Itanium Page Tables and TLB

Matthew Chapman, Ian Wienand, Gernot Heiser

UNSW-CSE-TR-0307

May 2003

disy@cse.unsw.edu.au
http://www.cse.unsw.edu.au/~disy/
Operating Systems and Distributed Systems Group
School of Computer Science and Engineering
The University of New South Wales
UNSW Sydney 2052, Australia

Abstract

The Itanium architecture offers considerable flexibility in managing the TLB. Besides features found in many architectures, such as TLB tags and superpages, it supports two quite unusual features. One is the choice of two hardware-walked page table formats, a linear array and a hashed page table. The other is an unusual TLB tagging scheme which, among others, allows a single TLB entry to map a page to several address spaces, thus reducing the consumption of TLB entries in the presence of sharing.

Only one page table format, the linear array, is presently supported in Linux. However, this format neither supports the use of arbitrarily mixed page sizes nor the sharing of TLB entries. We have implemented the hashed page table format in Linux and found that this change has negligible performance impact, which should pave the way for exploring an implementation of superpage support. We have also implemented sharing of TLB entries, and found that in normal Linux workloads the effect is somewhere between negligible and a moderate performance increase. We could, however, demonstrate that there are scenarios where TLB sharing can produce significant performance gains.

1 Introduction

The Itanium architecture [Int00] provides an enormous amount of flexibility to the operating system (OS), by supporting a variety of optional or alternative mechanisms for controlling basic system functionality.

In this report we focus on Itanium mechanisms for controlling the memory management unit (MMU). Here, Itanium supports three different mechanisms for handling misses in the translation lookaside buffer (TLB): hardware reload from one of two architecture-defined page table formats, or software reload (from an arbitrary OS-defined format). The two hardware-defined formats are called the *short-* and *long-format virtual hashed page table* (VHPT) formats respectively.

In addition, the Itanium TLB contains a feature not commonly found in other architectures (although it is based on the scheme used in the PA-RISC [WS92] and is similar but far more general to what is used in the ARM [Jag95]): each TLB entry is tagged with a *protection key* (PK), which can be used to modify the access rights specified by the TLB entry. In particular, PKs support sharing of pages between processes with different access rights via shared TLB entries. This has the potential of reducing the consumption of TLB entries (or increase TLB coverage) in scenarios where sharing of pages is significant. TLB coverage has been identified as a potential bottleneck in system performance, with TLB miss handling overheads of 20–40% reported even on single-tasked benchmarks [CBJ92, HH93, Tal95, KS02] (although generally with software-loaded TLBs).

Linux presently only supports the short-format VHPT. This means it cannot support sharing of TLB entries, which is supported by the architecture only with the long VHPT format. The long format is also required to support mixing page sizes (so-called *superpages*), which has been shown to improve performance significantly [NIDC02]. However, these features would be of little use if their benefit was compensated by a performance degradation resulting from the use of the long format page table.

The purpose of this report is to investigate the cost of using the long VHPT format. We present an implementation of the long-format page table in the Linux 2.5.67 kernel, as well as performance measurements to asses their impact. We have also implemented sharing of TLB entries and have investigated the potential performance benefits.

2 Itanium MMU and Page Tables

2.1 Address translation

Figure 1 shows how address translation works on Itanium. The top three bits of the virtual address, the *virtual region number* (VRN), are used to index into an array of eight *region registers* (RRs), yielding a *region ID* (RID). The remaining bits, minus the page offset, form the *virtual page number* (VPN). The architecture supports eleven different page sizes ranging from 4KB to 4GB.

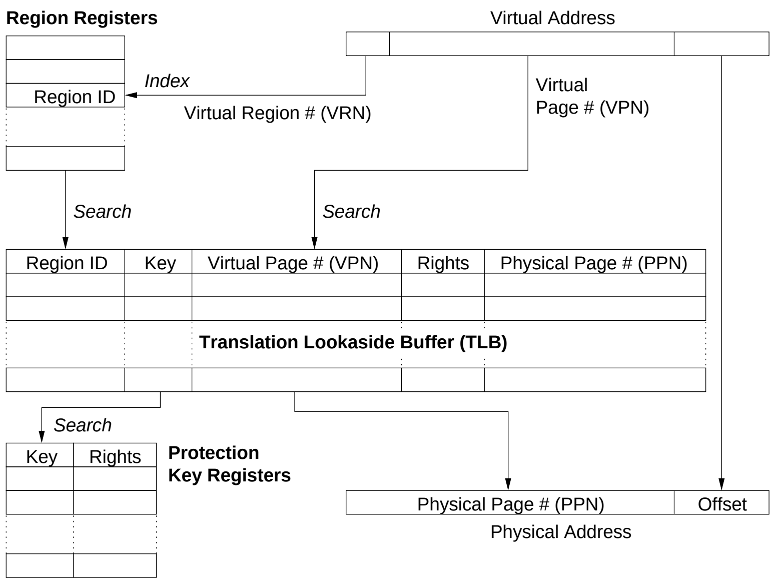

Figure 1: Itanium address translation

The VPN together with the RID is used as the key for an associative lookup of the TLB. If there is a match, the TLB entry will contain the correct address translation in the form of a *physical page number* (PPN), plus the access rights. If there is no match for the RID-VPN pair in the TLB, a TLB miss occurs, which results in a TLB reload by a hardware page table walker, if it is enabled. Otherwise an exception is taken (which would presumably result in a software TLB reload). An exception is also taken if the attempted access is not allowed by the rights field in the TLB entry.

