Itanium — A System Implementor's Tale

Charles Gray[†] Matthew Chapman[†‡] Peter Chubb[†‡] David Mosberger-Tang[§]
Gernot Heiser[†‡]
[†] *The University of New South Wales, Sydney, Australia*
[‡] *National ICT Australia, Sydney, Australia*
[§] *HP Labs, Palo Alto, CA*
cgray@cse.unsw.edu.au

Abstract

Itanium is a fairly new and rather unusual architecture. Its defining feature is explicitly-parallel instruction-set computing (EPIC), which moves the onus for exploiting instruction-level parallelism (ILP) from the hardware to the code generator. Itanium theoretically supports high degrees of ILP, but in practice these are hard to achieve, as present compilers are often not up to the task. This is much more a problem for systems than for application code, as compiler writers' efforts tend to be focused on SPEC benchmarks, which are not representative of operating systems code. As a result, good OS performance on Itanium is a serious challenge, but the potential rewards are high.

EPIC is not the only interesting and novel feature of Itanium. Others include an unusual MMU, a huge register set, and tricky virtualisation issues. We present a number of the challenges posed by the architecture, and show how they can be overcome by clever design and implementation.

1 Introduction

Itanium [7] (also known as IA64) was introduced in 2000. It had been jointly developed by Intel and HP as Intel's architecture for the next decades. At present, Itanium processors are used in high-end workstations and servers.

Itanium's strong floating-point performance is widely recognised, which makes it an increasingly popular platform for high-performance computing. Its small-scale integer performance is so far less impressive. This is partially a result of integer performance being very dependent on the ability of the hardware to exploit any instruction-level parallelism (ILP) available in the code.

Most high-end architectures detect ILP in hardware, and re-order the instruction stream in order to maximise it. Itanium, by contrast, does no reordering, but instead relies on the code generator to identify ILP and represent it in the instruction stream. This is called *explicitly-parallel instruction-set computing* (EPIC), and is based on the established (but to date not overly successful) *very-long instruction word* (VLIW) approach. EPIC is based on the realisation that the ILP that can be usefully exploited by reordering is limited, and aims at raising this limit.

The performance of an EPIC machine is highly dependent on the quality of the compiler's optimiser. Given the novelty of the architecture, it is not surprising that contemporary compilers are not quite up to the challenge [22]. Furthermore, most work on compilers is focusing on application code (in fact, mostly on SPEC benchmarks), so compilers tend to perform even worse on systems code. Finally, of the various compilers around, by far the weakest, GCC, is presently the default for compiling the Linux kernel. This poses a number of challenges for system implementors who strive to obtain good OS performance on Itanium.

Another challenge for the systems implementor is presented by Itanium's huge register file. This helps to keep the pipelines full when running CPU-bound applications, but if all those registers must be saved and restored on a context switch, the costs will be significant, Itanium's high memory bandwidth notwithstanding. The architecture provides a *register stack engine* (RSE) which automatically fills/spills registers to memory. This further complicates context switches, but has the potential for reducing register filling/spilling overhead [21]. The large register set, and the mechanisms for dealing with it, imply trade-offs that lead to different implementation strategies for a number of OS services, such as signal handling.

Exceptions are expensive on processors with high ILP and deep pipelines, as they imply a break in the execution flow that requires flushing the pipeline and wasting many issue slots. For most exceptions this is unavoidable but irrelevant if the exceptions are relatively infrequent (like interrupts) or a result of program faults. System calls, however, which are treated as exceptions on most architectures, are not faults nor necessarily infrequent, and must be fast. Itanium deals with this issue by providing a mechanism for increasing the privilege level without an exception and the corresponding

First published in *Proceedings of USENIX 05: General Track* Anaheim, CA, USA, April 2005.

pipeline flush, but it is subject to limitations which make it tricky to utilise.

Itanium's memory-management unit (MMU) also has some unusual properties which impact on OS design. Not only does it support a wide range of page sizes (which is nothing unusual), it also supports the choice of two different hardware page-table formats, a virtual linear array (called *short VHPT* format) and a hash table (called the *long VHPT* format). As the names imply, they have different size page table entries, and different performance and feature tradeoffs, including the support for superpages and the so-called *protection keys*. The hardware page-table walker can even be disabled, effectively producing a software-loaded TLB.

Protection keys loosen the usual nexus between protection and translation: access rights on pages are not only determined by access bits on page-table entries, but also by an orthogonal mechanism which allows grouping sets of pages for access-control purposes. This mechanism also supports sharing of a single entry in the translation lookaside buffer (TLB) between processes sharing access to the page, even if their access rights differ.

The original architecture is disappointing in a rather surprising respect: it is not fully virtualisable. Virtual-machine monitors (VMMs) have gained significant popularity in recent years, and Itanium is almost, but not quite, virtualisable. This creates a significant challenge for anyone who wants to develop an Itanium VMM. Fortunately, Intel recognised the deficiency and is addressing it with an architecture-extension called Vanderpool Technology [10], which is to be implemented in future CPUs.

This paper presents a discussion of the features of the Itanium architecture which present new and interesting challenges and design tradeoffs to the system implementor. We will discuss the nature of those challenges, and how they can be dealt with in practice. First, however, we present an overview of the Itanium architecture in the next section. In Section 3 we discuss the most interesting features of the Itanium's memory-management unit and the design tradeoffs it implies. In Section 4 we discuss issues with virtualisation of Itanium, while Section 5 presents a number of case studies of performance tradeoffs and micro-optimisation. Section 6 concludes the paper.

2 Itanium Architecture Overview

2.1 Explicitly-parallel instruction-set computing

As stated in the Introduction, Itanium's EPIC approach is based on VLIW principles, with several instructions contained in each instruction word. Scheduling of instructions, and specification of ILP, becomes the duty of the compiler (or assembly coder). This means that details of the processor pipelines and instruction latencies must be exposed in the architecture, so the compiler can emit correct code without the processor needing to scoreboard instruction dependencies.

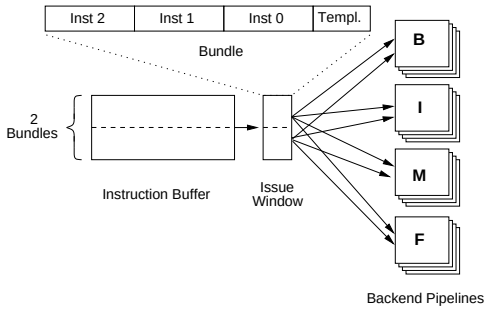

Figure 1: Instruction Issue

The Itanium approach to EPIC aims at achieving this without overly limiting the design space of future processors, i.e., by describing ILP in a way that does not depend on the actual number of pipelines and functional units. The compiler is encouraged to maximise ILP in the code, in order to optimise performance for processors regardless of pipeline structure. The result is a greatly simplified instruction issue, with only a few pipeline stages dedicated to the processor front-end (two front-end and six back-end stages, ignoring floating point, for Itanium 2). The shorter pipeline helps to reduce exception and mis-prediction penalties.

