Performance of Address-Space Multiplexing on the Pentium

Volkmar Uhlig Uwe Dannowski Espen Skoglund Andreas Haeberlen
Gernot Heiser

Lehrstuhl Systemarchitektur
Universität Karlsruhe
contact@l4ka.org

Abstract

This paper presents an analysis of the performance potential and limitation of the so-called small-space scheme, where several logical address spaces are securely multiplexed onto a single hardware address space. This can be achieved on the IA-32 architecture by using the segment registers to relocate address spaces transparently to the applications.

Our results show that the scheme can provide significant performance improvements in cases where processes with small working sets interact frequently, as is often the case in client-server applications, and particularly in microkernel-based systems. We also investigate how potentially costly revocation of mappings can be prevented by clustering communicating processes.

1 Introduction

The gap between processor and memory speed continues to widen in modern architectures. As a result, the dependence of system performance on high hit rates in the CPU caches is increasing. Computer architects achieve these high hit rates by increasing cache capacity, and increasing the depth of the cache hierarchy.

A large cache implies that there is a significant probability of finding part of the cache still hot after a context switch, and thus a possibility of reducing the indirect context switch costs resulting from a cold cache. This potential always exists when switching between threads belonging to the same address space. It also exists when switching between processes, provided that the architecture supports secure sharing of caches between different address spaces.

Similarly critical for performance is the translation look-aside buffer (TLB), which is a cache for address translations. A TLB miss implies a cost of at least a few dozen cycles (assuming a hardware-loaded TLB and a cache hit on the page table). Indirect context switching costs can, again, be reduced if the TLB(s) can be shared across contexts.

Most modern architectures support sharing of CPU caches and TLBs between address spaces, usually by tagging TLB entries with an address-space identifier (ASID) and using physically-tagged caches. However, this is not the case on the IA-32 architecture, which presently dominates the PC and low-end server markets. On this architecture TLBs and virtually tagged CPU caches must be flushed on an address-space switch, an operation which itself is costly on some processors, but also implies significant indirect costs resulting from subsequent cache and TLB misses. This imposes performance limits on applications with high context switching rates, such as client-server type applications and microkernel-based systems.

Liedtke has shown how the Pentium's segment registers can be used to simulate ASIDs, and has presented results showing context switching costs can thus be reduced by 2–4 times under favorable conditions [7].

This paper presents a thorough investigation of

the performance potential of this *small address-space* approach under a wide range of application scenarios where benefits can be expected. These scenarios are generally characterized by high context-switching rates and moderate working sets. We also examine ways to reduce the potentially high revocation costs associated with the promotion of processes which have outgrown their allocated "small" space.

Section 2 explains address-space multiplexing in detail and examines the best-case benefits. Section 3 presents the setup for the experiments, which are presented, together with their results, in Section 4. Section 5 discusses limitations of the scheme and how they can be overcome. Related work is presented in Section 6 and conclusions in Section 7.

2 Address-space Multiplexing

Before describing the small address-space approach in detail we give an overview of the cache and memory management architecture of IA-32 processors. As there are significant differences between processor generations we concentrate on the latest one, the Pentium 4.

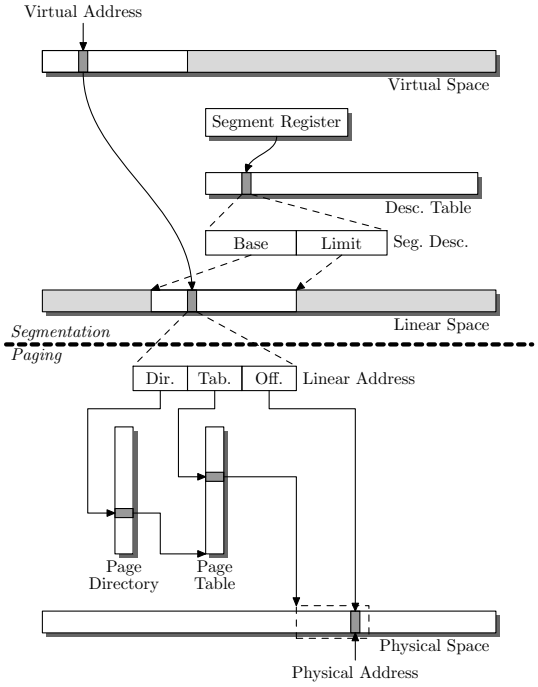


Figure 1: Segmentation and paging on IA-32.

2.1 Pentium 4 memory architecture

The Pentium 4 has an on-chip physically tagged L1 data cache (D-cache). Instead of an instruction cache (I-cache) it features a virtual *trace cache*, which stores instructions pre-translated into microcode for the Pentium 4 core. Because it is virtually tagged, the trace cache must be flushed on an address-space switch. All L2 caches are physically tagged and thus require no flushing.

The IA-32 virtual memory architecture uses a combination of segmentation and paging. A processor-issued 32-bit virtual address is first translated into a 32-bit linear address via a (usually implicitly specified) segment register. It is then translated into a physical address via a two-level page table, as shown in Figure 1. The page size is 4KB, but the page directory can map a whole 4MB super-page instead of pointing to a page table.[1] The hardware caches segment descriptor and page table lookups, the latter in separate instruction and data TLBs (called ITLB and DTLB from here on).

2.2 TLB tagging

Architectures featuring tagged TLBs essentially extend the virtual address by a per-process address-space tag, the ASID. The combination of ASID and virtual page number is what is being translated into a physical frame number on such processors. The basic idea behind the small-space approach is to reduce the application-usable part of the virtual address by a few bits, and use the segment registers to simulate an ASID in those bits, as shown in Figure 2. This is completely transparent to the application (except for the size of the usable address space.) By applying

[1] The architecture allows turning off page translation, in which case the linear address is the physical address. It also provides an alternative three-level page table format with 4KB and 2MB pages.