The TLB entry contains a further tag, the *protection key*. The use of protection keys can be disabled, in which case this field is ignored. If protection keys are enabled, the protection key of the TLB entry containing the address translation is used in an associative lookup of an array of *protection key registers* (PKRs). If there is a match, then the PKR contains a further rights field. The memory access is only allowed to go ahead if it is permitted by the rights fields in both, the TLB entry and the PKR. The architecture specifies that at least 2^{18} different protection keys are supported.

2.2 Sharing TLB entries

2.2.1 Regions

Itanium is, at first glance, a segmented architecture: The VRN comparable to a segment number in a segmented architecture, like the PowerPC [MSSW94]. As such it supports sharing of TLB entries on a region basis: If a certain memory object (e.g. a memory-mapped file) is to be shared between processes, it can be allocated in a region of its own, and the kernel can associate a unique RID with the shared object. In order to share TLB entries (and not just pages) all processes sharing that object must agree on its address, including its RID.

The number of regions available to a process (eight) is too small to support a proper segmented view of the address space, where each object (file) is mapped into its own segment. One region needs to be reserved for program text, one for private data, at least one for kernel use, leaving at most five for libraries or other shared memory. Instead, it makes more sense to view the RIDs as a generalised *address-space ID* (ASID), as it is used to associate TLB entries with their processes on the MIPS [Hei93] or Alpha [Dig92]. The difference to normal ASIDs is that each process can use up to eight different tags. The present Linux kernel uses RIDs like ASIDs, by assigning a unique set of RIDs to each process.

In this approach it is straightforward to share TLB entries between processes executing the same program: one region is reserved for the text segment, and the kernel can associate a unique RID with each presently active executable. Process-private data is allocated in a separate region, and associated with a per-process RID.

The present Linux kernel does not share RIDs between processes.

2.2.2 Protection keys

Protection keys support sharing of TLB entries between processes, even if their access rights to a page differ. The rights field in the TLB entry gives the maximum rights any user has on the page (e.g. X/O for a code page, R/O for a data page subject to copy-on-write, R/W for other data pages). The PKRs can then be used to further restrict access on a per-process basis, according to each process's access rights. In order to use this feature, one or more regions are dedicated to potentially sharable pages. Every object that is `mmap()`-ed (including shared library code), unless it is mapped privately, is allocated in one of those regions. The kernel then needs to allocate a unique PK to each such object. When a process accesses the object for the first time, a PK fault will be triggered. The kernel handles these faults by loading the PK and the appropriate permissions into one of the PKRs.

As the PKRs are now part of the process context, they need to be flushed or reloaded on a context switch. Their number is small, 16 in the present Itanium processors, so this adds little overhead.[1]

The present Linux kernel does not enable protection keys.

2.3 Page tables

As indicated earlier, the Itanium architecture supports two different hardware-walked page table formats, the *short-format VHPT* and the *long-format VHPT*.

[1] An interesting issue arising from this is whether 16 PKRs is enough to make a reasonable object working set accessible.

2.3.1 Short VHPT

A short-format VHPT is, name notwithstanding, a *linear virtual array page table* [LL82,CE85] that maps a single region, hence up to eight are required per process. Each page table entry (PTE) is 8 bytes (one word) long. It contains a physical page number, access rights, caching attributes and software-maintained present, accessed and dirty bits, plus some more bits of information not relevant here. A RID need not be specified in the short VHPT, as it is common to all pages in the region for the presently executing process.

The page size is also not specified in the PTE, instead it is taken from the *preferred page size* field contained in the region register. This implies that when using the short VHPT, the page size is fixed for each region of each process (although different processes can use different page sizes, and a process can use different page sizes in different regions).

The PTE also contains no protection key, instead the architecture specifies that the protection key is taken from the corresponding region register (and is therefore the same as the RID, except that the two might be of different length). This makes it impossible to specify different protection keys in a region if the short VHPT is used. Hence, sharing TLB entries of selected (shared) pages within a region is not possible with this page table format.

2.3.2 Long VHPT

A long VHPT is a proper hashed page table. Its entries are 32 bytes (4 words) long and contain all the information of the short VHPT entries, plus a page size specification, a protection key, and a tag. Hence, the long VHPT supports a per-page specification of page size and protection key. The tag field is used to check for a match on a hashed access and must be generated by specific instructions.

2.3.3 Comparison

The advantage of the short VHPT is that its entries are compact and highly localised. Since the Itanium's L1 cache line size is 64 bytes, a cache line can hold 8 short entries, and as they form a linear array, the mappings for neighbouring pages have a high probability of lying in the same cache line. Hence, locality in the page working set translates into very high locality in the PTEs, and the number of data cache lines required for PTEs is small.

In contrast, a long VHPT entry is four times as big, and only two fit in a cache line. The probability of two PTEs sharing a cache line is. Hence, the long VHPT format is much less cache-friendly than the short format.

The situation is the opposite as far as the TLB is concerned. On the one hand, at least three TLB entries are generally required to map the page table working set of each process, one for code and data, one for shared libraries and one for the stack.[2] On the other hand, one large superpage mapping is enough to map the (much smaller) long VHPT of a process. Hence, the long-format VHPT is more TLB-friendly than the short-format VHPT.

This tradeoff is likely to favour the short-format VHPT in cases where TLB pressure is low, i.e. where the total page working set is smaller than the TLB capacity. This is typically the case where processes have mostly small working sets and context switching rates are low to moderate. Many systems are likely to operate in that regime, which is probably the reason why present Linux only supports the short VHPT format.

The most important aspect of the two page table formats is that the short format does not support many of the Itanium's MMU features, in particular TLB entry sharing and mixed page sizes (superpages). The latter have been shown to lead to significant performance improvements [NIDC02], and they ought to be utilised by operating systems running on Itanium. Before embarking on a superpage implementation, however, it makes sense to assess the potential performance degradation resulting from the long VHPT format.