Itanium presents a RISC-like load/store instruction set. Instructions are grouped into 128-bit *bundles*, which generally hold three instructions each. Several bundles form an *instruction group* delimited by *stops*. Present Itanium processors use a two-bundle issue window (resulting in an issue of six instructions per cycle). By definition, all instructions in a group are independent and can execute concurrently (subject to resource availability).

Figure 1 shows the first few stages of the Itanium pipeline. Bundles are placed into the instruction buffer speculatively and on demand. Each clock cycle, all instructions in the issue window are dispersed into backend pipelines (branch, memory, integer and floating-point) as directed by the *template*, unless a required pipeline is stalled or a stop is encountered in the instruction stream.

Each bundle has a 5-bit template field which specifies which instructions are to be dispersed into which pipeline types, allowing the instruction dispersal to be implemented by simple static logic. If there are not enough backend units of a particular type to disperse an instruction, *split issue* occurs; the preceding instructions

are issued but that instruction and subsequent instructions must wait until the next cycle — Itanium issues strictly in order. This allows a compiler to optimise for a specific processor based on the knowledge of the number of pipelines, latencies etc., without leading to incorrect execution on earlier or later processors.

One aspect of EPIC is to make even *data* and *control speculation* explicit. Itanium supports this through *speculative load* instructions, which the compiler can move forward in the instruction stream without knowing whether this is safe to do (the load could be through an invalid pointer or the memory location overwritten through an alias). Any exception resulting from a speculative load is deferred until the result is consumed. In order to support speculation, general registers are extended by an extra bit, the *NaT* ("not a thing") bit, which is used to trap mis-speculated loads.

2.2 Register stack engine

Itanium supports EPIC by a huge file of architected registers, rather than relying on register renaming in the pipeline. There are 128 user-mode *general registers* (GRs), the first 32 of which are *global*; 16 of these are banked (i.e., there is a separate copy for privileged mode). The remaining 96 registers are explicitly renamed by using register windows, similar to the SPARC [23].

Unlike the SPARC's, Itanium's register windows are of variable size. A function uses an `alloc` instruction to allocate *local* and *output* registers. On a function call via the `br.call` instruction, the window is rotated up past the local registers leaving only the caller's output registers exposed, which become the callee's *input* registers. The callee can then use `alloc` to widen the window for new local and output registers. On executing the `br.ret` instruction, the caller's register window is restored.

The second, and most important, difference to the SPARC is the Itanium's *register stack engine* (RSE), which transparently spills or fills registers from memory when the register window overflows or underflows the available registers. This not only has the advantage of freeing the program from dealing with register-window exceptions. More importantly, it allows the processor designers to transparently add an arbitrary number of windowed registers, beyond the architected 96, in order to reduce memory traffic from register fills/spills. It also supports lazy spilling and pre-filling by the hardware.

Internally, the stack registers are partitioned into four categories — current, dirty, clean and invalid. *Current* registers are those in the active procedure context. *Dirty* registers are those in a parent context which have not yet been written to the backing store, while *clean* registers are parent registers with valid contents that have been written back (and can be discarded if necessary). *Invalid* registers contain undefined data and are ready to be allocated or filled.

The RSE operation is supported by a number of special instructions. The `flushrs` instruction is used to force the dirty section of registers to the backing store, as required on a context switch. Similarly, the `loadrs` instruction is used to reload registers on a context switch. The `cover` instruction is used to allocate an empty register frame above the previously allocated frame, ensuring any previous frames are in the dirty or clean partitions.

There is another form of register renaming: *register rotation*, which rotates registers within the current register window. This is used for so-called *software pipelining* and supports optimisations of tight loops. As this is mostly relevant at application level it is not discussed further in this paper.

2.3 Fast system calls

Traditionally, a system call is implemented by some form of invalid instruction exception that raises the privilege level, saves some processor state and diverts to some handler code. This is essentially the same mechanism as an interrupt, except that it is synchronous (triggered by a specific instruction) and therefore often called a *software interrupt*.

Such an exception is inherently expensive, as the pipeline must be flushed, and speculation cannot be used to mitigate that cost. Itanium provides a mechanism for raising the privilege level without an exception, based on *call gates*. The MMU supports a special permission bit which allows designating a page as a *gate page*. If an *`epc`* instruction in such a page is executed, the privilege level is raised without any other side effects. Code in the call page (or any code jumped to once in privileged mode) can access kernel data structures and thus implement system calls. (Other architectures, such as IA-32, also provide gates. The Itanium version is more tricky to use, see Section 5.2).

2.4 Practical programming issues

The explicit management of ILP makes Itanium performance critically dependent on optimal scheduling of instructions in the executable code, and thus puts a stronger emphasis on compiler optimisation (or hand-optimised assembler) than other architectures. In this section we discuss some of these issues.

2.4.1 Bundling and latencies

The processor may issue less than a full (six instruction) issue window in a number of cases (*split issue*). This can happen if the instructions cannot be issued concurrently due to dependencies, in which case the compiler

inserts *stops* which instruct the processor to split issue. Additionally, split issue will occur if the number of instructions for a particular functional unit exceeds the (processor-dependent) number of corresponding backend units available. Split issue may also occur in a number of processor-specific cases. For example, the Itanium 2 processor splits issue directly after serialisation instructions (`srlz` and `sync`).

Optimum scheduling also depends on accurate knowledge of instruction latency, defined as the number of cycles of separation needed between a producing instruction and a consuming instruction. Scheduling a consuming instruction within less than the producing instruction's latency does not lead to incorrect results, but stalls execution not only of this instruction, but also of all in the current and subsequent instruction-groups.

ALU instructions as well as load instructions that hit in the L1 cache have single-cycle latencies. Thus the great majority of userspace code can be scheduled without much consideration of latencies — one simply needs to ensure that consumers are in instruction groups subsequent to producers.

However, the situation is different for system instructions, particularly those accessing *control registers* and *application registers*. On the Itanium 2 processor, many of these have latencies of 2–5 cycles, a few (processor-state register, RSE registers and kernel registers) have latencies of 12 cycles, some (timestamp counter, interrupt control and performance monitoring registers) have 36 cycle latencies. This makes scheduling of systems code difficult, and the performance cost of getting it wrong very high.

2.4.2 Other pipeline stalls

Normally latencies can be dealt with by overlapping execution of several bundles (Itanium supports out-of-order completion). However, some instructions cannot be overlapped, producing unconditional stalls. This naturally includes the various serialisation instructions (`srlz`, `sync`) but also instructions that force RSE activity (`flushrs`, `loadrs`). Exceptions and the `rfi` (return from exception) instruction also produce unavoidable stalls, but these can be avoided for system calls by using `epc`.

There also exist other constraints due to various resource limitations. For example, while stores do not normally stall, they consume limited resources (store buffers and L2 request queue entries) and can therefore stall if too many of them are in progress. Similarly, the high-latency accesses to privileged registers are normally queued to avoid stalls and allow overlapped execution. However, this queue is of limited size (8 entries on Itanium 2); only one result can be returned per cycle, and the results compete with loads for writeback resources. Moreover, accesses to the particularly slow registers (timestamp counter, interrupt control and performance monitoring registers) can only be issued every 6 cycles.

