Transparent Large-Page Support for Itanium Linux

Master of Engineering
School of Computer Science and Engineering
The University of New South Wales

Ian Wienand

July 14, 2008

Abstract

The abstraction provided by virtual memory is central to the operation of modern operating systems. Making the most efficient use of the available translation hardware is critical to achieving high performance. The multiple page-size support provided by almost all architectures promises considerable benefits but poses a number of implementation challenges.

This thesis presents a minimally-invasive approach to transparent multiple page-size support for Itanium Linux. In particular, it examines the interaction between supporting large pages and Itanium's two inbuilt hardware page-table walkers; one being a virtual linear page-table with limited support for storing different page-size translations and the other a more flexible but higher overhead hash table based translation cache.

Compared to a single-page-size kernel, a range of benchmarks show performance improvements when multiple page-sizes are available, generally large working sets that stress the TLB. However, other benchmarks are negatively impacted. Analysis shows that the increased TLB coverage, resulting from the use of large pages, frequently does not reduce TLB miss rates sufficiently to make up for the increased cost of TLB reloads. These results, which are specific to the Itanium architecture, suggest that large-page support for Itanium Linux is best enabled selectively with insight into application behaviour.

Originality Statement

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

Ian Wienand, July 14, 2008

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Virtual memory underpins the operation of all modern general-purpose computing. The abstraction of a virtual address space frees the programmer from many difficult and time consuming memory management tasks and allows the operating system to implement security and policy.

The smallest unit of memory available is a *page*. In basic operation, each virtual page corresponds to a physical *frame* of the same size. Addresses within a virtual page are translated by hardware to the underlying physical address when accessed.

To find the virtual page to physical frame translation information, the hardware has a cache termed the *translation lookaside buffer* (TLB). The TLB, being a cache, holds only a limited number of entries and hence a *page table* is kept by the operating system to store and manage all virtual-to-physical translations. Filling TLB entries on a miss can either be done by the hardware autonomously navigating these page tables (*hardware-loaded TLB*), an exception which invokes the operating system (*software-loaded TLB*) or some combination of both.

Traditional operating systems use a fixed base page-size as the lowest common denominator; i.e. all memory allocations, no matter how large, are decomposed to a fixed size. This is generally a software limitation, since most architectures have a TLB which supports translations with varied page sizes.

The operating system generally utilises a base page-size somewhere between 4 KiB and 64 KiB. With a modern system containing gigabytes (or more) of physical memory, this results in millions of pages and frames requiring management. With such a large amount of translations to keep track of, overheads from TLB misses and consequent refill costs can easily become a bottleneck to system performance.

TLB *coverage* refers to how much virtual-memory address space can be translated by the limited number of TLB entries; more is universally better (for example, a TLB able to hold 16 translation entries for 4 KiB pages covers 64 KiB). Universally increasing the base page-size increases coverage, however this leads to unacceptable trade offs with internal fragmentation. We are therefore left to consider supporting translations with differing page sizes within the TLB.

1.1 Motivation

The most efficient way to maximise TLB coverage is to choose a page size for each translation as closely sized to the memory allocation it represents as possible (e.g., if the program requests 1 MiB of memory it

is mapped with a 1 MiB page). Ideally, this support should be provided in a manner transparent to the application programmer as part of the abstraction layer provided by the operating system.

General purpose operating system support for multiple page-sizes varies. The major commercial UNIX implementations of OpenVMS [NK98], HP-UX [SMPR98], IRIX [GS98] and Solaris [McD04, Low05] have support for multiple page-sizes in some form. Linux currently provides for non-transparent, pre-allocated large pages; alternative approaches for transparency have been proposed but have so far not gained acceptance.

1.1.1 Itanium

Since the TLB is implemented in the processor hardware, effective large-page support is very dependent on the target architecture. Intel's Itanium architecture promises the potential for greatly increased translation coverage by providing a TLB with support for a very wide range of page sizes.