3 Approach

In the following we describe our implementation of the long-format VHPT in the Linux 2.5.67 kernel. We also present the implementation of both forms of TLB-entry sharing discussed in Section 2.2: region-based sharing of TLB entries for text segments of executable programs, and protection-key-based sharing of TLB entries for `mmap()`-ed regions, including shared library code.

3.1 Long-format VHPT

The present Linux implementation simply maps the leaves of its multi-level pagetable into the virtual address space accessed by the hardware walker. This approach has the advantage that it is completely compatible with Linux's architecture-independent page table.

Replacing the native Linux pagetable with the long format VHPT would require extensive changes, since the multi-level pagetable paradigm is deeply entrenched. Instead we use the long format VHPT as a pagetable cache, or essentially another (software-managed) level of TLB [BKW94]. On a TLB reload miss, the software handler inserts the entry into both the long format VHPT and the TLB. A later TLB miss on the same entry is handled by the hardware walker reloading from the VHPT without invoking software. This approach has inherently more overheads, which could be eliminated by total re-implementation of the page table management in Linux. As such, the performance of our kernel will set a lower limit for what is achievable with long-format VHPTs.

[2] In fact, the present Linux implementation uses six regions for user code, and thus will require at least that many entries for mapping a single process' page tables.

In an SMP system the VHPT could either be global or per-CPU. A global VHPT has the advantage of allowing entries to be shared if the same address space runs on more than one CPU, but requires complicated locking on insertions. A per-CPU VHPT is more efficient for insertions but less efficient for purges, since more than one VHPT may need to be purged. Since insertions are more common than purges, we have initially decided to implement a per-CPU VHPT. To avoid touching all of the VHPTs on a purge, we keep track of which CPUs an address space has run on, and only purge the corresponding VHPTs.

An additional complication results from the hash function used by the long format VHPT implementation in Itanium CPUs, which is essentially the exclusive-OR of the RID and the VPN. Linux allocates context IDs sequentially, with a linear mapping from context ID to RID. There are frequently processes with close together RIDs and similar address space layout (e.g. after a fork()), which results in frequent hash collisions. As a quick and effective workaround, we switch the last two bytes of the context ID when constructing a region ID. This results in a wider distribution in the RIDs without wasting valuable RID space and without a complicated allocation scheme.

3.2 Region-based sharing of program text

Itanium Linux uses five regions, each with a different RID, for user processes (the remaining three are reserved for the kernel). One of these (VRN=2) is reserved for the text segment. We changed the RID allocation so that the RID for VRN 2 is no longer process-specific but program-specific. This is enough to ensure sharing of TLB entries for executable text segments.

In order to ensure that all processes running the same executable use the same RID, we store this RID value in the in-memory copy of the inode of the executable.

3.3 Protection-key-based sharing

To support sharing of TLB entries for `mmap()`-ed objects, we reserve one region (VRN=1) for shared mappings. Normally, all `mmap()`-ed memory (without `MAP_FIXED`) is allocated there, so we moved private mappings into region 4. Region 1 becomes essentially a single-address-space region: if two processes have access to the same address within that region, they will see the same data at that address. This region uses a fixed, global RID.

Memory management in the shared region is, at present, very simple: the region is filled from the bottom, and an address range, once allocated, is not freed (until reboot). While Itanium processors support 61-bit region offsets, we are presently limited to 40 bits because of Linux's three-level page tables.[3] This still supports 1TB of shared mappings, not a serious limitation (particularly as we could always fall back to the default way of handling `mmap()`). The protection keys are presently stored in virtual-memory-address data structures pointed to by the inode.

[3]This assumes a 16KB base page size, the default for Itanium Linux.

4 Evaluation

4.1 Test framework

In order to evaluate the performance of the two page table formats we used the standard lmbench [MS96] suite as well as Suite IX of the aim benchmark [SCO].

These standard benchmarks do not exhibit a significant amount of sharing (which is reasonable as far as the primary purpose of these benchmarks is concerned, as sharing is not prevalent in typical Unix environments). We therefore developed a benchmark which is specifically designed to evaluate the best-case performance benefits of sharing TLB entries. We call this benchmark "extreme", as it attempts to combine maximum sharing of text or data pages between processes with maximum stress on the TLB.

The extreme benchmark forks n child processes all running the same executable. Each child `mmap()`-s the same p pages. The child then executes a loop where it reads or executes a byte from each page and then performs a `yield()`. The benchmark can be configured to stress the instruction TLB (ITLB), data TLB (DTLB), or both.

With TLB sharing disabled, the extreme benchmark will thrash the TLBs as much as possible. With TLB sharing enabled it will share as many TLB entries as possible (up to the lesser of p and the TLB capacity).

All tests were run on a HP rx2600 server, which features dual 900MHz Itanium-2 CPUs. The processors have three levels of physically-addressed on-chip cache. The L1 is a split instruction and data cache, each 16KB, 4-way associative with a line size of 64 bytes and a one-cycle hit latency. The L2 is a unified 256KB 8-way associative cache with 128B lines and a 5 cycle hit latency. The L3 is 1.5MB large, 6-way associative, with a 128B line size and 12 cycles hit latency. The memory latency with the HP zx1 chipset is around 100 cycles.

The processors have separate fully associative data and instruction TLBs, each structured as two-level caches with 32 L1 and 128 L2 entries.

The per-CPU long-format VHPT was sized at 4MB.