A case study of minimising stalls resulting from latencies in system code is given in Section 5.3.

3 Memory-Management Unit

3.1 Address translation and protection

As mentioned earlier, the memory-management unit (MMU) of the Itanium has a number of unusual features. The mechanics of address translation and access-right lookup are schematically shown in Figure 2. The top three bits of the 64-bit virtual address form the *virtual region number*, which is used to index into a set of eight *region registers* (RRs) which contain *region IDs*.

The remaining 61 bits form the *virtual page number* (VPN) and the *page offset*. Itanium 2 supports a wide range of page sizes, from 4kB to 4GB. The VPN is used together with the region ID to perform a fully-associative lookup of the *translation lookaside buffer* (TLB). The region ID serves as a generalisation of the *address-space ID* (ASID) tags found on many RISC processors.

Like an ASID, the region ID supports the co-existence of mappings from different contexts without causing aliasing problems, but in addition allows for simple sharing of pages on a per-region basis: if two processes have the same region ID in one of their RRs, they share all mappings in that region. This provides a convenient way for sharing text segments, if one region is reserved for program code and a separate region ID is associated with each executable. Note that if region IDs are used for sharing, the processes not only share pages, but actually share the TLB entries mapping those pages. This helps to reduce TLB pressure.

A more unusual feature of the Itanium TLB is the *protection key* tag on each entry (which is a generalisation of the *protection-domain identifiers* of the PA-RISC [24]). If protection keys are enabled, then the key field of the matching TLB entry is used for an associative lookup of another data structure, a set of *protection key registers* (PKRs). The PKR contains a set of access rights which are combined with those found in the TLB to determine the legality of the attempted access. This can be used to implement write-only mappings (write-only mode is not supported by the rights field in the TLB).

Protection keys can be used to share individual (or sets of) pages with potentially different access rights. For example, if two processes share a page, one process with read-write access, the other read-only, then the page can be marked writable in the TLB, and given a protection key. In the one process's context, the rights field in the

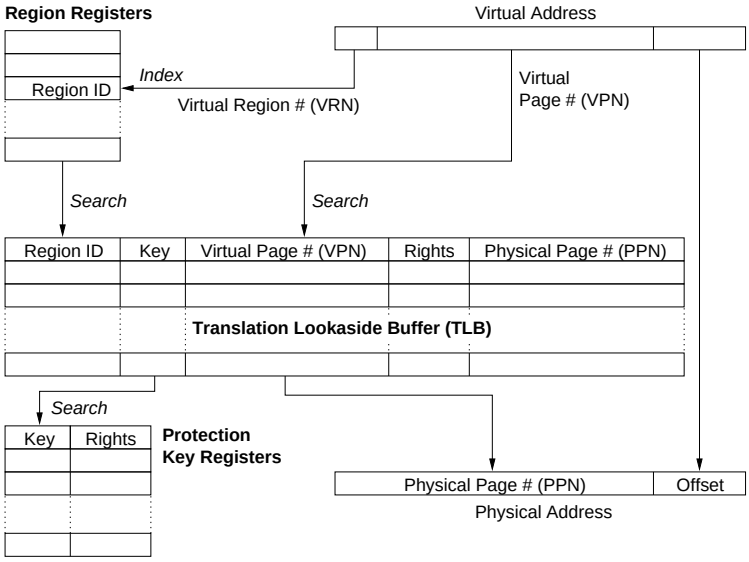

Figure 2: Itanium address translation and memory protection.

corresponding PKR would be set to read-write, while for the other process it would be set to read-only. The processes again share not only the page but also the actual TLB entries. The OS can even use the rights field in the TLB to downgrade access rights for everybody, e.g. for implementing copy-on-write, or for temporarily disabling all access to the page.

3.2 Page tables

The Itanium has hardware support for filling the TLB by walking a page table called the *virtual hashed page table* (VHPT). There are actually two hardware-supported page-table formats, called the *short-format* and *long-format* VHPT respectively. The hardware walker can also be completely turned off, requiring all TLB reloads to be done in software (from an arbitrary page table structure).

Turning off the hardware walker is a bad idea. We measured the average TLB refill cost in Linux to be around 45 cycles on an Itanium 2 with the hardware walker enabled, compared to around 160 cycles with the hardware walker disabled. A better way of supporting arbitrary page table formats is to use the VHPT as a hardware-walked software TLB [2] and reload from the page table proper on a miss.

Figure 3 shows the format and access of the two types of page table. The short-format VHPT is, name notwithstanding, a *linear virtual array page table* [5, 12] that is indexed by the page number and maps a single region, hence up to eight are required per process, and the size of each is determined by the page size. 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 region ID need not be specified in the short VHPT, as it is implicit in the access (each region uses a separate VHPT).

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 hardware walker can be used for only one page size per region. Non-default page-sizes within a region would have to be handled by (slower) software fills.

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 region ID, except that the two might be of different length). This makes it impossible to specify different protection keys in a region if the short-format VHPT is used. Hence, sharing TLB entries of selected (shared) pages within a region is not possible with this page table format.

The long VHPT is a proper hashed page table, indexed by a hash of the page number. Its size can be an arbitrary power of two (within limits), and a single table can be used for all regions. 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, a tag and a chain field. 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. The chain field is ignored by the hardware and can be used by the operating system to implement overflow chains.

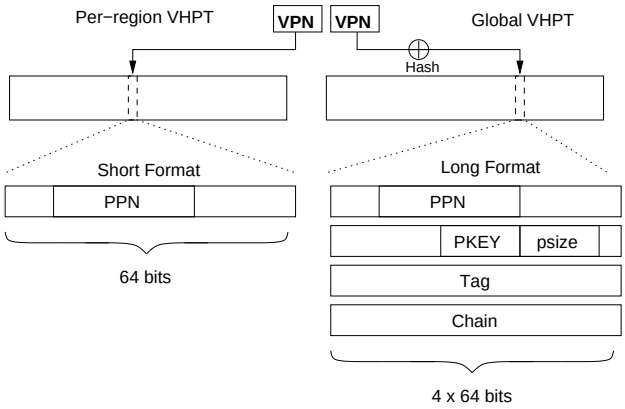

Figure 3: Short and long VHPT formats.

3.3 VHPT tradeoffs

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. Furthermore, hashing destroys locality, and hence the probability of two PTEs sharing a cache line is small, unless the page table is small and the page working set large (a situation which will result in collisions and expensive software walks). Hence, the long VHPT format tends to be less cache-friendly than the short format.

The long-format VHPT makes up for this by being more TLB friendly. For the short format, 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. Linux in fact, typically uses three regions for user code, and thus will require at least that many entries for mapping a single process's page tables. In contrast, a process's whole long-format VHPT can be mapped with a single large superpage mapping. Furthermore, a single long-format VHPT can be shared between all processes, reducing TLB entry consumption for page tables from ≥ 3 per process to one per CPU.