To provide for drastically lower translation refill times than relying on software alone, the Itanium also provides the ability to hardware load translation entries via two different execution models. Each mode requires translations in a particular fixed format suitable for the hardware to read and refill the TLB with.

1.2 Research Questions

Therefore effective large-page support for Itanium Linux poses the following questions:

1. How should the operating system transparently choose an appropriate page size when allocating memory?
2. What are the costs and benefits of modifying the operating system to enable large-page support with each of the two forms of hardware translation loading? Ultimately, which is better?

1.3 Contribution

Prior work has investigated large-page support schemes for Linux but until now none had been ported to the Itanium architecture. While the properties of the Itanium virtual memory implementation have been examined in prior work [CWH03, GCC+05] the interaction with large-page support has not been considered. The management of multiple page-sizes with the Itanium architecture has been studied with FreeBSD [Nav04], however the work did not pay particular attention to the processor's ability to hardware load translations.

This thesis therefore provides a unique view of the interaction between Linux large-page support and the Itanium memory management architecture.

1.4 Thesis Overview

The following is an outline that maps out the rest of this document:

Chapter 2 draws foundations for implementation choices from literature and prior work.

Chapter 3 introduces and describes the operation of the Itanium memory management unit (MMU) and examines some of its unique features.

Chapter 4 evaluates the tradeoffs of different Itanium MMU configurations.

Chapter 5 discusses the implementation of a transparent large-page scheme for Itanium Linux.

Chapter 6 evaluates the models presented.

Chapter 7 concludes and describes future work.

Chapter 2

Foundations

This chapter describes the foundations of the work presented in this thesis. We describe the problems with supporting multiple page-sizes, examine various solutions from prior work and draw conclusions for our implementation. A more extensive review can also be found in the associated literature review [Wie06].

Section 2.2 and Section 2.3 introduce the hardware and software constraints on supporting multiple page-sizes. Section 2.4 presents a categorisation and examination of exiting approaches. Section 2.5 presents the conclusions drawn from the examination of prior work.

2.1 Multiple Page-Sizes

It has long been identified that TLB size is not keeping pace with the ever growing amounts of memory attached to a modern system [Nav04]. Finding methods to expand TLB coverage is therefore a pressing issue for a modern systems.

Clearly, the easiest way to increase TLB coverage would be to simply design a TLB with more translation entries! As might be expected, practicalities make this approach extremely difficult. The TLB has a critical role in the processor pipeline and is required to process lookups in usually as little as one or two cycles. The fully-associative content-addressable memory (CAM) used to facilitate such low latency lookup is very expensive in terms of both transistor count and power usage and does not scale linearly.

It is therefore useful to be able to use varying page sizes within the limited TLB translations available. However, supporting multiple page-sizes raises issues for both hardware and software designers.

2.2 Hardware Constraints

In a single page-size system any virtual address can unambiguously be split into a *virtual page number* (VPN) and *offset*. The VPN is presented to the TLB for translation to the underlying physical page and the offset is then added to the physical page to construct the final physical address.

With multiple page-sizes a given virtual address no longer uniquely identifies a VPN. The split between VPN and offset bits now depends on what size page the given address is currently mapped within [TKHP92,