4.2 Lmbench results

Table 1 shows the performance results for the process and file benchmarks of lmbench. There is very little performance difference between the kernels with short and long format VHPTs. In most cases the differences are statistically insignificant. In average, there might be at most about a 1% performance penalty from using the long format, which seems irrelevant.

The only significant performance impact (other than the one-off uniprocessor results for `fstat()` and `open()/close()`) is for the `mmap()` latency benchmarks. This is probably a result of how this particular benchmark operates, and the fact that in our implementation the long-format VHPT is not the page table proper, but only a cache of the Linux page table, plus the reduced cache locality of VHPT entries.

Processor, Processes

Kernel	Null call	null I/O	stat	fstat	open close	signal install	signal handle	process fork	process execve
Uni	1.00(0)	1.01(1)	1.00(0)	1.06(1)	0.93(1)	1.00(2)	1.02(1)	0.97(0)	0.98(2)
Smp	1.00(3)	1.01(1)	1.00(1)	1.04(2)	0.98(1)	1.01(2)	0.97(4)	0.97(0)	0.98(4)
Sharing	1.01(1)	0.99(1)	1.01(0)	1.03(2)	0.97(2)	1.01(1)	1.01(1)	1.02(0)	1.07(6)

File select

Kernel	10 fd	100 fd	250 fd	500 fd	10 tcp	100 tcp	250 tcp	500 tcp
Uni	1.03(1)	1.01(0)	1.00(0)	1.00(0)	1.01(0)	1.00(0)	1.00(0)	1.00(0)
Smp	0.99(1)	1.01(0)	1.00(1)	1.01(1)	0.99(1)	1.00(1)	1.00(0)	1.00(0)
Sharing	1.03(0)	1.00(0)	1.00(0)	1.00(0)	1.00(1)	1.00(0)	1.00(0)	1.00(0)

File create/delete and VM system Latencies

	0K		1K		4K		10K	
Kernel	Create	Delete	Create	Delete	Create	Delete	Create	Delete
Uni	1.00(0)	1.00(1)	1.00(8)	1.00(0)	1.02(5)	1.00(1)	1.02(10)	1.01(1)
Smp	1.00(0)	0.99(1)	1.00(2)	1.00(1)	0.99(4)	0.99(1)	0.99(9)	0.99(1)
Sharing	0.99(0)	1.00(1)	0.99(26)	1.00(0)	0.99(25)	1.00(0)	1.01(16)	1.00(1)

Kernel	Mmap Latency	Prot Fault	Page Fault
Uni	0.89(1)	0.88(30)	0.91(20)
Smp	0.90(1)	1.05(42)	0.93(16)
Share	0.92(1)	0.73(16)	1.00(0)

Table 1: *Lmbench process and file operation benchmarks. Numbers indicate performance relative to the vanilla kernel: a figure $>$ 1.0 indicates better, $<$ 1.0 worse performance than the vanilla kernel. "Uni" is the performance of the long format VHPT relative to the short format VHPT in a uniprocessor kernel, "Smp" is the same for a multiprocessor kernel (on a two-CPU system). "Share" is the same as "uni", except with TLB entry sharing enabled. The results are averages over five lmbench runs. Numbers in parentheses indicate the standard deviation in units of the last quoted digit. (E.g. "1.00(1)" means 1.00 with a standard deviation of 0.01, while "1.00(11)" means 1.00 with a standard deviation of 0.11. Background colours are used to highlight significantly improved or deteriorated performance.*

The benchmark maps a number of pages, accesses them sequentially, unmaps, and repeats. It is therefore dominated by page table operations and TLB refills which mostly result in software reloads.

The protection-fault benchmark is not particularly meaningful for this investigation, as it attempts to measure small timing differences (in the order of 300 cycles) which is very unreliable (as indicated by the large standard deviations).

Sharing (unsurprisingly) also makes little difference in most cases, however there is a tendency toward improved performance, in average more than compensating for the (small) performance penalty of the long format VHPT.

The situation is similar with the local communication performance runs shown

Local Communication latencies

Kernel	Pipe	AF/ Unix	UDP	RPC/ UDP	TCP TCP	RPC/ TCP	TCP- conn
Uni	1.01(4)	1.00(2)	1.00(1)	1.00(2)	0.99(1)	1.01(1)	1.01(1)
Smp	0.99(1)	1.00(4)	0.75(52)	0.96(0)	1.11(35)	0.98(1)	1.12(33)
Sharing	1.00(1)	0.97(2)	1.00(1)	0.99(2)	0.96(1)	0.98(1)	0.99(0)

Local Communication bandwidths

Kernel	Pipe	AF/ Unix	TCP	File reread	Mmap reread	Bcopy (libc)	Bcopy (hand)	Memory read	Memory write
Uni	1.07(16)	1.00(2)	1.00(7)	1.00(0)	1.00(0)	1.00(1)	1.00(0)	1.00(0)	1.00(0)
Smp	0.99(1)	0.99(2)	1.42(111)	1.00(0)	1.00(0)	1.00(1)	1.00(0)	1.00(0)	1.00(0)
Sharing	1.13(15)	0.99(2)	0.98(5)	1.00(0)	1.00(0)	1.00(0)	1.00(0)	1.00(1)	1.00(1)

More Local Communication bandwidths

Kernel	File open close	Mmap open close	Aligned Bcopy (libc)	Partial Bcopy (hand)	Partial Mmap read	Partial Mmap write	Partial Mmap rd/wrt	Bzero copy	HTTP
Uni	1.00(0)	0.99(0)	1.00(1)	0.99(0)	1.00(0)	1.00(1)	1.00(0)	1.00(0)	0.99(0)
Smp	1.00(0)	0.99(0)	1.00(1)	1.00(1)	1.00(0)	0.99(2)	1.00(0)	0.99(0)	0.98(2)
Sharing	1.00(0)	1.00(0)	1.00(0)	1.00(1)	1.00(0)	0.98(3)	1.00(0)	0.99(1)	1.00(2)

Table 2: *Lmbench local communication performance. See Table 1 for explanation.*

in Table 2, except there is even less of a performance difference.

