Enhancing IA-64 Memory Management

Alan Au and Gernot Heiser
School of Computer Science and Engineering
University of New South Wales
Sydney 2052, Australia
{alanau,gernot}@cse.unsw.edu.au

December 14, 2000

Abstract

IA-64 is Intel Corporation's recently released 64-bit architecture. It includes features such as data/control speculation, instruction predication and a large number of parallel resources. Also included is a novel memory management unit that allows orthogonal translation and protection mechanisms to be used. This flexibility opens up opportunities for improved memory management techniques. This paper presents our enhancements to IA-64 Linux memory management with focus on improving effective TLB coverage.

1 Introduction

Operating systems that provide virtual memory typically implement protection by placing each process into a seperate address space. In such systems, a process's context includes the set of address mappings valid for that process. Possession of a mapping allows a process to both translate an address as well as give access to that address with the permission rights of the mapping. The *translation lookaside buffer* (TLB) is a hardware cache of these mappings.

TLBs can be either tagged or untagged. An untagged TLB, such as that found on the Intel Pentiums, requires that the TLB be fully flushed, a potentially expensive operation, on each context switch (Liedtke has demonstrated an optimisation to this for small address spaces [8]). Tagged TLBs employ context identifiers which can be associated with each process and are used in addition to the virtual page number to match an entry in the TLB. This negates the need for a TLB flush per context switch but still requires duplicate TLB entries for essentially same mappings in different contexts/processes.

Past studies [1,2] have shown that TLB handling costs can take up a significant part of an application's processing time. TLB coverage is one of the major factors in determining the TLB miss rate and hence the impact of TLB costs on application performance [2]. TLB miss handling overhead is turning into a bottleneck with real memory sizes increasing at a rapid rate, while TLB sizes are remaining essentially constant [11].

TLB coverage refers to how much memory the TLB can map. This in turn is directly related to the number of TLB entries and the page size of each TLB mapping. More indirect but equally important factors in determining *effective* TLB coverage are TLB associativity, application access patterns and the degree of multitasking. This paper addresses the last factor.

Because traditional memory management techniques require a seperate set of mappings per process, there will be high contention for the limited number of TLB slots under heavy multi-tasking conditions. Our idea is to reduce this contention and increase effective TLB coverage by allowing processes to share a subset of their TLB entries with some or all other processes. This approach is made possible by IA-64's ability to handle address translation and protection in an orthogonal manner. This means that it is possible to define the context of a process by a set of address mappings together with set of protection regions.

The following section looks at alternative approaches to increasing TLB coverage as well as other related work. Section 3 describes the IA-64 memory management features and explains how we use them to maximise TLB sharing. Section 4 details the modifications made to original IA-64 Linux to implement our TLB sharing scheme. Section 5 summarises our experiences and results.

2 Related Work

To date there are surprisingly few approaches to improving effective TLB coverage. We do not look at pure hardware considerations here, such as more entries in the TLB and higher TLB associativity. It is obvious that these two methods will improve TLB performance at the expense of silicon estate and decreased address translation speed.[1] The majority of approaches can be classified into two groups: those that directly influence TLB management and those that try to structure kernel and user TLB accesses for maximum performance.

In the first category are studies that have explored different page table structures, page sizes and superpages. Elphinstone [4, 5] has done extensive studies on how different page table structures influence TLB and overall system performance. His work was directed at 64-bit systems and sparse address spaces, concluding that such systems will receive maximum performance with hashed or guarded page tables and a software TLB cache.

Superpages increase TLB coverage by coalescing TLB entries mapping several smaller pages into a single larger mapping entry. The reason that superpages have not been used more extensively is that they have a few requirements that make them either difficult or unattractive to use. There have been several proposals to tackle some of the drawbacks of superpages. Online superpage promotion [10]

[1] Increased hardware resource benefits flatten out once typical working sets can be fully mapped in the TLB.

tries to dynamically predict the likely benefits of promoting to a superpage against the cost of doing the promotion to determine when and where to use superpages. A technique called *shadow memory* [12] has been put forward to mitigate the superpage requirement of large contiguous physical regions of memory which are difficult to manage and have potential adverse swapping effects.