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 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 hardware-loaded mixed page sizes (superpages) within a region. Superpages have been shown to lead to significant performance improvements [17] and given the overhead of handling TLB-misses in software, it is desirable to take advantage of the hardware walker. As Linux presently uses the short-format VHPT, doing so would require a switch of the VHPT format first. This raises the question whether the potential performance gain might be offset by a performance loss resulting from the large page-table format.

3.4 Evaluation

We did a comparison of page-table formats by implementing the long-format VHPT in the Linux 2.6.6 kernel. We ran the lmbench [15] suite as well as Suite IX of the aim benchmark [1], and the OSDL DBT-2 benchmark [18]. Tests were run on a HP rx2600 server with dual 900MHz Itanium-2 CPUs. The processors have three levels of 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. Using 16kB pages, the per-CPU long-format VHPT was sized at 16MB in our experiments, being four times the size needed to map the entire 2G physical memory.

The results for the lmbench process and file-operation benchmarks are uninteresting. They show that the choice of page table has little impact on performance. This is not very surprising, as for these benchmarks there is no significant space pressure on either the CPU caches or the TLB.

Somewhat more interesting are the results of the lmbench context-switching benchmarks, shown in Table 1. Here the long-format page table shows some noticeable performance advantage with a large number of processes but small working sets (and consequently high context-switching rates). This is most likely a result of the long-format VHPT reducing TLB pressure. The performance of the two systems becomes equal again when the working sets increase, probably a result of the better cache-friendliness of the short-format page table, and the reduced relative importance of TLB miss handling costs.

The other lmbench runs as well as the aim bench-

Context switching with 0K							
	2proc	4proc	8proc	16proc	32proc	64proc	96proc
U	0.98	1.00	0.95	0.88	0.98	1.44	1.34
M	0.94	0.96	0.95	0.96	1.23	1.30	1.27

Context switching with 4K							
	2proc	4proc	8proc	16proc	32proc	64proc	96proc
U	0.97	0.99	0.97	0.95	1.17	1.20	1.09
M	0.95	0.61	0.78	0.87	1.11	1.13	1.09

Context switching with 8K							
	2proc	4proc	8proc	16proc	32proc	64proc	96proc
U	0.99	0.98	0.96	0.97	1.31	1.17	1.08
M	0.95	0.91	0.96	1.00	1.29	1.15	1.06

Context switching with 16K							
	2proc	4proc	8proc	16prc	32prc	64prc	96prc
U	0.99	0.98	0.96	0.97	1.31	1.17	1.08
M	0.95	0.91	0.96	1.00	1.29	1.15	1.06

Context switching with 32K							
	2proc	4proc	8proc	16prc	32prc	64prc	96prc
U	0.98	0.99	1.04	1.30	1.04	1.03	1.00
M	0.94	0.96	1.00	1.01	0.87	1.00	1.00

Context switching with 64K							
	2proc	4proc	8proc	16prc	32prc	64prc	96prc
U	1.00	0.98	0.94	0.94	1.00	1.00	1.00
M	0.97	0.98	1.06	1.22	0.94	0.99	0.98

Table 1: Lmbench context-switching results. Numbers indicate performance with a long-format VHPT relative to the short-format VHPT: a figure > 1.0 indicates better, < 1.0 worse performance than the short-format page table. Lines marked "U" are for a uniprocessor kernel, while "M" is the same for a multiprocessor kernel (on a two-CPU system).

mark results were similarly unsurprising and are omitted for space reasons. Complete results can be found in a technical report [4].

The SPEC CPU2000 integer benchmarks, AIM7 and lmbench show no cases where the long-format VHPT resulted in significantly worse performance than the short-format VHPT, provided the long-format VHPT is sized correctly (with the number of entries equal to four times the number of page frames).

We also ran OSDL's DBT-2 benchmark, which emulates a warehouse inventory system. This benchmark stresses the virtual memory system — it has a large resident set size, and has over 30 000 TLB misses per second. The results show no significant performance difference at an 85% confidence level — for five samples, the long format VHPT gave 400(6) transactions per minute, and the short format page table gave 401(4) transactions per minute (standard deviation in the parentheses).

We also investigated TLB entry sharing, but found no significant benefits with standard benchmarks [4].

Based on these experiments, we conclude that long-format VHPT can provide performance as good or better than short-format VHPT. Given that long-format VHPT also enables hardware-filled superpages and TLB-entry-sharing across address-spaces, we believe it may very well make sense to switch Linux to the long-format VHPT in the future.

4 Virtualisation

Virtualisability of a processor architecture [20] generally depends on a clean separation between *user* and *system* state. Any instructions that inspect or modify the system state (*sensitive instructions*) must be privileged, so that the VMM can intervene and emulate their behaviour with respect to the simulated machine. Some exceptions to this may be permissible where the virtual machine monitor can ensure that the real state is synchronised with the simulated state.

In one sense Itanium is simpler to virtualise than IA-32, since most of the instructions that inspect or modify system state are privileged by design. It seems likely that the original Itanium designers believed in this clear separation of user and system state which is necessary for virtualisation. Sadly, a small number of non-virtualisable features have crept into the architecture, as we discovered in our work on the vNUMA distributed virtual machine [3]. Some of these issues were also encountered by the authors of vBlades [14], a recent virtual machine for the Itanium architecture.

The `cover` instruction creates a new empty register stack frame, and thus is not privileged. However, when executed with interruption collection off (interruption collection controls whether execution state is saved to the interruption registers on an exception), it has the side-effect of saving information about the previous stack frame into the privileged *interruption function state* (IFS) register. Naturally, it would not be wise for a virtual machine monitor to actually turn off interruption collection at the behest of the guest operating system, and when the simulated interruption collection bit is off, there is no way for it to intercept the `cover` instruction and perform the side-effect on the simulated copy of IFS. Hence, `cover` must be replaced with an instruction that faults to the VMM, either statically or at run time.

The `thash` instruction, given an address, calculates the location of the corresponding hashtable entry in the VHPT. The `ttag` instruction calculates the corresponding tag value. These instructions are, for some reason, unprivileged. However, they reveal processor memory management state, namely the pagetable base, size and format. When the guest OS uses these instructions, it obtains information about the real pagetables instead of its own pagetables. Therefore, as with `cover`, these instructions must be replaced with faulting versions.

Virtual memory semantics also need to be taken into account, since for a virtual machine to have reasonable performance, the majority of virtual memory accesses need to be handled by the hardware and should not trap to the VMM. For the Itanium architecture, most features can be mapped directly. However, a VMM will need to reserve some virtual address space (at least for exception handlers). One simple way to do this is to report a smaller virtual address space than implemented on the real processor, thereby ensuring that the guest operating system will not use certain portions. On the other hand, the architecture defines a fixed number of privilege levels (0 to 3). Since the most privileged level must be reserved for the VMM, this means that the four privilege levels in the guest must be mapped onto three real privilege levels (a common technique known as *ring compression*). This means there may be some loss of protection, though most operating systems do not use all four privilege levels.