The lmbench context switching performance results in Table 3 show again little effect from the page table format. Here, however, we find a significant effect from TLB entry sharing, with context switching performance improved by around 10–20% for large process numbers. This effect vanishes again as the process working sets get large, as then execution times are dominated by memory access times, which are independent of page tables and TLB sharing. The runs with sharing enabled are still faster in absolute terms (by about the same amount as in the smaller cases) but the relative time difference becomes too small to be relevant.

4.3 aim results

Table 4 shows the performance data for the aim benchmark. The picture is essentially the same as for lmbench: there is very little effect from either the page table format or TLB entry sharing.

The one exception is the fork test, which performs slightly worse with long page table format, and much worse with TLB entry sharing enabled. This is somewhat puzzling, as there seems to be no reason for this behavior, and it is inconsistent with the corresponding lmbench results. In fact, we took the corresponding code out of aim and ran it separately, with the following result:

Context switching with 0K							
Kernel	2proc	4proc	8proc	16proc	32proc	64proc	96proc
Uni	0.98(9)	1.00(4)	0.95(5)	0.88(7)	0.98(26)	1.44(6)	1.34(1)
Smp	0.94(17)	0.96(6)	0.95(6)	0.96(4)	1.23(22)	1.30(6)	1.27(4)
Sharing	0.99(9)	1.02(5)	1.02(5)	0.99(4)	1.01(21)	1.38(6)	1.34(2)

Context switching with 4K							
Kernel	2proc	4proc	8proc	16proc	32proc	64proc	96proc
Uni	0.97(10)	0.99(5)	0.97(5)	0.95(20)	1.17(17)	1.20(11)	1.09(4)
Smp	0.95(8)	0.61(82)	0.78(46)	0.87(28)	1.11(26)	1.13(6)	1.09(3)
Sharing	0.94(5)	0.96(5)	0.98(2)	0.87(9)	1.11(12)	1.13(7)	1.14(4)

Context switching with 8K							
Kernel	2proc	4proc	8proc	16proc	32proc	64proc	96proc
Uni	0.99(3)	0.98(7)	0.96(6)	0.97(15)	1.31(23)	1.17(15)	1.08(4)
Smp	0.95(3)	0.91(7)	0.96(6)	1.00(18)	1.29(25)	1.15(16)	1.06(6)
Sharing	0.97(5)	0.94(5)	0.96(4)	0.82(12)	0.91(50)	1.11(11)	1.12(7)

Context switching with 16K							
Kernel	2proc	4proc	8proc	16prc	32prc	64prc	96prc
Uni	0.99(3)	0.98(7)	0.96(6)	0.97(15)	1.31(23)	1.17(15)	1.08(4)
Smp	0.95(3)	0.91(7)	0.96(6)	1.00(18)	1.29(25)	1.15(16)	1.06(6)
Sharing	0.97(5)	0.94(5)	0.96(4)	0.82(12)	0.91(50)	1.11(11)	1.12(7)

Context switching with 32K							
Kernel	2proc	4proc	8proc	16prc	32prc	64prc	96prc
Uni	0.98(3)	0.99(4)	1.04(25)	1.30(69)	1.04(39)	1.03(4)	1.00(2)
Smp	0.94(3)	0.96(6)	1.00(10)	1.01(28)	0.87(44)	1.00(4)	1.00(1)
Sharing	0.96(2)	1.01(3)	1.01(3)	1.17(15)	1.04(18)	1.05(5)	1.02(2)

Context switching with 64K							
Kernel	2proc	4proc	8proc	16prc	32prc	64prc	96prc
Uni	1.00(4)	0.98(2)	0.94(8)	0.94(38)	1.00(8)	1.00(0)	1.00(0)
Smp	0.97(3)	0.98(1)	1.06(25)	1.22(57)	0.94(10)	0.99(2)	0.98(2)
Sharing	0.99(3)	1.00(2)	0.95(8)	1.69(56)	0.98(11)	1.00(1)	1.00(0)

Table 3: Lmbench *context switching performance. See* Table 1 *for explanation.*

Test	**Uni**	**Smp**	**Share**
fork_test	0.97(1)	0.95(1)	1.15(1)

This is much more in line with expectations. TLB entry sharing helps with `fork()`, as the shared mappings are not flushed from the TLB. We suspect that the fork results in the vanilla aim suite suffer from some unlucky cache conflicts.

4.4 Extreme results

Figure 2 shows results of the extreme benchmark, set up to stress the data TLB. The long-format VHPT brings some small performance improvement. This is a