The last two superpage designs work on average but can encounter situations where performance is worse than without incorporation of their superpage technique. For example, a mispredicted online page promotion can cause unnecessary copying of multiple base pages that are never used again. In contrast, our TLB sharing mechanism will never exhibit behaviour worse than the base case without using it. It must also be noted that all of the above TLB optimisations will work even if our sharing mechanism is employed though at this point it is unclear if the benefits will be diminished in a combined scheme.

The only other work that has been done to allow TLB entries to be shared is a modified TLB technique that employs *common-masks* [7]. Khalidi and Talluri describe additional TLB logic that can implement a matching function applied to context strings. Such a function would allow multiple contexts to match the same TLB entry. Thus their goal to enable TLB sharing is identical to ours. The main difficulty with their approach is to choose the comparator function that allows multiple contexts to be masked to a shared context. They specifically do not address this issue in a general manner but do provide an example implementation that allows a limited number of *common-regions* to be shared.

3 IA-64 Memory Management

This section provides an overview of the IA-64 memory management unit. For a full description, refer to the Intel documentation [6].

The IA-64 address space is split into eight equally sized *regions*. The most significant three bits of each 64-bit virtual address forms the *region number* and identifies the region that the address belongs to. Figure 1 is a schematic of the IA-64 address translation process.

IA-64 provides a *region register* for each region. These each contain 24-bit *region identifiers* that are effectively TLB tags. An address's region number is used to index one of the region registers to obtain a region identifier. This region identifier is used together with the virtual page number to search for a match in the TLB. A successful match yields a physical frame number and access permissions. So far, the translation process is not much different to other memory management units. But the IA-64 TLB entries contain an extra *protection key* field. The protection key is used to match against a set of *protection key registers*. Each protection key register contains a protection key and a set of access qualifiers that are applied to each translation that matches this protection key. In this way, IA-64 memory management allows for orthogonal address translation and protection.