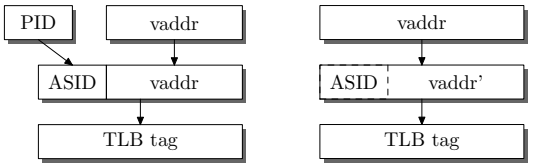


Figure 2: Tagging of TLB entries with (a) hardware supported and (b) emulated ASIDs.

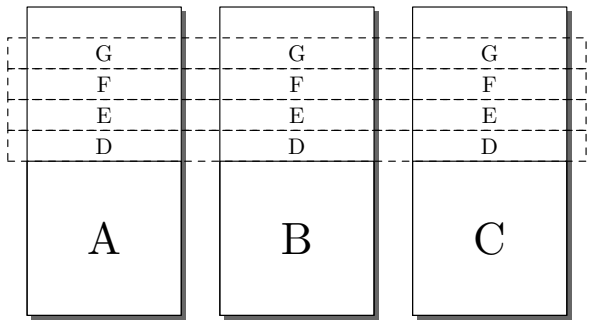


Figure 3: A system with 4 small and 3 large processes.

this trick to suitable address spaces only, we can ensure that processes requiring a full-size address space are still able to run.

Specifically, on the Pentium processor, the 4GB address space is split into two regions: a *large-space area* (of, say, 2GB), and a *small-space area*, which makes up the remainder of the hardware address space. The small-space area is itself split into a number of small-space regions, each of which can hold a small address space. A process which is to run in a small address space is assigned a small-space number, identifying which of the small-space regions it is to use. The segmentation registers are then used to relocate the process's address space into that region. A "large" process is not relocated and has access to the whole large-space area. The number of small processes is limited to the number of small-space regions.

At any time the address spaces of one large and all small processes are "visible" to the hardware, as shown in Figure 3. A TLB miss in a large process is, as usual, handled by the hardware walking the present page table. An initial miss in a small space generates a fault which is handled by the kernel by copying the appropriate page directory entry of the small process into the appropriate *relocated* entry of the current page table. In addition, a *global bit* is set on small-space TLB entries to prevent them from being flushed when switching between large processes.

As a result, any context switch between two small processes, or between a large and a small process, does not change the hardware addressing context, and does therefore not require any TLB or cache flushes. Flushing is only required when switching between large processes, and does not affect any TLB entries of small processes.

ITLB entries	128
ITLB repl.	31 cycles
DTLB entries	64
DTLB repl.	48 cycles
Associativity	Full, LRU repl.
TraceCache	12K, 8 way

Table 1: TLB configuration and replacement costs for a 1.4GHz Pentium 4 processor

2.3 Analysis of Costs

2.3.1 Savings per context switch

The potential savings resulting from the small-spaces scheme can be determined from the knowledge of characteristics of the architecture, as summarized in Table 1. The various contributions to context-switching costs are as follows.

Direct costs of flushing TLBs and trace cache, which can be fully eliminated for a suitable process switch. The savings amount to 480 cycles, the cost of executing a reload `CR3` instruction, which changes the page table and flushes the trace cache and the TLBs.

TLB replacement costs. These depend on how much of the TLB working set of the switched-to process can be kept alive during the execution of other processes. For a processor with fully associative TLBs, like the Pentium 4, this is the total TLB size, minus the combined TLB working sets of all processes which were running since the switched-to process was last executing. It is therefore very sensitive to the composition of the present process mix. The upper limit of this cost savings is given by TLB size times cost to reload a single entry, which is ∼4,000 cycles for the ITLB and ∼3,000 cycles for the DTLB on a 1.4GHz Pentium 4. This assumes that all page table lookups hit in the L2 cache, a reasonable assumption in a scenario where these savings are relatively high.

Trace-cache replacement costs. The cost of reloading the trace cache is difficult to measure, but we estimate it to be in the order of 5,000–10,000 cycles. Frequent branch mispredictions could add to that figure.

2.3.2 Context switch costs

The cost associated with the scheme is mostly that of reloading the segment registers, 253 cycles.

2.3.3 Overall savings

The maximum savings per context switch are not particularly high, hence the small-space scheme will only produce significant savings under conditions of high context-switching rates. These typically appear in client-server scenarios, particularly if the workload is reasonably evenly balanced between the client and the server. This includes scenarios where a client acts as a server for a third party, i.e., where several applications cooperate in a pipelined fashion.

Besides high context-switching rates, the ability to keep caches warm between context switches is critical to seeing a significant effect. This means that the total page working set of kernel and applications must be in the order of the TLB size. If the page working set is very small, the average saving per context switch is small. If it is too big, then too few hot entries remain in order to benefit.

Other than making qualitative statements of where effects are likely to be seen, it is difficult to estimate the amount of savings that can be expected in real-life scenarios. We therefore performed a quantitative study of potential interesting cases.

3 Experimental Test Bed

We used the L4 microkernel [6, 8] as the test bed for our experiments. One reason for our choice is that it is much easier to implement such a mechanism in a small kernel, compared to a large system like Linux. The experiments which required modifications to low-level components like page table handling and context switching, but also require API changes (to control the assignment of small-space regions to processes) were much easier to perform in L4.

A second, and equally important, reason is that microkernel-based systems are particularly sensitive to context-switching costs, as are any systems where components run as separate processes (so-called multi-server systems). We therefore ran most experiments using L^{4}Linux [4], which is a port of the Linux kernel to L4.

3.1 Address-space multiplexing in L4

The L4 version used in this work is L4Ka/Hazelnut, a portable re-implementation of Liedtke's original (100% assembler) kernel. L4Ka/Hazelnut is written in C/C++ with optional assembly implementation of some critical components, like delivery of IPC messages. In spite of being mostly written in a higher-level language, L4Ka/Hazelnut outperforms the original assembly kernel's IPC implementation on Pentium III and Pentium 4 systems.