Test	Uni	Smp	Share	Test	Uni	Smp	Share
add_double	1.00(0)	1.00(0)	1.00(0)	fun_cal15	1.00(0)	1.00(0)	1.00(0)
add_fbat	1.00(0)	1.00(0)	1.00(0)	sieve	0.99(4)	0.98(24)	1.10(9)
add_long	1.00(0)	1.00(0)	1.00(0)	mul_double	1.00(0)	1.00(0)	1.00(0)
add_int	1.00(0)	1.00(0)	1.00(0)	mul_fbat	1.00(0)	1.00(0)	1.00(0)
add_short	1.00(0)	1.00(0)	1.00(0)	mul_long	1.00(0)	1.00(0)	1.00(0)
creat-clo	1.00(1)	0.99(1)	1.00(0)	mul_int	1.00(0)	1.00(0)	1.00(0)
page_test	0.98(0)	0.98(1)	0.98(0)	mul_short	1.00(0)	1.00(0)	1.00(0)
brk_test	0.99(0)	1.00(1)	0.99(1)	num_rtns_1	1.00(1)	1.00(0)	0.99(1)
jmp_test	1.00(0)	1.00(0)	1.00(0)	new_raph	1.00(0)	1.00(0)	1.00(0)
signal_test	1.00(1)	0.99(0)	1.02(1)	trig_rtns	1.00(1)	1.01(0)	1.00(0)
exec_test	0.97(1)	0.98(0)	1.02(1)	matrix_rtns	1.00(0)	1.01(0)	1.02(0)
fork_test	0.98(2)	0.95(1)	0.88(3)	array_rtns	1.00(0)	1.00(0)	1.00(0)
link_test	1.00(0)	0.99(0)	1.01(1)	string_rtns	1.00(0)	1.00(0)	1.00(0)
disk_rr	1.02(3)	1.00(2)	0.99(4)	mem_rtns_1	1.00(0)	1.00(0)	1.01(6)
disk_rw	1.01(1)	1.01(3)	0.99(2)	mem_rtns_2	1.00(0)	1.00(0)	1.00(0)
disk_rd	1.04(0)	1.02(1)	1.02(1)	sort_rtns_1	1.00(0)	1.00(0)	1.00(0)
disk_wrt	1.02(2)	1.00(2)	1.01(2)	misc_rtns_1	0.99(2)	1.00(1)	1.01(1)
disk_cp	1.01(1)	1.00(3)	1.00(2)	dir_rtns_1	0.99(0)	1.00(0)	1.00(1)
disk_src	1.00(0)	0.99(1)	1.00(1)	series_1	1.02(1)	0.99(1)	1.00(1)
div_double	1.00(0)	1.00(0)	1.00(0)	shared_memory	1.01(0)	0.99(0)	0.98(5)
div_fbat	1.00(0)	1.00(0)	1.00(0)	tcp_test	1.00(0)	0.99(1)	0.99(0)
div_long	1.00(0)	1.00(0)	1.00(0)	udp_test	1.01(1)	0.99(1)	0.99(0)
div_int	1.00(0)	1.00(0)	1.00(0)	fi fo_test	1.04(0)	1.00(1)	1.05(1)
div_short	1.00(0)	1.00(0)	1.00(0)	stream_pipe	1.01(4)	0.97(4)	0.99(1)
fun_cal	1.00(0)	1.00(0)	1.00(0)	dgram_pipe	1.01(2)	0.99(4)	1.01(1)
fun_cal1	1.00(0)	1.00(0)	1.00(0)	pipe_cpy	1.10(0)	1.03(1)	1.09(1)
fun_cal2	1.00(0)	1.00(0)	1.00(0)	ram_copy	1.00(0)	1.00(0)	1.00(0)

Table 4: *aim benchmark performance. See Table 1 for explanation.*

result of the long format being more TLB friendly, and therefore compete less for TLB entries with the program data. TLB entry sharing reduces the overall execution time by about a factor of two, until the working set outgrows the TLB, when the benefit reduces sharply by about half and then continues degrading more gracefully. Other runs show that the total benefit is only very weakly dependent on the number of processes. This benchmark shows that the effect of TLB entry sharing can be significant, albeit in very special circumstances.

5 Related Work

Itanium is, to our knowledge, the first architecture supporting two different hardware-walked page table formats. Our study seems to be the first comparing the performance of the two formats. Past studies have compared the performance of different page tables on architectures with software-loaded TLBs [UNS+94, EHL99, DMY99], or used trace-driven analysis of changing the page table format in an otherwise unmodified architecture [HH93, THK95, KJW94, KT95].

We believe that our evaluation of TLB entry sharing, together with a similar

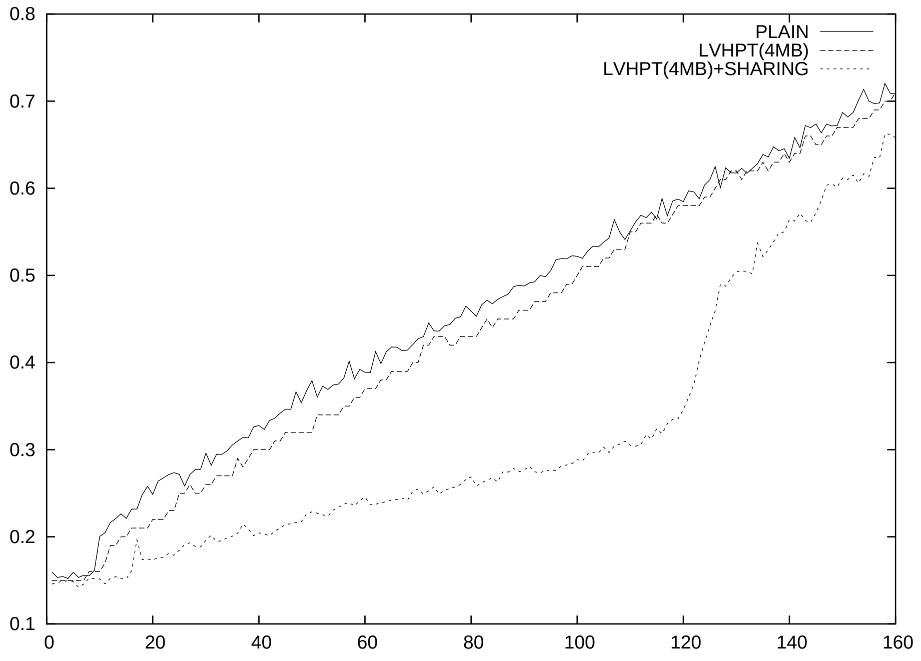

Figure 2: Execution time for the extreme DTLB benchmark for 10 processes. "Plain" refers to the standard kernel (short VHPT), "LVHPT(4MB)" uses the long-format VHPT (without shared text), and "LVHPT(4MB)+SHARING" enables full sharing (with long VHPT).

study we performed on the StrongARM [WWTH03], is the first if its kind. While there have been systems which share TLB entries, we could not find any publication quantifying its effects.

