Implementation of Fast Address-Space Switching and TLB Sharing on the StrongARM Processor

Adam Wiggins[1], Harvey Tuch[1], Volkmar Uhlig[2], and Gernot Heiser[1,3]

[1] University of New South Wales, Sydney 2052, Australia
[2] University of Karlsruhe, Germany
[3] National ICT Australia, Sydney, Australia
{awiggins,htuch,gernot}@cse.unsw.edu.au

Abstract. The StrongARM processor features virtually-addressed caches and a TLB without address-space tags. A naive implementation therefore requires flushing of all CPU caches and the TLB on each context switch, which is very costly. We present an implementation of fast context switches on the architecture in both Linux and the L4 microkernel. It is based on using domain tags as address-space identifiers and delaying cache flushes until a clash of mappings is detected. We observe a reduction of the context-switching overheads by about an order of magnitude compared to the naive scheme presently implemented in Linux.
We also implemented sharing of TLB entries for shared pages, a natural extension of the fast-context-switch approach. Even though the TLBs of the StrongARM are quite small and a potential bottleneck, we found that benefits from sharing TLB entries are generally marginal, and can only be expected to be significant under very restrictive conditions.

1 Introduction

A *context switch* occurs in a multi-tasking operating system (OS) whenever execution switches between different processes (i.e. threads of execution in different addressing/protection contexts). Such a switch requires the operating system to save the present execution context (processor registers) as well as the addressing context (virtual memory translation data, such as page tables). This generally requires a few dozen (or at worst a few hundred) instructions to be executed by the operating system, and the cost could therefore be of the order of 100 clock cycles [1].

Some architectures, however, make context switches inherently more expensive, as changing the addressing context can costs hundreds or thousands of cycles, even though it may take less than a dozen instructions. One such architecture is the ARM [2], which is especially targeted to embedded applications. High context switching costs are not an issue for many embedded systems, which may only consist of a single application with no protection between subsystems. Even if the system features several processes, context switching rates are often low enough so that the cost of individual context switches is not critical.

However, embedded systems are becoming increasingly networked, and increasingly have to execute downloaded code, which may not be trusted, or is possibly buggy and should not easily be able to crash the whole system. This leads to an increased use of protection contexts, and increases the relevance of context-switching costs.

Furthermore, there is a trend towards the use of microkernels, such as μITRON [3], L4 [4] or Symbian OS [5], as the lowest level of software in embedded systems. A microkernel provides a minimal hardware abstraction layer, upon which it is possible to implement highly modular/componentised systems that can be tailored to the specific environment, an important consideration for embedded systems which are often very limited in resources.

In a microkernel-based system, most system services are not provided by the kernel, and therefore cannot be obtained simply by the application performing an appropriate system call. Instead, the application sends a message to a *server* process which provides the service, and returns the result via another message back to the client. Hence, accessing system services requires the use of message-based inter-process communication (IPC). Each such IPC implies a context switch. Therefore, the performance of microkernel-based systems are extremely sensitive to context-switching overheads.

An approach for dramatically reducing context-switching overheads in L4 on the StrongARM processor has recently been proposed and analysed [6]. In this paper we present an implementation of this approach on two systems: Linux, a monolithic OS and L4KA::Pistachio [7], a new and portable implementation of the L4 microkernel under development at the University of Karlsruhe.

2 StrongARM Addressing and Caching

The ARM architecture features virtually-indexed and virtually-tagged L1 instruction and data caches. This ties cache contents to the present addressing context. Furthermore, and unlike most modern architectures, entries in the ARM's translation-lookaside buffer (TLB) are not tagged with an address-space identifier (ASID). As a consequence, multitasking systems like Linux flush the TLB and CPU caches on each context switch, an expensive operation. The direct cost for flushing the caches is 1,000–18,000 cycles. In addition there is the indirect cost of the new context starting off with cold caches, resulting in a number of cache misses ($\approx$ 70 cycles per line) and TLB misses ($\approx$ 45 cycles per entry). Worst case this can add up to around 75,000 cycles, or $\approx 350\mu$sec on a 200MHz processor.

These costs can be avoided by making systematic use of other MMU features provided: *domains* and *PID relocation* [6].