The small-space mechanism was implemented in L4Ka/Hazelnut as follows. Each process, including "small" ones, is associated in the normal way with a hardware address space (which means that the process has its own page table). However, the kernel does not use this hardware address space when dispatching a thread in a small process. Instead, it modifies the segment descriptors for the address space so that all memory accesses go into the small-space re-

gion allocated to the small process. As described in Section 2.2, the process's address space is populated lazily by copying first-level page table entries from the process's own page table into the current page table (that of the present "large" process). If such a thread accesses memory beyond the size of the small-space region, it is automatically promoted to a large space of its own. Hence, small spaces are completely transparent (except for performance) to user-level applications.

The kernel can be configured for a small-space area of 0.5, 1 or 2GB, with the balance of the 4GB hardware address space available for large processes. The small-space regions can differ in size, from a minimum of between 4MB and 16MB (depending on the configured size of the small-space area). As the default link address for Linux application programs on IA-32 is `0x08048000`, or just above 128MB, we used a size of 256MB for all small spaces in our experiment, to avoid having to relink Linux applications.

3.2 L^{4}Linux

L^{4}Linux is a single-server system—the Linux "kernel" runs as a single user-level process on top of L4, possibly side-by-side with other microkernel applications, such as real-time components. It is binary compatible with the normal Linux kernel for IA-32 and can be used with any IA-32-based Linux distribution.

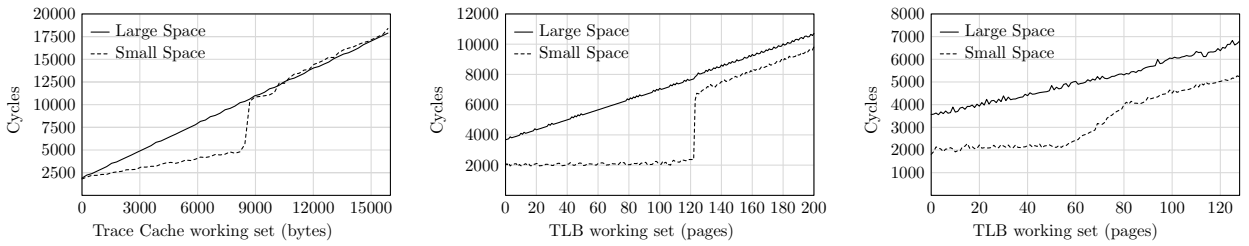
L^{4}Linux applications execute in their own separate address spaces on top of L4, and besides the L^{4}Linux server. Applications communicate with the server via a dedicated shared page mapped into the address space of applications and server. In order to achieve binary compatibility with native Linux, Linux system calls cause an exception, which is mirrored by the microkernel to an emulation library. The latter is some code mapped from the L^{4}Linux server into the application address space. The emulation library stores the system call parameters in the communication page and sends each Linux system call request via L4 IPC to the L^{4}Linux server. The L^{4}Linux server decodes the request, executes the Linux system call and stores the results in the shared communication page. Afterwards, it sends an IPC back to the application who reads the system call results into the respective registers and resumes execution after the system call instruction.

For every Linux system call two IPCs are required. Hence, cross-address space IPC performance is crucial for L^{4}Linux' overall performance. L4's IPC performance on IA-32 is dominated by hardware costs for switching to kernel mode and back. The actual IPC path in the kernel accounts for about 100 cycles only, whereas the two instructions for entering and leaving the kernel together add another 166 cycles. This is in addition to the context-switching costs discussed in Section 2.3.

Obviously, L^{4}Linux' performance suffers extremely from high context switch cost. A simple system call like `getpid`, which is one line of C code in the Linux kernel, causes the complete invalidation of DTLB, ITLB and trace cache on a Pentium 4 system. With small address spaces these costs can be reduced to two kernel entry and exits (compared to one on a monolithic Linux) and segment register reloads—basically the costs for a round-trip IPC on L4 which costs 550 cycles plus some additional costs for dispatching the system call in the L^{4}Linux server. The difference is around 1,700 cycles per Linux system call, depending on application size.

Considering the above mentioned costs, the most obvious candidate to put into a small space is the L^{4}Linux server itself. Since L^{4}Linux runs on global pages, and multiple address-space switches do not result in systematic TLB invalidations, the reduction of TLB misses can then be expected to approach that of a native Linux system,

There will still be a small amount of additional TLB misses, however: L4 uses one page for mapping thread control blocks for each thread (L^{4}Linux or application). Furthermore, one communication page per Linux application needs to be mapped in both, L^{4}Linux and the application, consuming another two pages per application. Hence, a client-server application scenario would consume 7 extra DTLB entries in L^{4}Linux compared to native Linux (9% of DTLB capacity). This, plus some extra processing cost for the IPC and slightly increased cache misses, are the costs resulting from running Linux at user level.



(a) Trace Cache (b) Instruction TLB (c) Data TLB

Figure 4: Execution time per IPC round trip (averaged over 100 executions) as a function of the trace cache and TLB working sets.

3.3 Linux applications

We also modified L^{4}Linux to support applications running in small spaces. This only required minor changes to the memory layout of Linux applications. More precisely, we had to do the following modifications:

1. We limited the application's (initial) virtual address-space size by allocating the stack at 256MB.

2. We changed the start mapping area for `mmap()` (used, e.g., for shared libraries) from 1GB to 167MB. Of course, this is rather limiting for large applications like databases, but the actual mapping address could easily be configured on a per-application basis. We never encountered a case where this was necessary, but then we don't tend to run large databases.