Sharing of TLB entries is rare in present operating systems and mostly restricted to the few using a segmented memory model. Windows CE [Mur98] on the StrongARM allocates dynamic libraries at fixed addresses and thereby shares their TLB entries across processes. CE, however, is aimed at embedded systems and so does not provide protection between processes, making this approach unsuitable for general-purpose operating systems. Even in embedded systems, where all programs can be considered "trusted", address-space-based protection is valuable, as it eases debugging. Furthermore, CE works by only supporting a limited number of processes (32), which is unacceptable for a general-purpose system. Hence, what can be learned from the CE approach is quite limited.

A number of alternate hardware approaches have been proposed. Koldinger et al. [KCE92] proposed a *protection lookaside buffer* (PLB) as a means to separate protection and translation, and thus better utilise the valuable TLB real estate. Their proposal was specifically aimed at single-address-space systems [CLFL94, HEV+98], which are characterised by a process-independent address mapping. However, Wilkes and Sears compared the PLB proposal with the MMU features of the PA-RISC and found that the latter provided a more effective solution [WS92]. They went on to propose a number of extensions to the PA-RISC scheme, some of which are implemented in Itanium. Recently, the PLB proposal was further developed into a model called *Mondrian memory protection* [WCA02]. This is an interesting approach that has the potential to improve MMU and cache design of future processors, but has yet to be implemented in actual hardware.

Khalidi and Talluri proposed a different hardware solution, where the normal *address-space ID* (ASID) tag in the TLB is supplemented with a mask selecting one of several shared regions [KT95]. TLB lookups can match on either ASID or region. The scheme is specifically targeted at sharing mappings for a relatively small number of heavily shared regions, especially shared library code. Their simulation-based study shows reductions of TLB miss rates by between zero and 64% from running multiple copies of the same program, but they do not present data on how this translates into actual performance. Their proposal is less general than the Itanium approach which supports both a huge number of potentially shared regions and sharing with different access rights.

Many architectures, such as the MIPS [KH92], support a *global bit* in a TLB entry, which makes a page accessible in all address spaces. This is too crude to be of much use other than for kernel pages, which on the MIPS can be put into an address space region that is inaccessible from user mode, and thus can be made visible independent of the address space as soon as the processor enters kernel mode.

Liedtke proposed a scheme that supported efficient handling of shared pages, even with different virtual page numbers [Lie93].

Several architectures with hardware-loaded TLBs, while providing support for sharing page tables [SPA91, Int01], do not support sharing of TLB entries.

6 Conclusions

The primary purpose of this report was to assess the performance impact of moving from the short to the long VHPT format on Itanium. While our implementation in Linux was minimal in the sense that it avoided a massive rewrite of page table handling, at the expense increased run-time overheads, we found that the performance differences between the two page table implementations were minimal, generally below 2%, and mostly within the noise margin. We can conclude from this that there is no performance argument against using the long format page table.

There are, however, strong arguments in favour of the long format, as it is required to support superpages and TLB entry sharing. We did not implement superpages, however, they have been shown to be very effective in improving performance under certain circumstances [NIDC02]. That work was performed on an Alpha processor with a software-loaded TLB and TLB miss penalties in the order of 100 cycles. The Itanium's hardware page table walker is much faster (25 cycles best case), hence we expect a reduced benefit from superpages on this architecture. Nevertheless, superpages are the best defence against overheads resulting from insufficient TLB coverage, and the promising performance of the long page table format will make it worthwhile to implement superpage support on the Itanium.

The results for TLB-entry sharing are less convincing. While we were able to show that TLB-entry sharing could result in a twofold performance increase, this was in an extreme situation constructed specifically to maximise the benefits. The lmbench and aim results showed much more moderate gains, between zero and 44%, with improvements quite small in most cases. This is more indicative of what can be expected in typical Linux workloads, which neither tend to exhibit a large degree of sharing nor particularly high context-switching rates. However, in scenarios where sharing is more significant and context-switching more frequent, as might be the case in database or other transaction-processing environments, more of the potential benefits of sharing TLB entries might be realised. We are making our code available in the hope that it might be tested in some such scenarios.

There are, however, several developments which have the potential to increase the importance of TLB-entry sharing. Virtual machines are increasingly used to run several copies of the same operating system on the same physical hardware [Wal02]. Furthermore, there is renewed interest in isolating buggy and potential malicious OS code (particularly device drivers) into separate protection contexts [SMLE02, WSG02]. These approaches have in common an increase in both sharing and switching between different contexts. Our results show that under such circumstances, significant performance benefits might be possible from TLB sharing.

The main advantage of TLB sharing is that it can be implemented totally transparently in the OS, with no cost other than a very modest degree of added kernel complexity.

Availability

Kernel patches for long VHPT format and TLB entry sharing are being made available at http://gelato.unsw.edu.au.