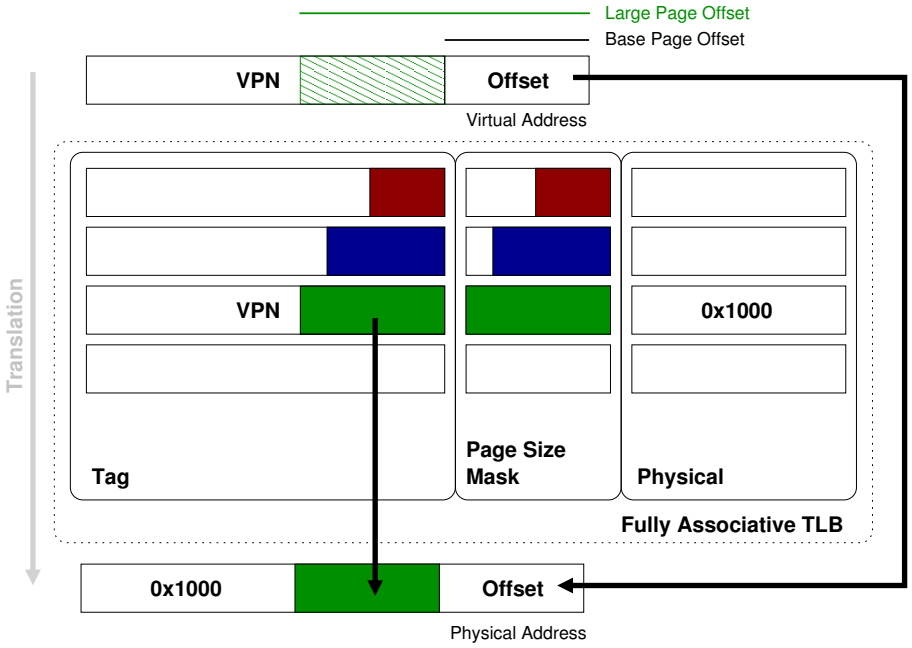


Figure 2.1: A fully associative TLB can be easily extended for multiple page-size support by adding a mask field to eliminate the offset bits for the physical address. Larger pages have more bits of the mask set, whilst a base page size has no bits set.

Sez04]. For example, virtual address `0x12345678` would have a VPN of `0x12345` with 4 KiB pages or VPN `0x1234` with 64 KiB pages.

In a fully-associative TLB the VPN in each TLB entry is checked for a match in parallel. Multiple page sizes can be implemented by extending each entry with a mask field specifying the VPN/offset split, as illustrated in Figure 2.1. A fully associative TLB requires a large bank of CAM and is thus limited in size. A larger TLB is usually implemented via *set-associativity*.

Set-associativity, illustrated in Figure 2.2, separates the TLB into several *ways* which each hold a portion of the TLB entries. This is done because it is more practical to create several smaller areas of CAM (the ways) rather than one very large array. At translation time a number of bits of the virtual address are used to index into each way; the entry at this index is then checked in each way, in parallel, for a match. Thus the parallel component of the lookup is restricted to the number of ways rather than the total number of entries as in a fully-associative cache. The index into the way must be known before the lookup can start; when presented with only a virtual address the TLB has no information to distinguish the page size and hence no way to find the split between offset and index bits. This results in a "chicken and egg" problem; the page size is used to index the TLB, but the page size is kept in the TLB.

Overcoming these problems has been the focus of much research. Models proposed include sub-blocking TLBs [HS84, TH94], "skewing" [Sez93, Sez04] and "Zip Coding" [Lie96]. Other approaches such as software managed address translation [JM97] and in-cache address translation [WEG+86] suggest avoiding the problem by removing the TLB altogether. None of these schemes have found large scale commercial implementations (a notable exception is MIPS, which implements a form of sub-blocking TLB). Current processors tend to implement multiple page-size support with either a small, fully-associative TLB (StrongARM, Alpha, UltraSPARC T1) or multiple TLBs each holding a different page size (UltraSPARC, IA-32). Itanium differs slightly by tying the first level cache and a small upper level TLB together closely with fixed-sized "pre-validated" entries (for details see [Lyo05]) but keeps a fully-associative second level TLB.