2.1 ARM Domains

The ARM architecture uses a two-level hardware-walked page table. Each TLB entry is tagged with a four-bit *domain ID*. A *domain access control register*

(DACR) modifies, for each of the 16 domains, the TLB-specified access rights for pages tagged with that domain. The domain register can specify that access is as specified in the TLB entry, that there is no access at all, or that the page is fully accessible (irrespective of the TLB protection bits).

Domains allow mappings of several processes to co-exist in the TLB (and data and instructions to co-exist in the caches[4]), provided that the mapped parts of the address spaces do not overlap. In order to achieve this, a domain is allocated to each process, and the domain identifier is essentially used as an ASID. On a context switch, instead of flushing TLBs and caches, the DACR is simply reloaded with a mask enabling the new process's domain and disabling all others. Shared pages can also be supported (provided they are mapped at the same address in all processes using them), by allocating one or more domains to shared pages.

Obviously, this scheme is much more restrictive than a classical ASID, due to the small number of domains. In a sense, however, it is also more powerful than an ASID: if TLB entries of shared pages are tagged with the same domain, the TLB entries themselves can be shared between the contexts, reducing the pressure on the relatively small number of TLB entries (32 data-TLB and 32 instruction-TLB entries). Such sharing of TLB entries is particularly attractive for shared libraries, which are very widely shared and tend to use up a fair number of instruction TLB entries.

2.2 StrongARM PID Relocation

The requirement of non-overlapping address spaces, while natural in a single-address-space operating system [8,9], is in practice difficult to meet in traditional systems such as Linux, and would severely limit the applicability of domains. At the least it requires using position-independent code not only for libraries but also for main program text, which implies a different approach to compiling and linking than in most other systems. For that reason, the StrongARM implementation [10] of ARM supports an address-space relocation mechanism that does not require position-independent code: Addresses less than 32MB can be transparently relocated into another 32MB partition. The actual partition, of which there are 64, is selected by the processor's *PID register*. This allows the operating system to allocate up to 64 processes at non-overlapping addresses, provided their text and data sections fit into 32MB of address space. Stacks and shared libraries can be allocated in the remaining 2GB of address space, although, in practice, the stack will also be allocated below 32MB.

2.3 Fast Context Switching on StrongARM

Using domains IDs as an ASID substitute and PID relocation of small address spaces, context switches can be performed without having to flush caches and

[4] The caches do not have any protection bits, so a TLB lookup is performed even on a cache hit, in order to validate access rights.

TLBs. The limited number of domains, however, imposes severe restrictions on this scheme (which is probably the reason Windows CE [11] does not use domains, and therefore does not provide protection between processes).

PID relocation, by itself, does not lead to sharing of TLB entries. In order to share TLB entries, memory must be shared (with unique addresses). This can be achieved by allocating shared regions (everything that is `mmap()`-ed, including shared library code) in a shared address-space region outside the 32MB area. In order to avoid collisions with the PID relocation slots, this shared area should be the upper 2GB of the address space.

The 32-bit address space is too small to prevent collisions outright. Hence, address space must be allocated so that collisions are minimised, but if they occur, protection and transparency are maintained (at the expense of performance). This can be done via an optimistic scheme that will try to allocate `mmap()`-ed memory without overlap as far as possible, uses domains to detect collisions, and only flushes caches if there is an actual collision [6].

The approach is based on making use of the ARM's two-level page table. Normally, the *translation table base register* is changed during a context switch to point to the new process's *page directory* (top-level page table). In order to detect address-space collisions, we never change that pointer. Instead we have it point to a data structure, called the *caching page directory* (CPD), which contains pointers to several processes' *leaf page tables* (LPTs). As page directory entries are also tagged with a region ID, it is possible to identify the process to which a particular LPT belongs.