We now present the basic principles of our TLB sharing scheme using the IA-

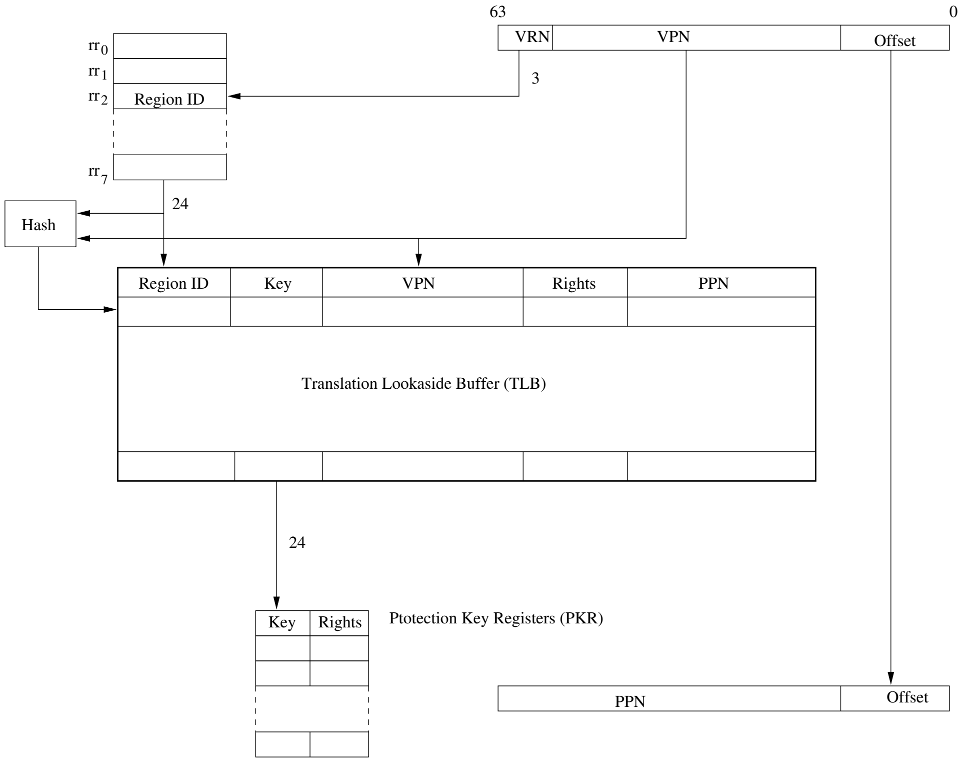

Figure 1: IA-64 Address Translation

64 MMU.

1. Executables are traditionally linked to a fixed address. As not all processes share the same executable, the protection keys are of not much help here. Instead we will reserve one region register to map a process' text and read-only data segments. The system must then allocate a unique region ID to each executable currently in use, which is used to tag a process' TLB entries for text and read-only data. On a context switch, the region register is reloaded with the ID of the new process' executable region.

2. Two approaches are possible for supporting *dynamically linked libraries* (DLLs). Each library can be given its own region ID (the architecture supports a minimum of 2^{18} IDs) and each program using a DLL loads one of the region registers with the DLL's region ID. The problem here is that there are only 8 region registers, and at least two of them are required to access program text and data. This would limit the number of DLLs that can be used. Since the VRN is part of the virtual address, a particular DLL must always use the same region register for the duration of the process' execution. Hence sharing region registers between DLLs cannot easily be done and is

likely to result in significant overhead (region-register thrashing).

Alternatively, protection keys can be used. We assign a unique key to each DLL presently in use (there are at least 2^{18}). One region ID is reserved for use by **all** DLLs, and one region register is reserved to always contain that ID. The rights field of TLB entries for DLLs are set to read-only (R/O) for the R/O data pages and execute-only for text pages. If protection keys were disabled this would make the library accessible to every process. The keys are used to restrict access to those processes which have actually linked the library: Upon linking, the DLL's key is entered into a protection key register. These registers are part of the process context and are saved and restored at process switch time.

This approach requires that DLLs are allocated at non-overlapping virtual addresses. This effectively introduces a flat, single address space for DLLs. As each DLL is mapped at an address which does not change during its lifetime, this approach also removes the requirement for generating *position-independent code*, and we have recently shown that this in itself leads to measurable performance gains [3].

3. Cloning of a data segments can be implemented by giving it a unique region ID which is loaded into the region register of parent and child. (Note that the hardware offers no improvements for copy-on-write pages, as created by forking. Parent and child have the same access to copy-on-write pages, hence protection keys do not help. Domain IDs do not help either, as the virtual address must not change when the real copy operation is performed.)

4. Memory-mapped files and other shared regions created by mmap() cannot be supported by region IDs, as access rights differ, in general, between processes mapping the same object. Protection keys can be used as in the case of DLLs. As with DLLs, TLB real estate can only be saved if the objects are mapped at the same address for each participating process. This is supported by Linux' mmap() semantics. In fact, DLLs in Linux are implemented via mmap(), so TLB sharing support for DLLs will automatically benefit other uses of mmap().

Similar mechanisms can be used to speed up other communication mechanisms, such as *pipes*. This can be efficiently implemented via a shared buffer to which one process has write permission and the other does not. Protection keys are designed to support access to shared data (mapped to the same address) where the participants have differing access rights. The sequential nature of pipes provides scope for optimising TLB use further, e.g., by explicitly discarding TLB entries no longer needed, or wrapping the buffer as soon as a page has been read.

4 Linux

This section describes the state of our Linux modifications.

4.1 Process Contexts

Each Linux process is assigned a context number of the format shown in figure 2.

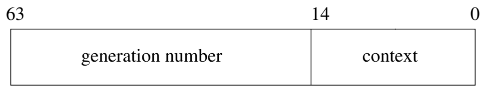

Figure 2: Linux Context Number Format

The *generation* is to ensure that old contexts are not used in the event of context wraps. Reloading a process' context is a simple matter of using the context number to form the region identifiers needed for each region register. To support TLB sharing of text, an extra *text context* of the same format as the general context number is added to process contexts. Reloading a process' context uses the general context number as previously but reloads the text region register using the text context.

4.2 Page Table Structure

IA-64 provides an optional hardware page table walker. The page table entries can be in one of two formats. Figure 3 shows the *short format* and Figure 4 the *long format*.