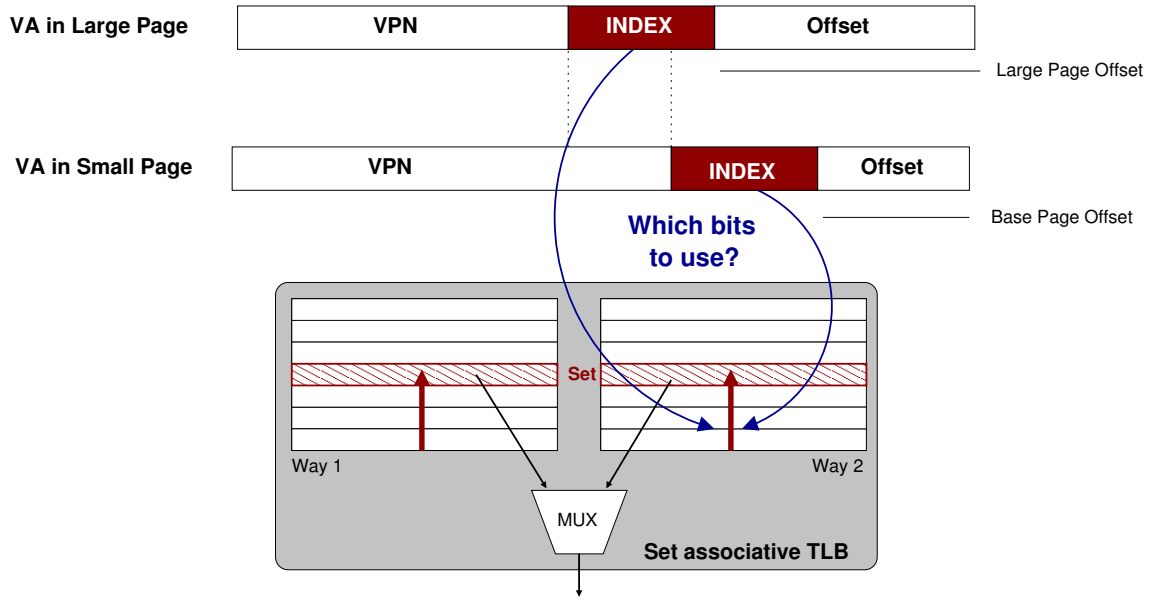


Figure 2.2: An illustration of issues arising from supporting multiple page-sizes in a set associative TLB. A set associative TLB separates the TLB into *ways*; an index is taken from the virtual address, and the entry at this index in each way is checked simultaneously for a match. This creates a classic "chicken and egg" problem between the index and page size.

2.3 Software Constraints

Supporting multiple page-sizes is not only problematic for hardware designers, but raises a number of issues for software designers.

2.3.1 Fragmentation

The tradeoffs of an increased base page-size have been evident since the first virtual memory implementations:

> There is a page size optimal in the sense that storage losses are minimized. As the page size increases, so increases the likelihood of waste within a segment's last page. As the page size decreases, so increases the size of a segment's page-table. Somewhere in between the extremes of too large and too small is a page size that minimizes the total space lost both to internal fragmentation and to table fragmentation. (Denning, 1970 [Den70])

In the quote above, Denning is referring to the concepts of *fragmentation*. If a page of memory is not fully used due to the object it stores being smaller than the page size, the left over and unusable space is lost to *internal fragmentation*. If the page size is reduced so to is internal fragmentation, however allocations become more scattered in memory and end up creating many small holes of free memory. *Contiguous* memory refers to allocations of consecutive physical frames; a virtual page of a given size must be backed by contiguous physical memory of the same size. Many small gaps therefore inhibit contiguous allocation, a situation referred to as *external fragmentation*.