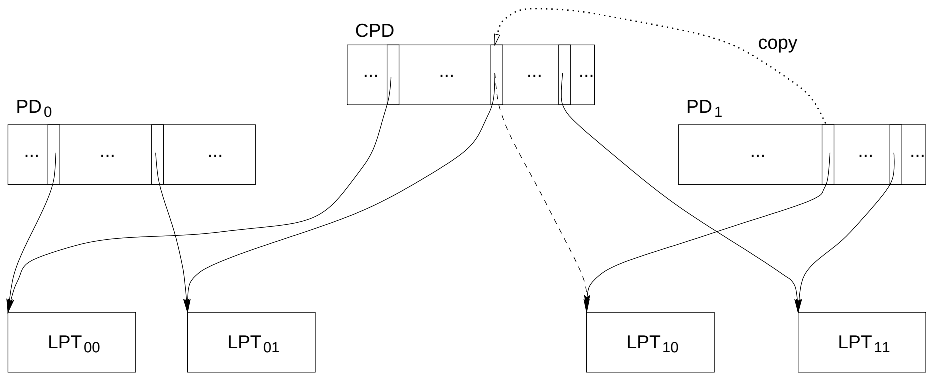

Fig. 1. Caching page directory (CPD) and per-address-space page tables

Fig. 1 illustrates this. The CPD is a cache of PD entries of various processes, tagged with their domain IDs. If two address spaces overlap, then after a context switch an access might be attempted which is mapped via a CPD entry that belongs to another process (as indicated by the domain tag). As the DACR only enables access to the present process' domain, such an access will trigger a fault.

The kernel handles this by flushing TLB and caches, reloading the CPD entry from the currently running process' PD, and restarting execution. Flushes will then only be required under one of the following circumstances:

1. a process maps anything (e.g. using the `MAP_FIXED` flag to the Linux `mmap()` system call) into the PID-relocation region between 32MB and 2GB, and the corresponding PID(s) are presently in use;
2. a process maps anything (using `MAP_FIXED`) into the shared region above 2GB and it collides with a mapping of another process;
3. there is no more space in the shared region for all mappings (which need to be aligned to 1MB);
4. the kernel is running out of domains to uniquely tag all processes and shared memory regions.

Given the small number of available domains (16), and the fact that the use of `MAP_FIXED` is discouraged, the last option is the most likely, except when very large processes are running.

3 Implementation

3.1 Fast Address-Space Switching

With the approach described in Section 2.3, the TLB, caches and page tables do not normally get touched on a context switch. All that needs to be done is to reload the DACR with a mask enabling access to the domain associated with the newly scheduled process, as well as any shared domains used by that process. The DACR value becomes part of the process context.

Flushes are required at context switch time if the new process does not have an associated domain, and no free domain is available, or when a shared mapping is touched that has no domain allocated. In that case, a domain ID must be preempted, and all CPD entries tagged with that domain need to be invalidated. With a total of 16 domains, and some domains being required for shared regions, this is not an infrequent event.

Cache and TLB flushes are also required if an address collision is detected. This is the case when a process accesses a page which is tagged in the CPD with a domain the process has no access to. In order to minimise the occurrence of collisions, the following steps are taken in Linux:

– The data and bss segments as well as the stack are allocated in the 32MB region. This does not limit the size of the heap, as `malloc()` will `mmap()` more space if `brk()` fails.
– `mmap()`-ed areas are, where possible, grouped (by process) into non-overlapping 1MB blocks allocated in the top 2GB of the address space.

In L4, all such mappings are under full control of the user-level code, hence no specific steps are required in the L4 kernel in order to minimise collisions.

Whenever caches are flushed, all domains are marked *clean*. A domain is marked dirty if a process with writable mappings in that domain is scheduled. Clean domains can be revoked without flushing caches.

3.2 TLB Sharing

On Linux we also implemented sharing of TLB entries for pages shared between processes. The implementation will transparently share TLB entries for memory shared via `mmap()`, provided the pages are mapped at the same address in both processes. The approach will not share TLB entries for program executables (as opposed to library code), as it is unlikely to produce any benefit on the StrongARM. In order to share the executable between processes, the two processes would have to be located in the same 32MB slot. This would cause maximum collisions on their stacks and data segments, unless those were staggered within the 32MB region. The programs could no longer be linked with their data segment at a fixed address (which introduces run-time overhead for addressing data). Furthermore, the need to share a single 32MB region between several processes would be too limiting to make the scheme worthwhile.

The present implementation allocates a separate domain for each shared region. The domain ID is kept in the VMA data structure linked to the in-memory inode.