Acknowledgements

This work would not have been possible without the equipment donations by Intel and Hewlett Packard under the *Itanium Processor Family University Grants Program*. We also gratefully acknowledge the financial support by HP, the Gelato Federation and the Australian Research Council under its Linkage Program.

References

[BKW94] Kavita Bala, M. Frans Kaashoek, and William E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proceedings of the*

1st USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 243–253, Monterey, CA, USA, 1994. USENIX/ACM/IEEE. 7

[CBJ92] J. Bradley Chen, Anita Borg, and Norman P. Jouppi. A simulation based study of TLB performance. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*. ACM, 1992. 3

[CE85] Douglas W. Clark and Joel S. Emer. Performance of the VAX-11/780 translation buffer: Simulation and measurement. *ACM Transactions on Computer Systems*, 3:31–62, 1985. 6

[CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12:271–307, 1994. 14

[Dig92] Digital Equipment Corp., Maynard, MA, USA. *Alpha Architecture Handbook*, 1992. 5

[DMY99] Cort Dougan, Paul Mackerras, and Victor Yodaiken. Optimizing the idle task and other MMU tricks. In *osdi99*, pages 229–237, New Orleans, LA, USA, February 1999. USENIX. 13

[EHL99] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. Page tables for 64-bit computer systems. In *Proceedings of the 4th Australasian Computer Architecture Conference (ACAC)*, pages 211–226, Auckland, New Zealand, January 1999. Springer Verlag. Available from URL http://www.cse.unsw.edu.au/~disy/papers/. 13

[Hei93] Joseph Heinrich. *MIPS R4000 User's Manual*. Prentice Hall, 1993. 5

[HEV$^+$98] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, July 1998. 14

[HH93] Jerry Huck and Jim Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)*, pages 39–50. ACM, 1993. 3, 13

[Int00] Intel Corp. *Itanium Architecture Software Developer's Manual*, February 2000. URL http://developer.intel.com/design/itanium/family. 3

[Int01] Intel Corp. *IA-32 Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2001. URL ftp://download.intel.com/design/Pentium4/manuals/245472.htm. 15

[Jag95] Dave Jagger, editor. *Advanced RISC Machines Architecture Reference Manual*. Prentice Hall, July 1995. 3

[KCE92] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architectural support for single-address-space operating systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–86, 1992. 14

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992. 15

[KJW94] Yousef A. Khalidi, Vikram P. Joshi, and Dock Williams. A study of the structure and performance of MMU handling software. Technical Report SMLI TR-94-28, Sun Microsystems Labs, Mountainview, CA, USA, June 1994. 13

[KS02] Gokul B. Kandiraju and Anand Sivasubramaniam. Going the distance for TLB prefetching: An application-driven study. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, Anchorage, USA, May 2002. 3

[KT95] Yousef A. Khalidi and Madhusudhan Talluri. Improving the address translation performance of widely shared pages. Technical Report TR-95-38, Sun Microsystems Laboratories, Mountain View CA, February 1995. 13, 15

[Lie93] Jochen Liedtke. A high resolution MMU for the realization of huge fine-grained address spaces and user level mapping. Arbeitspapiere der GMD No. 791, German National Research Center for Computer Science (GMD), Sankt Augustin, Germany, 1993. 15

[LL82] Henk M. Levy and P. H. Lipman. Virtual memory management in the VAX/VMS operating system. *IEEE Computer*, 15(3):35–41, March 1982. 6

[MS96] Layy McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 USENIX Technical Conference*, San Diego, CA, USA, January 2996. 9

[MSSW94] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture: A Specifi cation for a New Family of RISC Processors*. Morgan Kaufmann, 1994. 4

[Mur98] John Murray. *Inside Microsoft Windows CE*. Microsoft Press, 1998. 14

[NIDC02] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, USA, December 2002. 3, 7, 15

[SCO] SCO Inc. Aim benchmarks. http://www.caldera.com/developers/-community/contrib/aim.html. 9

[SMLE02] Michael M. Swift, Steven Marting, Henry M. Levy, and Susan G. Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th SIGOPS European Workshop*, pages 101–107, St Emilion, France, September 2002. 16

[SPA91] SPARC International Inc., Menlo Park, CA, USA. *The SPARC Architecture Manual, Version 8*, 1991. http://www.sparc.org/standards.html. 15

[Tal95] Madhusudhan Talluri. *Use of Superpages and Subblocking in the Address Translation Hierarchy*. Phd thesis, University of Wisconsin-Madison Computer Sciences, 1995. Technical Report #1277. 3

[THK95] Madhusudha Talluri, Mark D. Hill, and Yousef A. Khalid. A new page table for 64-bit address spaces. In *Proceedings of the 15th ACM Symposium on OS Principles (SOSP)*, pages 184–200, Copper Mountain Resort, Co, USA, December 1995. 13

[UNS$^+$94] Richard Uhlig, David Nagle, Tim Stanley, Trevor Mudge, Stuart Sechrest, and Richard Brown. Design tradeoffs for software-managed TLBs. *ACM Transactions on Computer Systems*, pages 175–205, 1994. 13

[Wal02] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, USA, 2002. 16

[WCA02] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002. 15

[WS92] John Wilkes and Bart Sears. A comparison of protection lookaside buffers and the PA-RISC protection architecture. Technical Report HPL-92-55, HP Labs, Palo Alto, CA, USA, March 1992. 3, 15

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the denali isolation kernel. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, USA, December 2002. 16

[WWTH03] Adam Wiggins, Simon Winwood, Harvey Tuch, and Gernot Heiser. Legba: Fast hardware support for fine-grained protection. In *8th Asia-Pacifi c Computer Systems Architecture Conference (ACSAC)*, April 2003. Submitted. 14