The Itanium architecture provides separate control over instruction translation, data translation and register-stack translation. For example, it is possible to have register-stack translation on (virtual) and data translation off (physical). There is no way to efficiently replicate this in virtual mode, since register-stack references and data references access the same virtual address space.

Finally, if a fault is taken while the register-stack engine is filling the current frame, the RSE is halted and the exception handler is executed with an incomplete frame. As soon as the exception handler returns, the RSE resumes trying to load the frame. This poses difficulties if the exception handler needs to return to the guest kernel (at user-level) to handle the fault.

Future Itanium processors will have enhanced virtualisation support known as Vanderpool Technology. This provides a new processor operating mode in which sensitive instructions are properly isolated. Additionally, this mode is designed so as to allow the guest operating system to run at its normal privilege level (0) without compromising protection, negating the need for ring compression. Vanderpool Technology also provides facilities for some of the virtualisation to be handled in hardware or firmware (*virtualisation acceleration*). In concert these features should provide for simpler and more efficient virtualisation. Nevertheless, there remain some architectural features which are difficult to virtualise efficiently and require special treatment, in particular the translation modes and the RSE issue described above.

5 Case studies

In this section we present three implementation studies which we believe are representative of the approaches that need to be taken to develop well-performing systems software on Itanium. The first example, implementation of signals in Linux, illustrates that Itanium features (in this case, the large register file) lead to different tradeoffs from these on other architectures. The second example investigates the use of the fast system-call mechanism in Linux. The third, micro-optimisation of a fast system-call path, illustrates the challenges of EPIC (and the cost of insufficient documentation).

5.1 Efficient signal delivery

In this section we explore a technique to accelerate signal delivery in Linux. This is an exercise in intelligent state-management, necessitated by the large register file of the Itanium processor, and relies heavily on exploiting the software conventions established for the Itanium architecture [8]. The techniques described here not only improved signal-delivery performance on Itanium Linux, but also simplified the kernel.

In this section we use standard Itanium terminology. We use *scratch register* to refer to a caller-saved register, i.e., a register whose contents is *not* preserved across a function-call. Analogously, we use *preserved register* to refer to a callee-saved register, i.e., a register whose contents *is* preserved across a function-call.

5.1.1 Linux signal delivery

The canonical way for delivering a signal in Linux consists of the following steps:

- On any entry into the kernel (e.g., due to system call, device interrupt, or page-fault), Linux saves the scratch registers at the top of the kernel-stack in a structure called `pt_regs`.
- Right before returning to user level, the kernel checks whether the current process has a signal pending. If so, the kernel:
  1. saves the contents of the *preserved* registers on the kernel-stack in a structure called `switch_stack` (on some architectures, the `switch_stack` structure is an implicit part of `pt_regs` but for the discussion here, it's easier to treat it as separate);
  2. calls the routine to deliver the signal, which may ignore the signal, terminate the process, create a core dump, or arrange for a signal handler to be invoked.

The important point here is that the combination of the `pt_regs` and `switch_stack` structures contain the full user-level state (machine context). The `pt_regs` structure obviously contains user-level state, since it is created right on entry to the kernel. For the `switch_stack` structure, this is also true but less obvious: it is true because at the time the `switch_stack` structure is created, the kernel stack is empty apart from

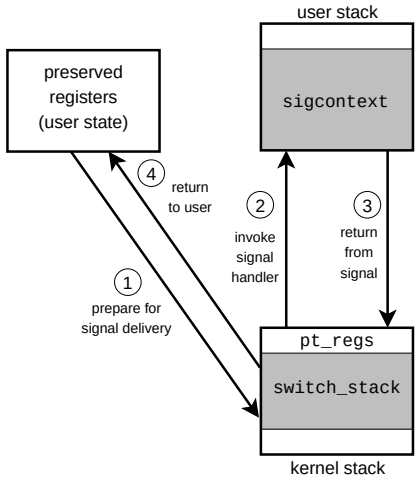

Figure 4: Steps taken during signal delivery

the `pt_regs` structure. Since there are no intermediate call frames, the preserved registers must by definition contain the original user-level state.

Signal-delivery requires access to the full user-level state for two reasons:

1. if the signal results in a core dump, the user-level state needs to be written to the core file;
2. if the signal results in the invocation of a signal handler, the user-level state needs to be stored in the `sigcontext` structure.

5.1.2 Performance Considerations

The problem with the canonical way of delivering a signal is that it entails a fair amount of redundant moving of state between registers and memory. For example, as illustrated in Figure 4, the preserved registers:

1. get saved on the kernel stack in preparation for signal delivery
2. get copied to the user-level stack in preparation for invoking a signal handler
3. get copied back to the kernel stack on return from a signal-handler
4. need to be restored from the kernel-stack upon returning execution to user level.

On architectures with small numbers of architected registers, redundant copying of registers is not a big issue, particularly since their contents is likely to be hot in the cache anyway. However, with Itanium's large register file, the cost of copying registers can be high.

When faced with this challenge, we decided that rather than trying to micro-optimise the moving of the state, a better approach would be to *avoid* the redundant moves in the first place. This was helped by the following observations:

- For a core dump, the preserved registers can be reconstructed after the fact with the help of a kernel-stack unwinder. Specifically, when the kernel needs to create a core dump, it can take a snapshot of the current registers and then walk the kernel stack. In each stack frame, it can update the snapshot with the contents of the registers saved in that stack frame. When reaching the top of the kernel-stack, the snapshot contains the desired user-level state.
- There is no *inherent* reason for saving the preserved registers in the `sigcontext` structure. While it is *customary* to do so, there is nothing in the Single-UNIX Specification [19] or the POSIX standard that would require this. The reason it is not necessary to include the preserved registers in the `sigcontext` structure is that the signal handler (and its callees) automatically save preserved registers before using them and restore them before returning. Thus, there is no need to create a copy of these registers in the `sigcontext` structure. Instead, we can just leave them alone.

In combination, these two observations make it possible to completely eliminate the `switch_stack` structure from the signal subsystem.

We made this change for Itanium Linux in December 2000. At that time, there was some concern about the existence of applications which rely on having the full machine-state available in `sigcontext` and for this reason, we left the door open for a user-level compatibility-layer which would make it appear as if the kernel had saved the full state [16]. Fortunately, in the four years since making the change, we have not heard of a need to activate the compatibility layer.

To quantify the performance effect of saving only the minimal state, we forward-ported the original signal-handling code to a recent kernel (v2.6.9-rc3) and found it to be 23–34% slower. This relative slowdown varied with kernel-configuration (uni- vs. multi-processor) and chip generation (Itanium vs. Itanium 2). The absolute slowdown was about 1,400 cycles for Itanium and 700 cycles for Itanium 2. We should point out that, had it not been for backwards-compatibility, `sigcontext` could have been shrunk considerably and fewer cache-lines would have to be touched during signal delivery. In other words, in a design free from compatibility concerns, the savings could be even bigger.

Table 2 shows that saving the minimal state yields signal-delivery performance that is comparable to other architectures: even a 1GHz Itanium 2 can deliver signals about as fast as a 2.66GHz Pentium 4.