Dynamically linked libraries account for a large amount of executed code in the system. In particular the standard C library consumes multiple megabytes of code and data. This code is relocated and linked at run time whereby code and data is relocated for the respective application. Shared libraries are commonly divided into one read-only part containing code and constant data, which is directly followed by a writable part.

In order to support sharing of library code, we modified the dynamic linker to separate the library's text and data segments. The data segments are allocated within the lower 32MB region, while the code is mapped into the shared area above 2GB.

Each library is given a preferred link address which is, at present, stored globally in the system. The first application using the library creates a copy of the binary image and, instead of performing a relocation of a privately mapped view, the copied image is relocated and saved. Afterwards, other processes can use the same image and map it into their address spaces. For security reasons we have two copies, one which can only be written with root privileges and a private per-user copy. When the pre-allocated link address is not available (e.g., it is already in use by another library), the linker falls back to the private linking scheme.

The separation of code and data has certain drawbacks. Relocation information is based on the fixed layout of the library in the address space. The ARM architecture has a very restrictive set of immediate operations. Hence, most immediate values are generated by storing offsets or absolute values interleaved with the code and using PC-relative addressing.

To separate code and data, and, in particular, share the code over multiple address spaces, we had to replace the mechanism to reference the global offset table (GOT) storing references to functions and global data. Instead of calculating the GOT address via a PC-relative constant (per function!), we divided the address space into slots of 1MB and maintain a table of GOT addresses for each slot of shared library code; larger libraries allocate multiple slots. At link time the code is rewritten from PC-relative references into PC-relative references within the GOT slot. ARM's complex addressing scheme allows inlining this address computation, which therefore results in no extra overhead. Finally, we eliminated the constant reference in the procedure linkage table (PLT).

TLB entry sharing was not implemented in L4.

4 Evaluation

4.1 Benchmarks

Linux

lmbench We use the following benchmarks from lmbench [12]: lat_ctx, hot_potato and proc_create. These are the subset of lmbench which can be expected to be sensitive to MMU performance.

lat_ctx measures the latency of context switches. It forks n processes, each of which touches k kilobytes of private data and then uses a pipe to pass a token round-robin to the next process.

hot_potato consists of two processes sending a token for and back. The *latency* test uses file locks, a FIFO, a pipe or UNIX sockets for synchronisation.

proc_create tests the latency of process creation. It consists of the following: the *fork test* tests the latency of `fork()` followed by an immediate `exit()` in the child. In the *exec test* the child performs an `exec()` to a program which immediately exits. The *shell test* times the latency of the `system()` service.

extreme We use a synthetic benchmark, which we call extreme, designed to establish the maximum performance gain from TLB entry sharing. It forks n child processes all running the same executable. Each child `mmap()`-s the same p pages, either private or shared. The child then executes a loop where it reads a byte from each page and then performs a `yield()`. The benchmark is designed to stress the data TLB.

When using private mappings the benchmark will not benefit from sharing TLB entries; it will thrash the TLBs as much as possible. With shared mappings it will share as many DTLB entries as possible (up to the lesser of p and the TLB capacity).

L4 The L4 benchmarks measure IPC times similarly to the Linux hot potato benchmark. A server process fires up a number of client processes, each of which IPCs back to the server (for synchronisation) and waits. The server process then IPCs random client processes, which immediately reply. The average latency of a large number (100,000) of such ping-pong IPCs measured. This benchmark concentrates on the property most critical to a microkernel-based system — the IPC cost. The benchmark is run for a varying number of client processes.

Table 1. Lmbench performance of original Linux vs. fast address-space switch ("fast") kernel. Numbers in parentheses indicate standard deviations of repeated runs. The last column shows the performance of the FASS kernel relative to the original kernel.