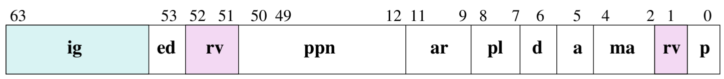

Figure 3: PTE - Short Format

The short format is used in IA-64 Linux as it is well suited to the Linux 3-level page table structure. The L3 page table pages are mapped into a virtual inner array to form the hardware page table. Thus, the hardware and software page tables are always automatically in sync.

On a TLB refill, the physical frame number is obtained directly from the short format page table entry but other values, in particular the page size and the protection key value (if used), are taken from the region register of the faulting address. Our scheme requires finer control over protection key values than this allows so we adopt the long format approach.

With the long format, the hardware walker treats the page table as a hash table. So to use the long format, there are two options:

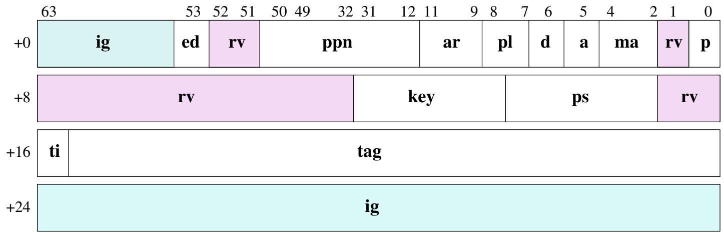

Figure 4: PTE - Long Format

1. Keep Linux's underlying 3-level page table and maintain the hardware page table separately.
2. Modify the software page table to better fit the long format.

We intend to adopt the first approach as this initially requires substantially less code change. But at this stage, the hardware page table has been effectively disabled by never inserting a page table page mapping into the TLB.[2]

4.3 Address Space

The following table (taken from [9]) shows how the process address space is allocated.

Region	Current Usage	Page size	Scope	Mapping
7	cached	large (256MB)	global	identity
6	uncached	large (256MB)	global	identity
5	vmalloc	kconfig (8KB)	global	page table
4	stack	kconfig (8KB)	process	page table
3	data	kconfig (8KB)	process	page table
2	text	kconfig (8KB)	process	page table
1	shared memory	kconfig (8KB)	process	page table
0	IA-32 emulation	kconfig (8KB)	process	page table

When a process is executed, the following are some of the tasks which must be performed by the kernel and the runtime loader to map the new process and its DLLs into the address space.

The kernel's tasks include:

- Parse the executable program headers and map in the load sections.
- Map in the runtime loader.

[2]The hardware page table is not actually disabled because that would cause a different set of exceptions to be raised, requiring more of the interrupt vector code to be modified.

The runtime loader's tasks include:

- Relocate itself.
- Map in shared libraries into the address space.
- Read the relocation information and apply the relocations.

As the table above shows, executable text and data are mapped into separate regions. What may be less obvious is that shared library text *and* data are all mapped into the shared region by the runtime loader. Recall from section 3 that our TLB sharing scheme involves reserving an entire region for shared libraries with the same region identifier for all processes. The obvious thing to do would be to use the existing Linux shared region (region one) for this purpose. To do this, one of two options can be used:

1. Allow shared library code and data to remain in the shared region. The implications of this for our scheme would be the need for two protection keys per shared library; one for data and one for code.
2. Purify the shared region so that only truly sharable address regions are found there. That is, remove the shared library data from the region.

The later has been chosen. This involved modifying the runtime loader so that shared library data is loaded into the data region while shared library code is retained in the shared region. The relocation code was also modified to accommodate this new address space arrangement.

Also recall that it is necessary in our design to ensure that shared libraries do not overlap. We have modified the `mmap()` code to return unique addresses for file mappings. The first time a file is mapped, it is allocated a new unique address in the shared region. Subsequent mappings of the same file, even by different processes, will use the previously allocated address.

4.4 Protection Keys

At this stage, protection keys have been enabled but proper key management has not been implemented. The first thing we wanted to establish was that the IA-64 protection key mechanisms do indeed function correctly (at least in the simulator but later for real hardware as well). As described in section 4.2, original Linux uses the short format page table and only needs 8 bytes per entry. To use protection keys, an extra 8 bytes is needed to store the second long word of the long format entry. So instead of allocating one page per L3 node, two pages are used. The first page is just the original L3 node (ie. contains physical frame number). The second page contains the protection key. Entries with the same offset in the first and second pages correspond to the same mapping.