Apart from substantially speeding up signal delivery, this technique (which is not Itanium-specific) simplified the kernel considerably: it eliminated the need to maintain the `switch_stack` in the signal subsystem and

Chip		SMP cycles	SMP μs	UP cycles	UP μs
Itanium 2	1.00 GHz	3,087	3.1	2,533	2.5
Pentium 4	2.66 GHz	8,320	3.2	6,500	2.4

Table 2: Signalling times with Linux kernel v2.6.9-rc3. (SMP = multiprocessor kernel, UP = uniprocessor kernel)

System Call	Dynamic `break`	Dynamic `epc`	Static `break`	Static `epc`
`getpid()`	294	18	287	12
`getppid()`	299	77	290	54
`gettimeofday()`	442	174	432	153

Table 3: Comparison of system call costs (in cycles) using the standard (`break`) and fast (`epc`) mechanism, both with dynamically and statically linked binaries

removed all implicit dependencies on the existence of this structure.

5.2 Fast system-call implementation

5.2.1 Fast system calls in Linux

As discussed in Section 2.3, Itanium provides gate pages and the `epc` instruction for getting into kernel mode without a costly exception. Here we discuss the practicalities of using this mechanism in Linux.

After executing the `epc` instruction, the program is executing in privileged mode, but still uses the user's stack and register-backing store. These cannot be trusted by the kernel, and therefore such a system call is very limited, until it loads a sane stack and RSE backing-store pointer. This is presently not supported in Linux, and thus the fast system-call mechanism is restricted by the following conditions:

- the code cannot perform a normal function call (which would create a new register stack frame and could lead to a spill to the RSE backing store);
- the code must not cause an exception, because normal exception handling spills registers. This means that all user arguments must be carefully checked, including checking for a possible NaT consumption exception (which could normally be handled transparently).

As a result, fast system calls are presently restricted to handcrafted assembly language code, and functionality that is essentially limited to passing data between the kernel and the user. System calls fitting those requirements are inherently short, and thus normally dominated by the exception overhead, so good candidates for implementing in an exception-less way.

So far we implemented the trivial system calls `getpid()` and `getppid()`, and the somewhat less trivial `gettimeofday()`, and `rt_sigprocmask()`. The benefit is significant, as shown in Table 3: we see close to a factor of three improvement for the most complicated system call. The performance of `rt_sigprocmask()` is not shown. Currently glibc does not implement `rt_sigprocmask()`, so it is not possible to make a meaningful comparison.

5.3 Fast message-passing implementation

Linux, owing to its size and complexity, is not the best vehicle for experimenting with fast system calls. The L4 microkernel [11] is a much simpler platform for such work, and also one where system-call performance is much more critical. Message-passing inter-process communication (IPC) is the operation used to invoke any service in an L4-based system, and the IPC operation is therefore highly critical to the performance of such systems. While there is a generic (architecture-independent) implementation of this primitive, for the common (and fastest) case it is usually replaced in each port by a carefully-optimised architecture-specific version. This so-called *IPC fast path* is usually written in assembler and tends to be of the order of 100 instructions. Here we describe our experience with micro-optimising L4's IPC operation.

5.3.1 Logical control flow

The logical operation of the IPC fast path is as follows, assuming that a sender invokes the `ipc()` system call and the receiver is already blocked waiting to receive:

1. enter kernel mode (using `epc`);
2. inspect the *thread control blocks* (TCBs) of source and destination threads;
3. check that fast path conditions hold, otherwise call the generic "slow path" (written in C++);
4. copy message (if the whole message does not fit in registers);
5. switch the register stack and several other registers to the receiver's state (most registers are either used to transfer the message or clobbered during the operation);
6. switch the address space (by switching the page table pointer);
7. update some state in the TCBs and the pointer to the current thread;
8. return (in the receiver's context).

The original implementation of this operation (a combination of C++ code, compiled by GCC 3.2, and some assembly code to implement context switching) executed in 508 cycles with hot caches on an Itanium-2 machine. An initial assembler fast path to transfer up to 8 words, only loosely optimised, brought this down to 170

```
56      BACK_END_BUBBLE.ALL
  30      BE_EXE_BUBBLE.ALL
    16      BE_EXE_BUBBLE.GRALL
    14      BE_EXE_BUBBLE.ARCR
  15      BE_L1D_FPU_BUBBLE.ALL
    10      BE_L1D_FPU_BUBBLE.L1D_DCURECIR
     5      BE_L1D_FPU_BUBBLE.L1D_STBUFRECIR
  11      BE_RSE_BUBBLE.ALL
     4      BE_RSE_BUBBLE.AR_DEP
     6      BE_RSE_BUBBLE.LOADRS
     1      BE_RSE_BUBBLE.OVERFLOW
```

Figure 5: Breakdown of bubbles provided by the PMU.

cycles. While this is a factor of three faster, it is still on the high side; on RISC architectures the operation tends to take 70–150 cycles [13].[1]

5.3.2 Manual optimisation

An inspection of the code showed that it consisted of only 83 instruction groups, hence 87 cycles were lost to bubbles. Rescheduling instructions to eliminate bubbles would potentially double performance!

An attempt at manual scheduling resulted not only in an elimination of bubbles, but also a reduction of the number of instruction groups (mostly achieved by rearranging the instructions to make better use of the available instruction templates). The result was 39 instruction groups executing in 95 cycles. This means that there were still 56 bubbles, accounting for just under 60% of execution time.

The reason could only be that some instructions had latencies that were much higher than expected. Unfortunately, Intel documentation contains very little information on instruction latencies, and did not help us further.

Using the *perfmon utility* [6] to access Itanium's *performance monitoring unit* (PMU) we obtained the breakdown of the bubbles summarised in Figure 5. The data in the figure is to be read as follows: 56 bubbles were recorded by the counter `back_end_bubble.all`. This consists of 30 bubbles for `be_exe_bubble.all`, 15 bubbles for `be_l1d_fpu_bubble.all` and 11 bubbles for `be_rse_bubble.all`. Each of these is broken down further as per the figure.

Unfortunately, the Itanium 2 Processor Reference Manual [9] is not very helpful here, it typically gives a one-line summary for each PMU counter, which is insufficient to understand what is happening. What was clear, however, was the register stack engine was a significant cause of latency.

5.3.3 Fighting the documentation gap

Register-stack-engine stalls In order to obtain the information required to optimise the code further, we saw no alternative to systematically measuring the latencies between any two instructions which involve the RSE. The results of those measurements are summarised in Table 4. Some of those figures are surprising, with some seemingly innocent instructions having latencies in excess of 10 cycles. Thus attention to this table is important when scheduling RSE instructions.

Using Table 4 we were able to reschedule instructions such that almost all RSE-related bubbles were eliminated, that is, all of the ones recorded by counters `be_exe_bubble.arcr` and `be_rse_bubble.ar_dep`, plus most of `be_rse_bubble.loadrs`. In total, 23 of the 25 RSE-related bubbles were eliminated, resulting in a total execution time of 72 cycles. The remaining 2 bubbles (from `loadrs` and `flushrs` instructions) are unavoidable (see Section 2.4.2).