Wilson et al. [WJNB95] identify fragmentation in general as "an inability to reuse memory which is free". They further identify the root cause of external fragmentation as placing objects with dissimilar lifespans in adjacent areas. The slab allocator [Bon94] is a caching allocator for objects which attempts to ameliorate

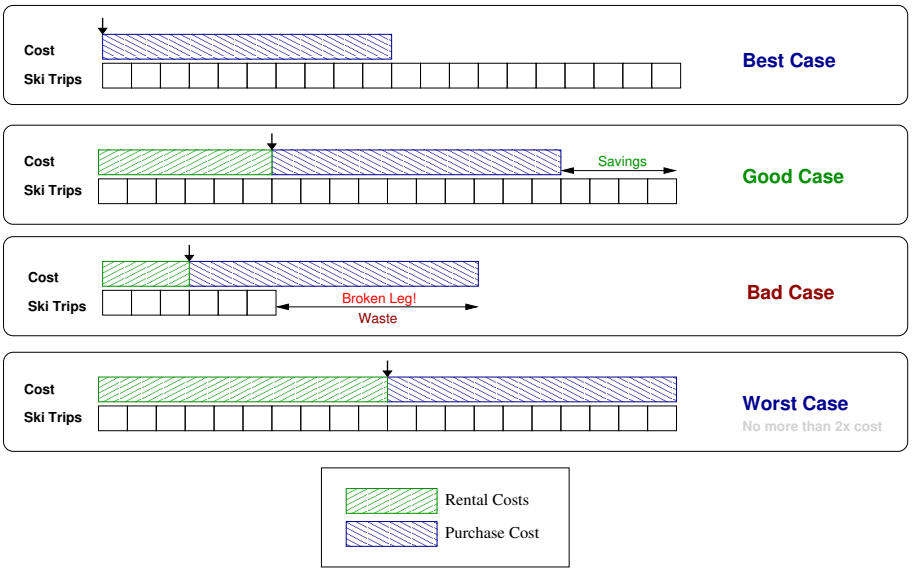


Figure 2.3: The *ski-rental problem* echos issues with page promotion [ROKB95]. When should the skier take the fixed cost of upgrading from renting to purchasing? For an operating system, when does the cost of making a large page outweigh the benefits?

these effects. By pre-allocating a region of memory for a number of objects (a *slab*) reallocation after deallocation reuses the same memory, keeping objects together and fragmentation to a minimum. The requirement of a known object size makes this policy unsuitable for a general purpose memory allocator. Other schemes rely on the OS being able to move physical frames to create areas of free contiguous memory [GS98, GW07].

2.3.2 Complexity

Managing multiple page-sizes necessarily adds overheads beyond simply supporting a single page-size. One major overhead is deciding if and when to promote a base page to a larger page size. This problem falls into the general category of "competitive algorithms" [MMS90], one of the more famous being the "ski-rental" problem [Kar92]:

> Consider a novice skier. Ski rental is $10 per day, but to purchase the same skis would be $100. Should the skier rent or buy?

An *optimal off-line* policy would have the skier purchase the skis if they were sure to ski 10 or more days. However, given the novice cannot know this before going skiing, they must use an *online* policy with a *threshold* to decide when to make their purchase. Some complexities of this situation are illustrated in Figure 2.3.

Romer et al. [ROKB95] examine schemes for promoting base pages to a larger page-size (i.e. "buying the skis") by tracking page usage in an *online* fashion. In summary, the scheme records TLB misses against a large page that, if mapped, would have prevented them. When a certain threshold of preventable misses is met the large page is instantiated in the system. Keeping complete counters is an expensive proposition; figures of multiple thousands of cycles per TLB miss are described. Romer shows that the overhead of a close-to-optimal solution are too great in practice and proposes more approximate solutions which require less measurement. Subsequent work by Fang el al. [FZC+01] found that on a superscalar machine a low

instructions per cycle (IPC) fault handler can have a large negative effect on a high IPC application — since the OS trap generally clears the processor pipeline, hardware resources are wasted as the less efficient miss handler runs. They also took into account other side-effects such as cache pollution caused by increased time spent in the fault handlers. The work reinforces the need for simplicity in managing large pages.

These overheads can become quite large and add latency to time-critical virtual memory fault paths. Thus the implementation of multiple page-size support must focus on ensuring any slowdown in fault handling is made up by an overall reduced number of faults thanks to greater TLB coverage.