3. In L^{4}Linux systems the emulation library pages are located outside the valid application address space at address `0xA0000000`, in order to provide a usable application address space of 2.5GB. This needed to be relocated inside the small-space size. We chose the region `0x4000`–`0x6000`, which is not normally used by Linux applications. We also modified `mmap()` to deny explicit maps into that memory region.

Having implemented our changes, Linux applications were able to execute unmodified. The results obtained are described below.

4 Experiments and Results

4.1 Ping-Pong

In order to establish upper limits on the performance benefits of small spaces we used a benchmark with high context switching rates and adjustable working sets. A simple ping-pong application, normally used to benchmark IPC performance, systematically filled TLBs or the trace cache. Afterwards, it performed a ping-pong IPC by sending an empty message to a thread in a different address space and receiving an empty response. It then proceeded accessing the previously accessed cache entries. The partner thread did not perform any operation other than the reply.

The generated synthetic loads for the different caches were as follows:

Trace Cache: continuous execution of `nop` instructions (0x90). The instruction is 1 byte long.

I-TLB: relative jump by 4155 bytes. The jump instruction itself is 5 bytes long.

D-TLB: 4-byte read access at every 4160 bytes.

The reason for not using offsets of page granularities in the TLB measurements is to avoid evictions from the second level cache due to self interference.

The ping-pong results in Figure 4 show, as expected, that with growing trace cache working set the IPC cost grows much slower with small spaces, culminating in a two-fold difference in execution time at the point when the trace cache reaches its capacity (at 8K `nop`s, corresponding to the trace cache size of 12KB). Once the trace cache is exhausted, large and small spaces perform identically.

Even more dramatic than the trace cache working set measurements is the dependency on the ITLB working set size, with small spaces performing about 3.5 times faster than large spaces at the point were the TLB capacity is exhausted. In contrast to the trace cache case, however, the small-space scenario still has a performance advantage, even with large working sets. This results from the fact that the kernel IPC code in the trace cache survives throughout the execution of the smaller process.

The DTLB working set has a similar effect, although much less dramatic than in the case of the ITLB. Still, execution times are up to twice as fast with small spaces, and are significantly lower throughout all working set sizes.

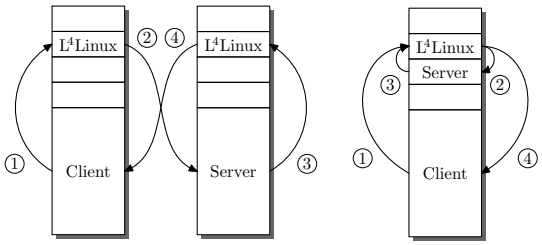
4.2 Linux vs. L⁴Linux

The performance of L⁴Linux has been thoroughly analyzed before [4]. The result was a very significant performance difference between Linux and L⁴Linux on micro-benchmarks (more than a factor of three in the case of `getpid()`) and a 5–10% slowdown on lmbench and hbench (on a 133MHz Pentium).

Basic system calls present a worst-case scenario for L⁴Linux, and a best-case scenario for the small-space approach. We measured the performance penalty of such system calls of L⁴Linux over native Linux, and the difference between the L⁴Linux server running in a large or a small space. As can be seen from the results in Table 2, the overhead of running Linux as a server is dramatically reduced by putting the server into a small space.

	Native	*L⁴Linux*	
System Call	*Linux*	*Large*	*Small*
`getpid` ()	1540 (0.7)	3837 (1.7)	2200
`gettimeofday` ()	1840 (0.7)	4295 (1.7)	2547
`read` (`4K`)	4241 (0.8)	7067 (1.4)	5198
`write` (`4K`)	5332 (0.8)	8746 (1.4)	6368

Table 2: Comparison cost in cycles (normalized to small space costs in parentheses) of selected Linux system calls in native Linux, with L⁴Linux executing in a small or large space (Pentium 4, 1.4GHz).



(a) Large spaces. (b) Small spaces.

Figure 5: Pipe based communication on an L⁴Linux system for applications running in large versus small spaces.

4.3 Client-Server Pipe

In a client-server environment on a monolithic operating system such as Linux, communication is either based on pipes, Unix sockets, or IP-based sockets. We repeated the ping-pong experiment on a Linux system, using a pipe to communicate between a client and a server task.

We then performed the same experiment on an L⁴Linux system with the L⁴Linux server running in a small space. We measured the pipe performance with both application processes running either in large or small address spaces. As indicated in Figure 5, this results in two hardware context switches per packet (label 2 and 4) in the former case, zero in the latter.

Figure 6 shows a comparison of the three setups

with different working set sizes and the corresponding communication costs in cycles. The results show the same behavior of large vs. small spaces as seen in Figure 4. Remarkably, the L^{4}Linux system in a small space outperforms native Linux on this test, as long as trace cache and ITLB capacity is not exceeded. This is a clear indication that native Linux also stands to benefit from small spaces.

4.4 cat | wc

As a typical pipe-based real life Unix application we piped a large file through `wc`, a utility which counts words of a text. The text itself was delivered from multiple chained `cat`s. We measured the cycles and number of TLB misses for a 10MB file for large and small space configurations.

The number of ITLB misses were reduced by 75% and the number of DTLB misses by 86%. The total execution time was 7.4% shorter with small spaces. Obviously, since `cat` and `wc` are tiny applications, most of the savings have to be accounted to the reduced number of DTLB misses.

4.5 X11 and `x11perf`

We ran `x11perf`, the performance test program for the X server. We used this application for two reasons: it stresses communication between two Linux tasks, itself and the X server, and it demonstrates the benefits of small spaces in a part of the system that we believe the user is very sensitive to—the GUI. `x11perf` exercises a set of drawing primitives of the X Windows System Protocol [10], such as dots, rectangles, circles, different line styles and fillings, text rendering, bitmap copies, etc., and measures the performance of these operations.

