 1995.

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

[10] J. Liedtke. *L4 Reference Manual*. GMD/IBM, Sept. 1996. Available from URL http://www.inf.tu-dresden.de/-∼mh1/l3/.