2.4 Software Policy Categorisation

We can divide page size polices into three groups [Szm00, Wie06]

1. **Global policies** use fixed-sized large pages, either preselected before mapping or chosen closer to runtime creation of a mapping. Larger pages are never *promoted* (grown) or *demoted* (shrunk).
2. **Static policies** are categorised by a "best effort" promotion and demotion strategy which can grow and shrink page sizes, but will not copy pages to smaller or larger physically contiguous regions to achieve this.
3. **Dynamic policies** implement promotion and demotion, copying frames to be physically contiguous when appropriate.

Below we classify existing implementations within these categories.

2.4.1 Global Policies

Pinning can be used to map a large, static region such as kernel code or a frame buffer with a single translation entry. Each pinned entry is exempt from normal refill policy, which usually chooses entries to remove based on frequency of access. Thus a pinned entry is suitable for a frequently accessed, static region, but is not a general purpose solution.

Another global policy is to increase the base page-size, which leads to a corresponding increase of TLB coverage. With very large memory systems and long running processes small losses to fragmentation are generally not a major concern for running applications. There are, however, subtleties that can arise from an increased page size. For example, in a modern operating system unused physical memory is used as a cache for disk storage. Any file with a size not exactly modulo the page size will have a page holding the *file tail* which will suffer from internal fragmentation. For many applications accessing many small files, this can cause a significant decrease in effective disk cache size [KP06].

The current Linux approach to large-page support is HugeTLB which separates a pre-allocated area of memory at early boot to be mapped with a single larger page size. Application programmers must either use special `mmap` flags or open a file on a specially mounted file system to access the large-page backed memory (a full explanation is provided in Section 3.5). This scheme has the advantage of reduced complexity — on a fault the system need only check if the faulting address is in the large-page region to determine the page size. It is not, however, transparent, requiring both API and ABI changes.

Winwood et al. [WSF02] propose a more fine grained approach based upon modifications to the current Linux virtual memory model, presented in Figure 2.4. Each `vm_area`, which represents an object in a

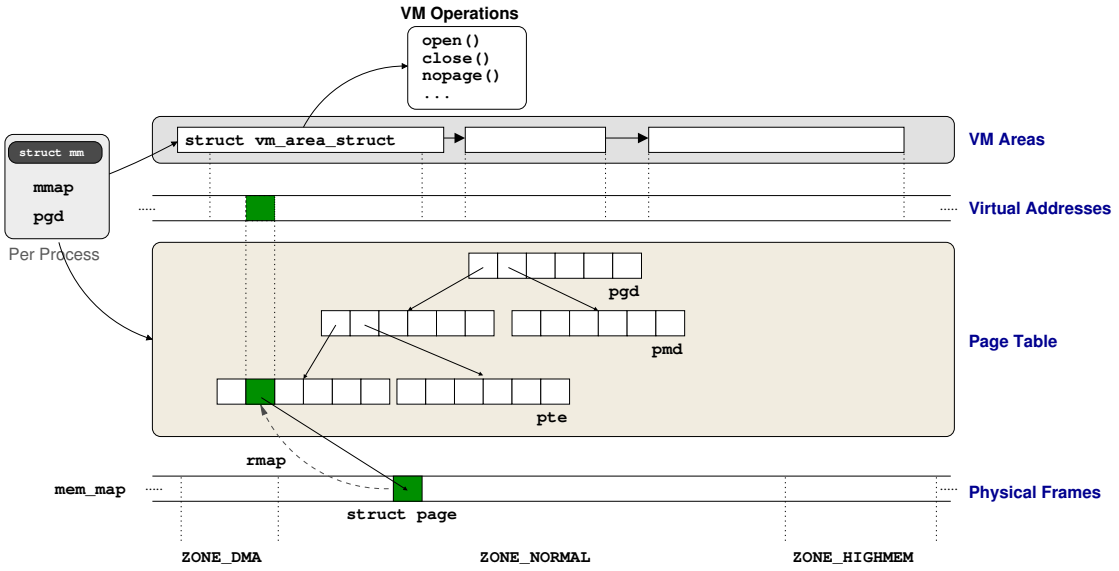


Figure 2.4: Linux memory layout [WSF02]. Allocations within a processes virtual address space are described by a `vm_area_struct`. Pages within these areas are mapped by a page-table to the underlying physical pages tracked in the `mem_map` array.