For the experiment, the L^{4}Linux-server and the X server each ran in a small space. Ignoring ordinary preemptive scheduling, no hardware address space switches were required.

We measured total execution times (in cycles) for the different `x11perf` operations. To our surprise the overall performance did only improve for a quite limited number of tested X functions, especially functions not accessing the video memory at all.

X operation	Large Space	Small Space	Improvement
Dot	202	200	1 %
QueryPointer	54499	48144	13 %
GetProperty	58911	53706	22 %
GetImage 10x10	149863	145258	3 %

Table 3: Comparison of cycles per object for multiple X operations tested by x11perf. For the large setup, only L^{4}Linux was running in a small space. In the small setup the X server was executing in a small space. Multiple objects may be drawn at a time.

We account our finding to the rather large TLB footprint of the X server, and to the bundling of multiple drawing functions in one X request. We conclude that the break-down on a per-object basis does not lead to a significant performance difference.

Table 3 gives the average cycles and ratio for the X server executing in a large versus a small space. In general we see a performance benefit below 1% for drawing functions.

4.6 Web Server + FastCGI

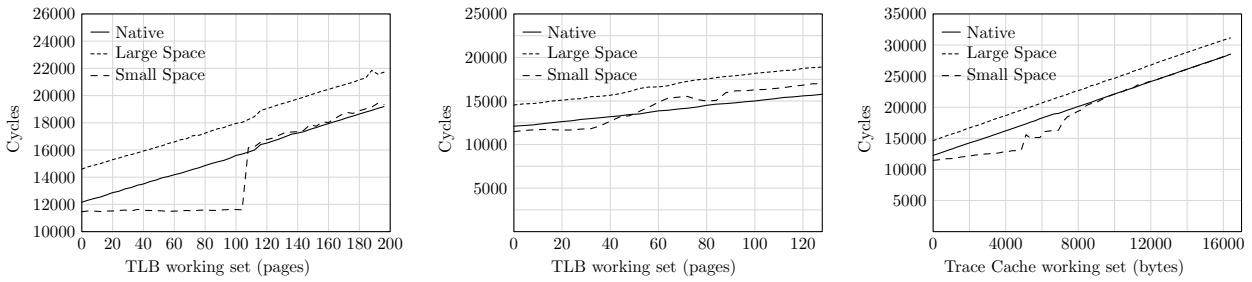
Web-serving with active content is a classical 3 tier application scenario. The remote client requests a web page with active content and the web server either responds out of the cache or forwards the request to another server, e.g., a database. If the web and database servers run on the same node, this results in local communication (e.g., through Unix sockets).

Depending on their working-set sizes, it may be beneficial to run the web-server and the database server in small address spaces.

For our analysis we configured Apache with FastCGI support. FastCGI [1] is a high-performance language-independent CGI extension based on Unix sockets for local and TCP/IP sockets for remote communication.

We implemented a small "Hello world" FastCGI server. The server was benchmarked running in small and large address spaces. All HTTP requests were received over the network from another machine.

Running on a standard Linux system our setup re-



(a) Instruction TLB (b) Data TLB (c) Trace Cache

Figure 6: Performance comparison of pipe-based client-server communication running on monolithic Linux and L^{4}Linux.

sults in two full context switches for each FastCGI request (Apache → FastCGI → Apache). Using small spaces we expected to see a measurable decrease in TLB and cache misses.

To handle one incoming HTTP request Apache and Linux generate about 290 DTLB and 250 ITLB replacements. This is obviously far beyond the capacity of the TLBs and we therefore did not see any performance improvement running the FastCGI server in a small address space. On the other hand, this result is indicative of significant performance problems of the Apache server itself.

In the next experiment we replaced Apache by a very small and efficient web server—thttpd.[2] thttpd is a single-threaded server supporting file serving and CGI. We extended thttpd with an experimental FastCGI interface and repeated our measurements.

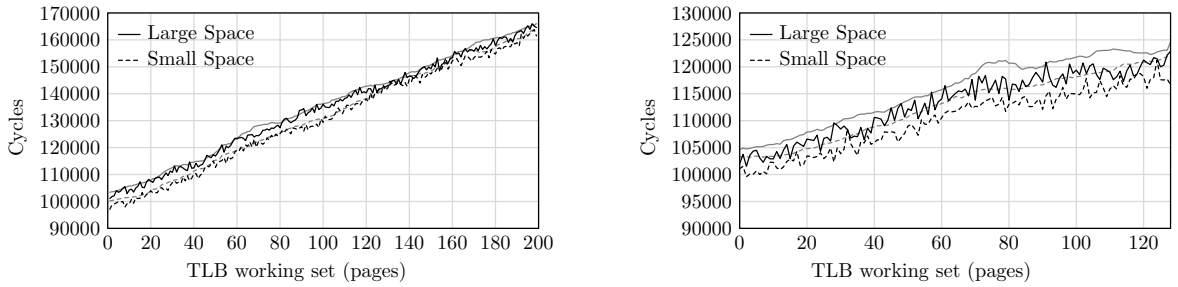
thttpd's working set is significantly smaller than Apache's, leading to expected performance improvement results. Our benchmark measures processing time from the point in time that an HTTP request is received by thttpd, until the full reply is generated and about to be sent back to the client. Figure 7 compares the processing times of HTTP requests with FastCGI running in small and large spaces with different working set sizes. On average we gain about 5% to 6% performance improvement running the FastCGI server in a small address space.

[2]According to netcraft.com, thttpd has the fifth-largest number of installations worldwide.

4.7 MySQL

Considering the memory requirements for large database servers, they do not sound like prime candidates for executing in small spaces. Nevertheless, large applications like databases can indeed benefit from small spaces if they are serving local clients which are thin enough to run in small spaces themselves (e.g., HTTP servers). Whether there are any benefits or not from using small spaces depends on the working set needed to handle database requests.