Benchmark	*original*	*fast*	*ratio*
lmbench hot potato latency [μs]			
`fcntl`	39 (50)	25 (3)	1.56
`fifo`	263 (1)	15.6 (0.1)	17
`pipe`	257 (3)	15.4 (0.1)	17
`unix`	511 (10)	30.7 (0.1)	16
lmbench hot potato bandwidth [MB/s]			
`pipe`	8.77 (0.02)	14.76 (0.03)	1.7
`unix`	12.31 (0.02)	12.94 (0.00)	1.05
lmbench process creation latency [μs]			
`fork`	4061 (4)	3650 (4)	1.1
`exec`	4321 (12)	3980 (10)	1.08
`shell`	54533 (40)	51726 (27)	1.05

4.2 Results

StrongARM results were taken from a system with 32MB of RAM, a 200MHz SA-1100 CPU and no FPU. The StrongARM has a 32-entry ITLB and a 32-entry DTLB, both fully associative. It has a 16kB instruction and an 8kB data cache, both fully virtual and 32-way associative, and no L2 cache.

lmbench hot potato and process creation results for Linux are shown in Table 1. Latencies of basic Linux IPC mechanisms (FIFOs, pipes and Unix sockets) in the original kernel are between 35% and 75% of the worst-case value of $2 \times 350\mu$sec quoted in Section 2. File locking is faster, as the test code touches very few pages and cache lines, which reduces the indirect costs of flushing.

The table shows the dramatic effect the fast-context-switching approach has on basic IPC times, with FIFO, pipe and socket latency reduced by more than an order of magnitude. Pipe bandwidth is also significantly improved (by 70%), while the improvement of the socket bandwidth is marginal. Even process creation latencies are improved by 5-10%. This is a result of lazy flushing of caches, which in many cases can defer the cleanup of a process's address space and cache contents until the next time caches are flushed anyway, hence reducing the number of flushes.

Fig. 2 shows lmbench context switch latencies. For the case of zero data accessed (i.e. pure IPC performance) the improvement due to fast context switching is dramatic, between almost two orders of magnitude (factor of 57) for two processes and a factor of four for 13 processes. After that, domain recycling kicks in (three domains are reserved for kernel use in ARM Linux), and the relative improvement is reduced. However, IPC costs in the fast kernel remain below 60% of the cost in the original kernel.

The runs where actual work is performed between the IPCs (in the form of accessing memory) show that the absolute difference between the IPC times

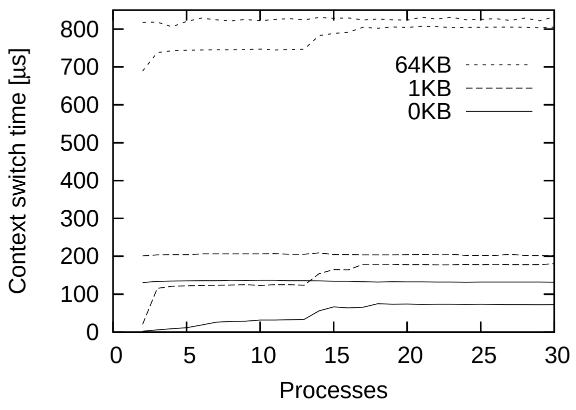

Fig. 2. Lmbench context switching latency (lat_ctx) as a function of the number of processes for different amount of memory accessed. Higher lines are for the original kernel, lower lines for the kernel with fast context switching.

remain similar, at least for smaller number of processes. This is a reflection of the actual IPC overhead being almost unaffected by the amount of memory accessed between IPCs.

Fig. 3 shows a magnified view of the zero-memory case of Fig. 2, with an additional set of data points corresponding to TLB entry sharing turned on. It is clear from this graph that the TLB is not a bottleneck in these benchmarks.

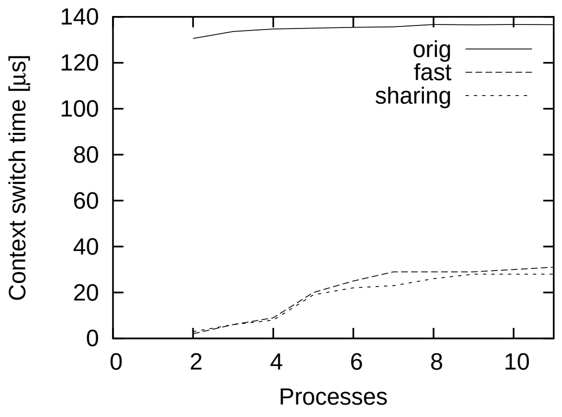

Fig. 3. Lmbench context switching latency over the number of processes with (zero accessed memory), comparing original kernel, fast kernel, and fast kernel with TLB entry sharing enabled.