System-instruction latencies Of the remaining 31 bubbles, 16 are due to counter `be_exe_bubble.grall`. These relate to general register scoreboard stalls, which in this case result from accesses to long-latency registers such as the *kernel register* that is used to hold the current thread ID. Hence we measured latencies of system instructions and registers. For this we used a modified Linux kernel, where we made use of gate pages to execute privileged instructions from a user-level program. The modified Linux kernel allows user-space code to create gate pages using `mprotect()`. Executing privileged instructions from user-level code greatly simplified taking the required measurements.

Our results are summarised in Table 5. Fortunately, register latencies are now provided in the latest version of the Itanium 2 Processor Reference Manual [9], so they are not included in this table. Unlike the RSE-induced latencies, our coverage of system-instruction latencies is presently still incomplete, but sufficient for the case at hand. Using this information we eliminated the 16 remaining execution-unit-related bubbles, by scheduling useful work instead of allowing the processor to stall.

Data-load stalls This leaves 15 bubbles due to data load pipeline stalls, counted as `be_l1d_fpu_bubble.l1d_dcurecir` and `be_l1d_fpu_bubble.l1d_stbufrecir`. The Itanium 2 Processor Reference Manual explains the former as "back-end was stalled by L1D due to DCU recirculating" and the latter as "back-end was stalled by L1D due to store buffer cancel needing recirculate", which is hardly enlightening. We determined that the store buffer recirculation was most likely due to address conflicts between loads and stores (a load following a store to the same cache line within 3 cycles), due to the way we had scheduled loads and stores in parallel. Even

From	To	cyc	PMU counter
mov ar.rsc=	RSE_AR	13	BE_RSE_BUBBLE.AR_DEP
mov ar.bspstore=	RSE_AR	6	BE_RSE_BUBBLE.AR_DEP
mov =ar.bspstore	mov ar.rnat=	8	BE_EXE_BUBBLE.ARCR
mov =ar.bsp	mov ar.rnat=	8	BE_EXE_BUBBLE.ARCR
mov =ar.rnat/ar.unat	mov ar.rnat/ar.unat=	6	BE_EXE_BUBBLE.ARCR
mov ar.rnat/ar.unat=	mov =ar.rnat/ar.unat	6	BE_EXE_BUBBLE.ARCR
mov =ar.unat	FP_OP	6	BE_EXE_BUBBLE.ARCR
mov ar.bspstore=	flushrs	12	BE_RSE_BUBBLE.OVERFLOW
mov ar.rsc=	loadrs	12	BE_RSE_BUBBLE.LOADRS
mov ar.bspstore=	loadrs	12	BE_RSE_BUBBLE.LOADRS
mov =ar.bspstore	loadrs	2	BE_RSE_BUBBLE.LOADRS
loadrs	loadrs	8	BE_RSE_BUBBLE.LOADRS

Table 4: Experimentally-determined latencies for all combinations of two instructions involving the RSE. RSE_AR means any access to one of the registers `ar.rsc`, `ar.bspstore`, `ar.bsp`, or `ar.rnat`.

From	To	cyc	PMU counter
epc	ANY	1	-
bsw	ANY	6	BE_RSE_BUBBLE.BANK_SWITCH
rfi	ANY	13	BE_FLUSH_BUBBLE.BRU (1), BE_FLUSH_BUBBLE.XPN (8), BACK_END_BUBBLE.FE (3)
srlz.d	ANY	1	-
srlz.i	ANY	12	BE_FLUSH_BUBBLE.XPN (8), BACK_END_BUBBLE.FE (3)
sum/rum/mov psr.um=	ANY	5	BE_EXE_BUBBLE.ARCR
sum/rum/mov psr.um=	srlz	10	BE_EXE_BUBBLE.ARCR
ssm/rsm/mov psr.l=	srlz	5	BE_EXE_BUBBLE.ARCR
mov =psr.um/psr	srlz	2	BE_EXE_BUBBLE.ARCR
mov pkr/rr=	srlz/sync/fwb/ mf/invala_M0	14	BE_EXE_BUBBLE.ARCR
probe/tpa/tak/thash/ttag	USE	5	BE_EXE_BUBBLE.GRALL

Table 5: Experimentally-determined latencies for system instructions (incomplete). ANY means any instruction, while USE means any instruction consuming the result.

after eliminating this, there were still DCU recirculation stalls remaining.

While investigating this we noticed a few other undocumented features of the Itanium pipeline. It seems that most *application register* (AR) and *control register* (CR) accesses are issued to a limited-size buffer (of apparently 8 entries), with a "DCS stall" occurring when that buffer is full. No explanation of the acronym "DCS" is given in the Itanium manuals. It also seems that a DCU recirculation stall occurs if a DCS data return coincides with two L1 data-cache returns, which points to a limitation in the number of writeback ports. We also found that a DCU recirculation stall occurs if there is a load or store exactly 5 cycles after a move to a region register (RR) or protection-key register (PKR). These facts allowed us to identify the remaining stalls, but there may be other cases as well.

We also found a number of undocumented special split-issue cases. Split issue occurs after `srlz`, `sync` and `mov =ar.unat` and before `mf` instructions. It also occurs between a `mov =ar.bsp` and any B-unit instruction, as well as between an M-unit and an `fwb` instruction. There may be other cases.

We also found a case where the documentation on mapping of instruction templates to functional units is clearly incorrect. The manual says "$M_A M_L I - M_S M_A I$ gets mapped to ports M2 M0 I0 – M3 M1 I1. If M_S is a `getf` instruction, a split issue will occur." However, our experiments show that the mapping is really M1 M0 I0 – M2 M3 I1, and **no** split issue occurs in this case. It seems that in general the *load* subtype is allocated first.

Version	cycles	inst. grps	bubbles
C++ generic	508	231	277
Initial asm	170	83	87
Optimised	95	39	56
Final	36	33	3
Optimal	34	32	2
Archit. limit	9	9	0

Table 6: Comparison of IPC path optimisation, starting with the generic C++ implementation. *Optimised* refers to the version achieved using publicly available documentation, *final* denotes what was achieved after systematically measuring latencies. *Optimal* is what could be achieved on the present hardware with perfect instruction scheduling, while the *architectural limit* assumes unlimited resources and only single-cycle latencies.

5.3.4 Final optimisation

Armed with this knowledge we were able to eliminate all but one of the 15 data-load stalls, resulting in only 3 bubbles and a final execution time of 36 cycles, or 24ns on a 1.5GHz Itanium 2. This is extremely fast, in fact unrivalled on any other architecture. In terms of cycle times this is about a factor of two faster than the fastest RISC architecture (Alpha 21264) to which the kernel has been ported so far, and in terms of absolute time it is well beyond anything we have seen so far. This is a clear indication of the excellent performance potential of the Itanium architecture.