To evaluate potential performance gains in the realm of database servers, we measured the execution times for a number of small select-queries to a database served by MySQL [2], a popular, relatively light-weight, open source database server used in numerous heavily loaded web server setups around the globe. During our measurements we set up the MySQL server to run in a single-threaded mode. This was done to avoid the expensive operation of forking off a separate worker processes upon each database request, causing an invalidation of TLBs and virtually tagged caches. Performance did not suffer by running it in a single thread since the server in our experiment never needed to handle simultaneous re-



(a) Instruction TLB

(b) Data TLB

Figure 7: Processing time for thttpd and FastCGI request for different FastCGI server TLB working set sizes. The black lines show the minimum number of cycles seen (i.e., best case), while the gray lines show the average.

quests. Moreover, we believe that the server running on a more modern operating system would be better off distributing its load using cheap intra-address-space threads anyway.

Our experiments show that there is not much to gain from using MySQL in a small spaces environment. With very simple queries we gained about 3% performance increase by using small spaces. More complex queries were completely dominated by the execution time in the MySQL server, and had too large a working set to have any impact on the performance numbers. Other more light-weight database servers with a smaller working set for common queries might be able to benefit more from using small spaces.

4.8 LxDoom

As a representative for a whole class of applications—games—we chose LxDoom, a version of the 3D shoot'em'up game Doom originally released by iD software. As with all interactive setups, a higher update rate of displayed information is likely to result in increased acceptance by the user. As such, the frame rate with which LxDoom can render the player's view on the screen is a reasonable measure for application performance.

For our experiments we replayed a previously recorded session file (a five minutes stroll through the first level) in LxDoom's benchmark mode with sound output disabled. In this mode, the recorded session is rendered as fast as possible and the achieved frame rate is reported afterwards. Additionally, we counted the number of ITLB misses, DTLB misses, cycles the trace cache spent building micro-ops from instructions fetched, and cycles the trace cache spent to deliver micro-ops to the out-of-order execution core. Figure 8 shows the normalized numbers for executing the X server in a small space compared to a large space.

Running the X server in a small space, we see a dramatic 63% reduction of ITLB misses. Since the instructions executed remained the same, the number of cycles the trace cache spent delivering micro-ops to the core is inverse to the execution time. The reduction in the number of cycles the trace cache spent building micro-ops shows an increased hit rate in the trace cache, which in turn causes less ITLB misses. The number of DTLB misses, however, is reduced by as little as 2%. This confirms that the data working set of both LxDoom and the X server is far beyond the DTLB capacity.

We conclude that the small space approach has impact on the trace cache and ITLB only, but still re-

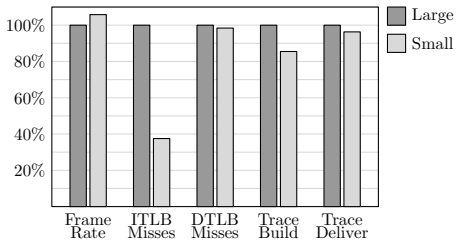


Figure 8: A comparison of LxDoom performance measurements when running the X server in either a large or a small space. Numbers are normalized to executing it in a large space.

sults in a 6% higher frame rate.

4.9 Make and Friends

Another prominent macro-benchmark is the compilation of programs, like the Linux kernel source tree. First, we let gcc compile a C++ source file of L4Ka/Hazelnut and measured execution time, DTLB misses, and ITLB misses in different system configurations. Second, we measured these values in the same configurations for a complete build of L4Ka/Hazelnut after a `make clean`. The system configurations were (a) Linux in a small space, (b) L^{4}Linux and all Linux tasks in small spaces, and (c) all Linux tasks in small spaces, but L^{4}Linux in a large space. The L4Ka/Hazelnut source tree consisted of 40 header files (161KB), 32 C++ files (513KB), and 5 assembler files (24KB). All files could be held in the file cache thus avoiding disk accesses.

The results were reproducible, but neither surprising nor satisfying: We could measure a reduction of both ITLB and DTLB misses by no more than 3% for configurations (b) and (c) compared to configuration (a). The execution time remained the same (compile: 0.52s, make: 7.3s). The working set of the measured jobs is simply too large (compile: 660K TLB misses, make: 9.4M TLB misses) to draw any benefits from the small space approach. The differences between configurations (b) and (c) were negligible.

4.10 SawMill

SawMill [3] is a multi server operating system running on top of L4. It is based on a Linux 2.2.5 kernel decomposed into separate orthogonal services. Services are protected from each other by running as servers within separate address spaces. As any multi-server operating system, SawMill extremely stresses IPC performance and context switching overhead by remotely invoking functions within other system servers.

We evaluated the performance impact of small spaces on SawMill by running a TCP/IP stack and a driver for a network interface card (NIC) in separate address spaces. A ping packet was then sent to the machine, and the round-trip time from when the interrupt arrived in the network driver until the ICMP reply packet was enqueued at the network card was measured.

Handling of a single ping packet requires: (a) a full context switch to the driver, (b) an intra-space context switch to the *bottom half* handler thread in the driver's address space, (c) a full context switch to the IP stack, and (d) a full context switch back to the driver. As such, three full context switches involving TLB and trace cache invalidation can be saved by using small spaces.

	Large Spaces	Small Spaces	Improvement
Cycles	38950	31800	18 %
DTLB Misses	46	5	89 %
ITLB Misses	44	0	100 %

Table 4: Execution times and TLB misses when handling a ping packet in the SawMill NIC driver and TCP/IP stack.

Table 4 summarizes our measurements. We see that 89% of the data TLB misses and 100% of the instruction TLB misses are saved by running the NIC driver and TCP/IP stack servers in small spaces. These savings, together with trace cache savings, results in an 18% performance improvement.