5 Conclusion

This paper has presented the design and initial implementation of a scheme to increase effective TLB coverage by maximising the sharing of TLB entries. This has been made possible by IA-64's MMU features which allow address translation and protection to be decoupled.

There is still much work to done. Some of the things that may be looked at as future work include:

- Protection key management to minimize context switch and reload times. There are a minimum of 16 protection key registers which should be enough to hold the protection domain of most processes without replacement but this should be analysed more formally. Appropriate replacement algorithms should be developed if necessary.
- Per region page tables can be considered for the different levels of sharing. Figure 5 shows an example of this.

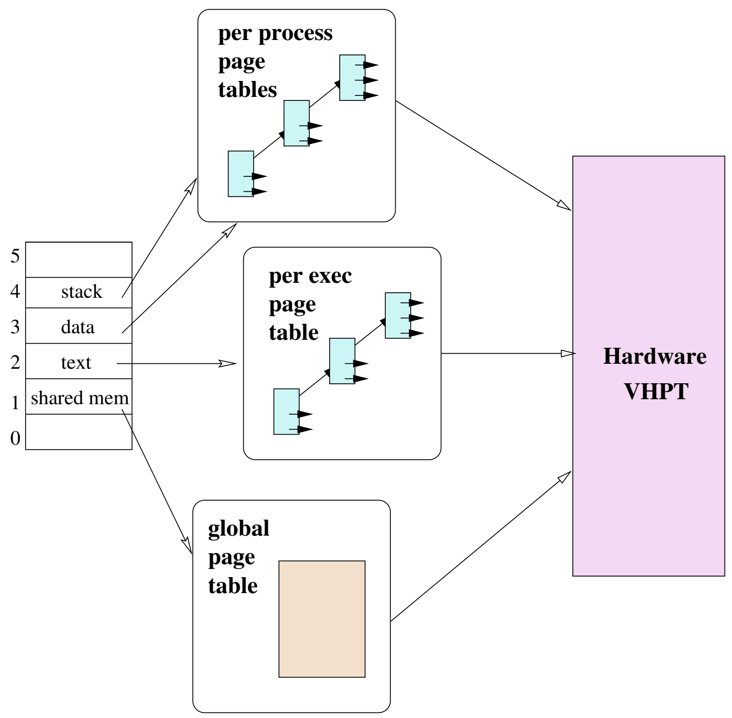

Figure 5: Per Region Page Tables

- Benchmarking the system to quantify the performance benefits. The TLB sharing scheme outlined should have maximum benefits under high multi-tasking loads, high context switch rates and heavy TLB usage. This needs to be confirmed.

References

[1] 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.

[2] 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] Luke Deller and Gernot Heiser. Linking programs in a single address space. In *Proceedings of the 1999 USENIX Technical Conference*, pages 283–294, Monterey, Ca, USA, June 1999.

[4] Kevin Elphinstone. *Virtual Memory in a 64-bit Microkernel*. PhD thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, March 1999. Available from http://www.cse.unsw.edu.au/∼disy/papers/.

[5] 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/.

[6] Intel Corp. *IA-64 Architecture Software Developer's Manual Volume 2: IA-64 System Architecture*, January 2000. URL http://developer.intel.com/design/ia-64/index.htm, order no 245318-001.

[7] 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.

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

[9] David Mosberger and Don Dugger. IA-64 linux kernel internals. URL http://www.linuxia64.org/, February 2000.

[10] Theodore H. Romer, Wayne H. Ohllrich, Anna R. Karlin, and Brian N. Bershad. Reducing TLB and memory overhead using online superpage promotion. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*, pages 176–87, Santa Margherita Ligure, Itay, June 1995. ACM.

[11] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. Recency-based TLB preloading. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, pages 117–127, Vancouver, Canada, June 2000. ACM.

[12] M Swanson, L Stoller, and J Carter. Increasing tlb reach using superpages backed by shadow memory. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA)*, pages 204–213. ACM, 1998.