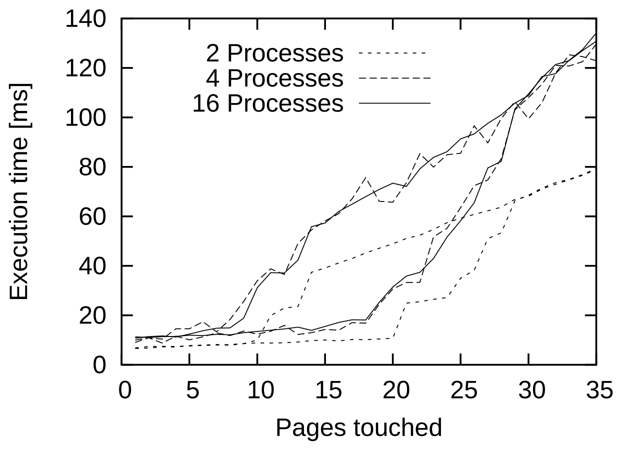

Fig. 4. Execution time over number of pages accessed for the extreme DTLB benchmark for 2, 4 and 16 processes. Higher lines are for the private mappings, lower lines for the shared mappings.

Consequently, the improvement of IPC performance is not dramatic, varying between zero and 23%.

We used the extreme benchmark in order to determine the best-case effect of TLB entry sharing. Results are shown in Fig. 4, which compares the fast kernel with private mappings (and hence no TLB entry sharing) and with shared mappings (and TLB entry sharing). With private mappings execution slows markedly as soon as more than about ten pages are touched. With shared mapping, performance remains essentially constant until about 20 pages are touched. Performance is the same once the number of pages reaches 32, which is the capacity of the ARM's data TLB. Execution times differ by factors of up to 9 (two processes) or 3.7 (16 processes). The effect is less pronounced with larger number of processes, as library code is not shared in this benchmark, and for larger process numbers the instruction TLB coverage is insufficient.

In practical cases the performance benefits from TLB entry sharing will be somewhere in between those of Figures 3 and 4, but most likely closer to the former. The reason is that the window for significant benefits from TLB entry sharing is small. The following conditions must hold:

– high context switching rates
– large amount of data shared between processes
– page working set no larger than the TLB size.

This combination is rare in present day applications. We made similar observations when examining the effect of TLB entry sharing on the Itanium architecture [13].

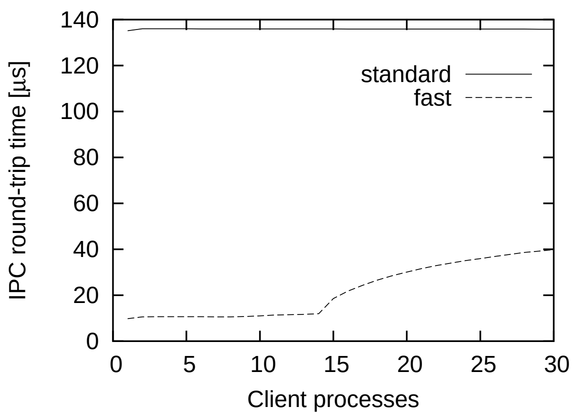

Fig. 5. L4 IPC times with standard and fast context switching implementation.

The effect of fast context switching in the L4 microkernel is shown in Fig. 5. In the standard kernel the cost of a round-trip IPC is around 135μsec (67.5μsec per context switch or about 1/5 of the worst-case figure of 350μs), quite independent of the number of processes. Fast context switching reduces the round-trip IPC cost to a minimum of 10μsec, a more than thirteen-fold improvement, rising slowly with the number of processes to 12μsec with 14 client processes (total of 15 processes), still a more than eleven-fold improvement. The increase of the IPC cost with larger number of processes is probably a result of increased competition for TLB entries and cache lines, an effect that is invisible in the standard kernel, as all caches are flushed on each context switch.

From 15 client processes the IPC cost rises faster. This is the point where the number of active processes (16) exceeds the number of domains available (15, as one is reserved for kernel use).

With a further increase of the number of processes, the IPC cost increase slows down. This may surprise at first, as the probability of the client thread having a domain allocated decreases. However, when a domain is recycled, all caches are flushed, making all allocated domains "clean" and therefore cheap to preempt. Hence, the direct and indirect cost of flushing caches is amortised over several IPCs. The IPC cost in the fast kernel stays well below that of the standard kernel.