4.11 Summary

The measurements performed with L^{4}Linux running in a small space on top of L4 allow us to estimate the benefits of small spaces to a native Linux system without actually having to implement the scheme in Linux. As explained in Section 3.2, running the L^{4}Linux server in a small space, while not eliminating microkernel overhead completely, eliminates practically all address-space switching overheads resulting from running Linux at user level. Hence, the relative performance differences between applications running in small and in large address spaces, as summarize in Table 5, should be a good indication of the expected benefits of small spaces to a native Linux system. As the table shows, these can be anywhere between zero and 74%.

Application	Improvement
L^{4}Linux	40–70 %
Client-Server Pipe	19–74 %
`cat | wc`	7.6 %
X11 + x11perf	0–13 %
Apache + FastCGI	0 %
Thttpd + FastCGI	5–6 %
MySQL	0–3 %
Doom	6 %
Gcc + Make	0 %
SawMill	18 %

Table 5: Average execution time improvements of using small address spaces over large address spaces.

5 Limitations

As described in Section 3.1, small address spaces are implemented in a manner which makes them completely transparent to the applications using them. This was achieved by automatically transferring the small space into a larger one when the it accessed memory outside of its limited address space. However, if an application is to make use of a small space, the system must be aware of the fact that the application has only a limited virtual memory region to work within and take appropriate steps to make sure that it stays within its bounds. Most notably, stack space can no longer be located in the upper part of the virtual memory area (typically just below 3GB) and dynamically loaded libraries must be located so that they fit into the limited virtual space. Fortunately, these adjustments can be done transparently to the application by the dynamic linker and the operating system. The application binary will usually still be linked to some relatively high virtual address, though (typically just above 128MB and upwards), and relinking of the binary is therefore necessary if the application is to run in a truly small space.

5.1 Revocation

Another more pressing issue is the limited number of available ASIDs in the system, i.e., the virtual memory area dedicated to small spaces. Recalling that running unmodified Linux applications requires at least 256MB of virtual memory, and the 4GB address space must be shared between a large space and all small spaces, it is apparent that the number of ASIDs in the system is vastly insufficient. A mechanism for recycling ASIDs must as such be devised.

Unfortunately, recycling of ASIDs is extremely costly. It requires that one existing ASID must be preempted (i.e., revoked from the application using it). Revoking the ASID itself is not expensive at all, but the fact that page table entries might have been lazily copied to *any* hardware address space in the system implies that the affected page table entries in the small space area must be purged from every single existing page table. Considering the possible number of existing tasks (i.e., page tables), it is evident that such an operation must not be performed frequently.[3]

Given the tremendous cost of preempting ASIDs, a sensible ASID scheduling policy is paramount to the performance of the system. Finding such a policy is hard, though. Not only must the system spend un-

[3] Alternatively, lazy purging of page table entries is possible. Such a scheme would, however, dramatically increase the worst case switching time between threads, which would, in particular, result in unacceptably high worst-case interrupt latencies.

necessary cycles doing another preemption should the scheduling algorithm choose an inappropriate ASID for revocation, but the very nature of the applications running in small spaces tend to aggravate the cost of the preemption. That is, an application running in a small space must often respond quickly to some request or else one or more other applications in the system might suffer heavily from the increased latency. After all, performance was the reason for running the application in a small space in the first place. We have so far not been able to come up with a reasonable scheduling policy that decreases the ASID recycling frequency to a level suitable for deployment in a real system. As the following section describes, however, there exists other solutions to solving the ASID preemption problem.

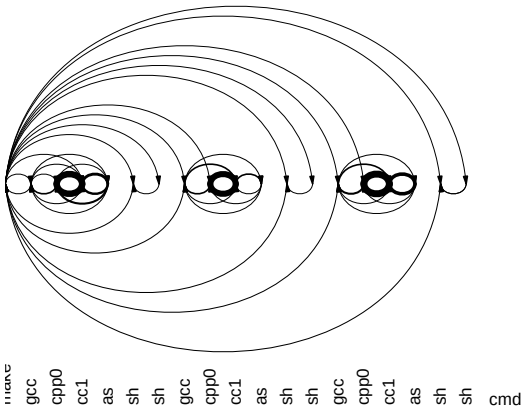


Figure 9: Compiling three source files in the Linux kernel tree. Switches always happens in clockwise fashion. The line width indicates the number of times a certain switch occurs (the width being a logarithmic function on the number of switches).

5.2 Application Groups

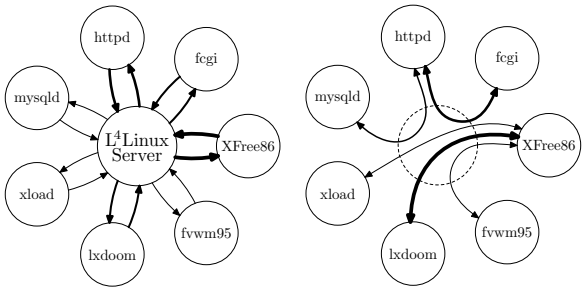
Observing that the number of ASIDs is too small and that ASID preemption is deemed too expensive to be performed frequently, one might conclude that ASID recycling is impractical, and only a limited number of small space tasks can therefore be supported. However, looking at the applications that benefit from executing in small spaces, i.e., client-server applications, one realizes that they tend to mostly communicate with a very limited group of other applications.

For example, as seen in Figure 9, the different pipeline stages of a compilation (`cpp`, `cc1`, and `as`) will only communicate with each other. Likewise, in Figure 10, a web server with dynamic content will only communicate with L^{4}Linux and certain other servers (e.g., FastCGI servers), and so on.

Moreover, the communication between the tasks within a group is usually such that one task issues a system call that releases another blocked task (e.g., sending a signal or providing some more data to an empty pipeline). As such, context switches will usually occur between different tasks within the group. Only when the operating system performs a scheduling decision, or when a task explicitly communicates with some task outside of the group—both infrequent operations—will some other task outside of the group be scheduled.