The achieved time of 36 cycles (including 3 bubbles) is actually still slightly short of the optimal solution on the present Itanium. The optimal solution can be found by examining the critical path of operations, which turns out to be 34 cycles (including 2 unavoidable bubbles for `flushrs` and `loadrs`). Significant manual rescheduling the code would (yet again) be necessary to achieve this 2 cycle improvement.

The bottlenecks preventing optimisation past 34 cycles are the kernel register read to obtain the current thread ID, which has a 12 cycle latency, and the latency of 12 cycles between `mov ar.bspstore=` (changing the RSE backing store pointer) and the following `loadrs` instruction. Also, since many of the instructions are system instructions which can only execute on a particular unit (M2), the availability of that unit becomes limiting. Additionally, it seems to be impossible to avoid a branch misprediction on return to user mode, as the predicted return address comes from the return stack buffer, but the nature of IPC is that it returns to a different thread. Eliminating those latencies would get us close to the architectural limit of Itanium, which is characterised as having unlimited resources (functional units) and only single-cycle latencies. This limit is a mind-boggling 9 cycles! The achieved and theoretical execution times are summarised in Table 6.

The almost threefold speedup from 95 to 36 cycles made a significant difference for the performance of driver benchmarks within our component system. It would not have been possible without the powerful performance monitoring support on the Itanium processor, particularly the ability to break down stall events. The PMU allowed us to discover and explain all of the stalls involved.

This experience also helped us to appreciate the challenges facing compiler writers on Itanium. Without information such as that of Tables 4 and 5 it is impossible to generate truly efficient code. A compiler could use this information to drive its code optimisation, eliminating the need for labour-intensive hand-scheduled assembler code. Present compilers seem to be far away from being able to achieve this. While we have not analysed system-call code from other operating systems to the same degree, we would expect them to suffer from the same problems, and benefit from the same solutions. However, system-call performance is particularly critical in a microkernel, owing to the high frequency of kernel invocations.

6 Conclusion

As has been shown, the Itanium is a very interesting platform for systems programming. It presents a number of unusual features, such as its approach to address translation and memory protection, which are creating a new design space for systems builders.

The architecture provides plenty of challenges too, including managing its large register set efficiently, and overcoming hurdles to virtualisation. However, the most significant challenge of the architecture to systems implementors is the more mundane one of optimising the code. The EPIC approach has proven a formidable challenge to compiler writers, and almost five years after the architecture was first introduced, the quality of code produced by the available compilers is often very poor for systems code. Given this time scale, the situation is not likely to improve significantly for quite a number of years.

In the meantime, systems implementors who want to tap into the great performance potential of the architecture have to resort to hand-tuned assembler code, written with a thorough understanding of the architecture and its complex instruction scheduling rules. Performance improvements by factors of 2–3 are not unusual in this situation, and we have experienced cases where performance could be improved by an order of magnitude over GCC-generated code.

Such manual micro-optimisation is made harder by the unavailability of sufficiently detailed documentation.

This, at least, seems be something the manufacturer should be able to resolve quickly.

Acknowledgements

This work was supported by a Linkage Grant from the Australian Research Council (ARC) and a grant from HP Company via the Gelato.org project, as well as hardware grants from HP and Intel. National ICT Australia is funded by the Australia Government's Department of Communications, Information Technology, and the Arts and the ARC through *Backing Australia's Ability* and the ICT Research Centre of Excellence programs.

We would also like to thank UNSW Gelato staff Ian Wienand and Darren Williams for their help with benchmarking.

Notes

1. The results in [13] were obtained kernels that were not fully functional and are thus somewhat optimistic. Also the processors used had shorter pipelines than modern high-end CPUs and hence lower hardware-dictated context switching costs. The figure of 70–150 cycles reflects (yet) unpublished measurements performed in our lab on optimised kernels for ARM, MIPS, Alpha and Power 4.

References

[1] Aim benchmarks. `http://sourceforge.net/projects/aimbench`.

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

[3] Matthew Chapman and Gernot Heiser. Implementing transparent shared memory on clusters using virtual machines. In *Proc. 2005 USENIX Techn. Conf.*, Anaheim, CA, USA, Apr 2005.

[4] Matthew Chapman, Ian Wienand, and Gernot Heiser. Itanium page tables and TLB. Technical Report UNSW-CSE-TR-0307, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, May 2003.

[5] Douglas W. Clark and Joel S. Emer. Performance of the VAX-11/780 translation buffer: Simulation and measurement. *Trans. Comp. Syst.*, 3:31–62, 1985.

[6] HP Labs. *Perfmon*. `http://www.hpl.hp.com/research/linux/perfmon/`.

[7] Jerry Huck, Dale Morris, Jonathan Ross, Allan Knies, Hans Mulder, and Rumi Zahir. Introducing the IA-64 architecture. *IEEE Micro*, 20(5):12–23, 2000.

[8] Intel Corp. *Itanium Software Conventions and Runtime Architecture Guide*, May 2001. `http://developer.intel.com/design/itanium/family`.

[9] Intel Corp. *Intel Itanium 2 Processor Reference Manual*, May 2004. `http://developer.intel.com/design/itanium/family`.

[10] Intel Corp. *Vanderpool Technology for the Intel Itanium Architecture (VT-i) Preliminary Specification*, Jan 2005. `http://www.intel.com/technology/vt/`.

[11] L4Ka Team. L4Ka::Pistachio kernel. `http://l4ka.org/projects/pistachio/`.

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

[13] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Herrman Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for extensibility). In *Proc. 6th HotOS*, pages 28–31, Cape Cod, MA, USA, May 1997.

[14] Daniel J. Magenheimer and Thomas W. Christian. vBlades: Optimised paravirtualisation for the Itanium processor family. In *Proc. 3rd Virtual Machine Research & Technology Symp.*, pages 73–82, 2004.

[15] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proc. 1996 USENIX Techn. Conf.*, San Diego, CA, USA, Jan 1996.

[16] David Mosberger and Stéphane Eranian. *IA-64 Linux Kernel: Design and Implementation*. Prentice Hall, 2002.

[17] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In *Proc. 5th OSDI*, Boston, MA, USA, Dec 2002.

[18] Open Source Development Labs. *Database Test Suite*. `http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_suite`.

[19] OpenGroup. The Single UNIX Specification version 3, IEEE std 1003.1-2001. `http://www.unix-systems.org/single_unix_specification/`, 2001.

[20] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Comm. ACM*, 17(7):413–421, 1974.

[21] Ryan Rakvic, Ed Grochowski, Bryan Black, Murali Annavaram, Trung Diep, and John P. Shen. Performance advantage of the register stack in Intel Itanium processors. In *2nd Workshop on EPIC Architectures and Compiler Technology*, Istambul, Turkey, Nov 2002.

[22] John W. Sias, Matthew C. Merten, Erik M. Nystrom, Ronald D. Barnes, Christopher J. Shannon, Joe D. Matarazzo, Shane Ryoo, Jeff V. Olivier, and Wen-mei Hwu. Itanium performance insights from the IMPACT compiler, Aug 2001.

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

[24] 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, Mar 1992.