These results have been obtained on a mostly unoptimised kernel. For example, 10μsec (2000 cycles) per round-trip IPC is actually very high for L4. Even with the generic IPC code in L4Ka::Pistachio we would expect to see a figure of less than 5μsec. We suspect that the kernel still has some performance bug, possibly an excessive cache footprint. Furthermore, coding the critical IPC path in assembler is known from other architectures to reduce the cost of simple IPC operations by another factor of 2–4. An optimised round-trip IPC should be around 2μsec. Such improvements would be essentially independent of the number of processes, and thus have the effect of shifting the lower line in Fig. 5 by a constant amount (significantly increasing the relative benefit of fast address-space switching).

5 Conclusion

Our results show that fast context switching, based on using domain IDs as address-space tags, is a clear winner on the StrongARM processor, in spite of the small number of available domains. We found no case where the overheads associated with maintaining domains outweighed their benefits. For basic IPC operations the gain was at least an order of magnitude, but even process creation times benefited.

There seems to be no reason not to use this approach in a system like Linux. In a microkernel, however, where the performance of systems built on top is critically dependent on the IPC costs, fast context switching is essential.

In contrast, the benefits of sharing TLB entries are marginal. It seems that this will only show significant benefits in a scenario characterised by high context-switching rates, significant sharing, and the TLB big enough to cover all pages if entries are shared, but too small of entries are not shared. The combination of high context-switching rates and intensive sharing of pages is rare in today's computer systems.

Acknowledgements

The Linux implementation of this work was carried out while Adam Wiggins was an intern at Delft University of Technology, Netherlands. We would like to thank all the members of the Delft's UbiCom project, in particular Jan-Derk Bakker, Koen Langendoen and Erik Mouw, for making this work possible.

Availability

Patches for fast-context switching support in Linux are available from http://www.cse.unsw.edu.au/~disy/Linux/, L4Ka::Pistachio for StrongARM, including support for fast-context switching, is available from http://l4ka.org.

References

1. Liedtke, J., Elphinstone, K., Schönberg, S., Härtig, H., Heiser, G., Islam, N., Jaeger, T.: Achieved IPC performance (still the foundation for extensibility). In: Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS), Cape Cod, MA, USA (1997) 28–31
2. Jagger, D., ed.: Advanced RISC Machines Architecture Reference Manual. Prentice Hall (1995)

3. ITRON Committee, TRON Association: μITRON4.0 Specification. (1999) http://www.ertl.jp/ITRON/SPEC/mitron4-e.html.
4. Liedtke, J.: On μ-kernel construction. In: Proceedings of the 15th ACM Symposium on OS Principles (SOSP), Copper Mountain, CO, USA (1995) 237–250
5. Mery, D.: Symbian OSversion 7.0 functional description. White paper, Symbian Ltd (2003) http://www.symbian.com/technology/whitepapers.html.
6. Wiggins, A., Heiser, G.: Fast address-space switching on the StrongARM SA-1100 processor. In: Proceedings of the 5th Australasian Computer Architecture Conference (ACAC), Canberra, Australia, IEEE CS Press (2000) 97–104
7. L4Ka Team: L4Ka — Pistachio kernel. http://l4ka.org/projects/pistachio/ (2003)
8. Chase, J.S., Levy, H.M., Feeley, M.J., Lazowska, E.D.: Sharing and protection in a single-address-space operating system. ACM Transactions on Computer Systems **12** (1994) 271–307
9. Heiser, G., Elphinstone, K., Vochteloo, J., Russell, S., Liedtke, J.: The Mungi single-address-space operating system. Software: Practice and Experience **28** (1998) 901–928
10. Intel Corp.: Intel StrongARM SA-1100 Microprocessor Developer's Manual. (1999)
11. Murray, J.: Inside Microsoft Windows CE. Microsoft Press (1998)
12. McVoy, L., Staelin, C.: lmbench: Portable tools for performance analysis. In: Proceedings of the 1996 USENIX Technical Conference, San Diego, CA, USA (2996)
13. Chapman, M., Wienand, I., Heiser, G.: Itanium page tables and TLB. Technical Report UNSW-CSE-TR-0307, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia (2003)