If applications are grouped together according to such relationships, only switches between groups require ASID preemption. Thus, ASIDs are no longer recycled one by one, but in groups. Such recycling of ASIDs, however, is cheap since the ASIDs within the group are restricted to one single hardware address space—there is no need to purge the ASIDs from all hardware spaces in the system. A simple switch of the hardware address space is sufficient.

Having the ability to group applications, a simple optimization can then be implemented by observing that some tasks (e.g., the L^{4}Linux server) communicate with all tasks in the system and therefore should be a member of *all* application groups. This can be implemented with the help of global pages, i.e., pages with TLB entries that are not affected by TLB flushes. By having these global small-space tasks operate on global pages, their TLB working set will not be flushed when switching between application groups. Figure 11 illustrates such a system with multiple groups and a single task which is global among all groups.



(a) (b)

Figure 10: In L^{4}Linux it seems that all tasks communicate with the Linux server only (a). As (b) shows, however, they actually communicate in small groups.

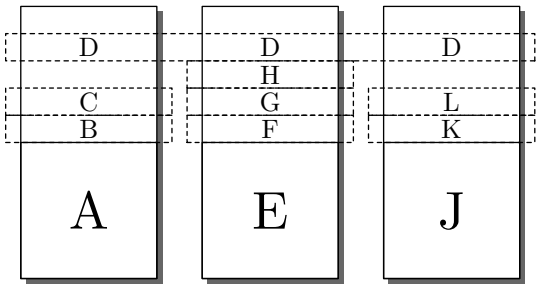


Figure 11: A system with three application groups $\{A, B, C\}$, $\{E, F, G, H\}$, and $\{J, K, L\}$ and one global small space task D.

6 Related Work

Small spaces were originally proposed by Liedtke, who also presented some indicative data on IPC performance in L4 with and without small spaces [7]. However, these results had been obtained from a kernel modified specifically for the benchmark presented in the report—to date no complete implementation of small spaces had been performed in L4. Small spaces have, apparently, been implemented in EROS [12], but no performance figures are available.

The basic idea of multiplexing a large hardware address space between smaller ones is not restricted to IA-32, not even to segmented architectures. However, some degree of hardware support is necessary to make it work securely. Such hardware support is available on the StrongARM processor [5]. It features a PID register, whose sole purpose it is to relocate a small address space transparently to the application. It is utilized in Windows CE [9], for example. Windows CE does not provide full protection of address spaces, though. The ARM *domains* [11] can be used to provide full protection for small spaces on the StrongARM, and can even be used for more general multiplexing of the hardware address space between processes [13]. No performance data are available at this time. However, given that the StrongARM has a virtual L1 D-cache, in addition to a virtual L1 I-cache and untagged TLBs, and its hardware support for small spaces is more direct, and less costly to use than on the Pentium 4, we can expect that the performance benefits of small spaces are higher on that architecture than on IA-32.

7 Conclusions

We have investigated the performance potential and limitations of the idea of securely and transparently multiplexing small logical address spaces within a single hardware address space. We implemented the scheme in the L4 microkernel and used L^{4}Linux and SawMill as the execution environment.

Our results show that the benefits of small spaces are most pronounced in microkernel-based systems, such as L^{4}Linux and SawMill, where performance can be improved by up to 70%. Cases where no significant performance improvements were evident could be explained with working sets exceeding TLB capacity. However, we found that client-server applications running on Linux, as long as they feature high context switching rates and small working sets, can also benefit, often in the order of 5–10%, in extreme cases up to 70%. We also found some cases where L^{4}Linux with small spaces outperforms native Linux. We conclude from these results that support of small spaces would be beneficial for native Linux too.

The main limitation of the scheme is the poten-

tially high revocation cost once processes outgrow their allocated small spaces. We have shown that this cost can be avoided by appropriate grouping of communicating processes. However, an implementation of such a scheme remains to be done.

Availability

The L4Ka/Hazelnut kernel, including support for small spaces, is available from `http://l4ka.org/`.

References

[1] Mark R. Brown. FastCGI: A High-Performance Gateway Interface. In *Programming the Web - a search for APIs*, 1996.

[2] Paul DuBois. *MySQL*. New Riders, December 1999.

[3] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The SawMill Multiserver Approach. In *9th SIGOPS European Workshop*, Kolding, Denmark, September 2000.

[4] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of microkernel-based systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, SaintMalo, France, October 1997.

[5] Intel Corporation. *Intel StrongARM SA-1110 Microprocessor Developer's Manual*.

[6] Jochen Liedtke. Improving IPC by kernel design. In *14th ACM Symposium on Operating System Principles (SOSP)*, Asheville, NC, December 1993.

[7] Jochen Liedtke. Improved Address-Space Switching on Pentium Processors by Transparently Multiplexing User Address Spaces. Technical report, GMD — German National Research Center for Information Technology, November 1995.

[8] Jochen Liedtke. On μ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, Copper Mountain Resort, CO, December 1995.

[9] John Murray. *Inside WindowsCE*. Microsoft Press, September 1998.

[10] Robert W. Scheifler. RFC 1013: X Window System Protocol, version 11: Alpha update April 1987, June 1987. Status: UNKNOWN.

[11] David Seal, editor. *ARM Architecture Reference manual*. Addison-Wesley, 2nd edition, 2001.

[12] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *17th ACM Symposium on Operating Systems Principles*, December 1999.

[13] Adam Wiggins and Gernot Heiser. Fast address-space switching on the StrongARM SA-1100 processor. In *5th Australasian Computer Architecture Conference*, pages 97–104, Canberra, Australia, January 2000. IEEE CS Press.