processes virtual memory, is extended to contain a page size. Lower layers are modified to find and use the page size where appropriate. Modifications were also made to allow `pgd` and `pmd` levels of the page-tables (shown in Figure 2.4) to contain translations; this results in extra complexity when walking the page table as each entry must be checked to see if it is a pointer or a translation, but allows for potentially shorter paths. To avoid problems with external fragmentation Winwood et al. use a special pre-allocated zone. Thus as implemented, the approach comes under a global classification, however the general principles could be expanded into a more dynamic system. Page size is set via calls to `madvise`, meaning there is a programmer visible API.

Support for large pages in Solaris 2.6 through Solaris 8 was via a specialised form of System V shared memory referred to as *intimate shared memory* (ISM) [McD04]. This is similar to the Linux HugeTLB concept; shared memory is requested as "intimate" via a flag to `shmat()`. Unlike Linux, however, processes sharing ISM areas share page table entries as well. *Dynamic ISM* (DISM) was added to Solaris 8 (Update 3) to allow dynamic resizing of ISM areas; particularly useful for databases which previously required a shutdown-restart cycle to change the size of ISM caches. Solaris 9 expanded ISM to support intermediate large-page sizes. *Multiple page-size support* (MPSS) was introduced with Solaris 9 as a method for allowing applications to request larger pages without needing to use ISM. Like the HP-UX (Section 2.4.2) and IRIX (Section 2.4.3) schemes, MPSS requires an application (or administrator) to request certain page sizes for the application. MPSS support is available via a number of methods including shared library wrappers and flags in binary headers which can be set via compiler options or administrative tools. Current work is looking at making these hints automatically [Low05].

2.4.2 Static Policies

When using a static policy, an initial page-size for a region is chosen based on some heuristics. Static approaches must support demotion to smaller page sizes as this is requirement for correct transparent operation. For example, a common operation is to use the `memprotect` system call to modify the permissions

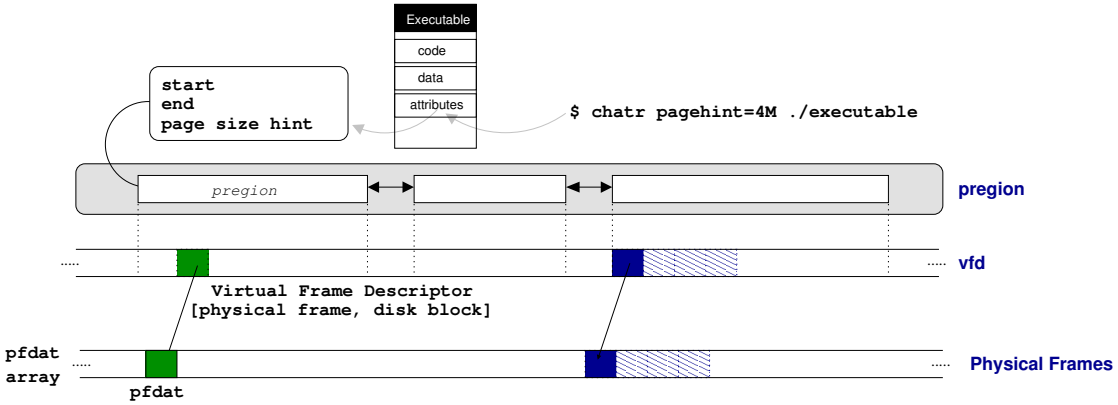


Figure 2.5: A simplified view HP-UX memory management (hardware independent side).

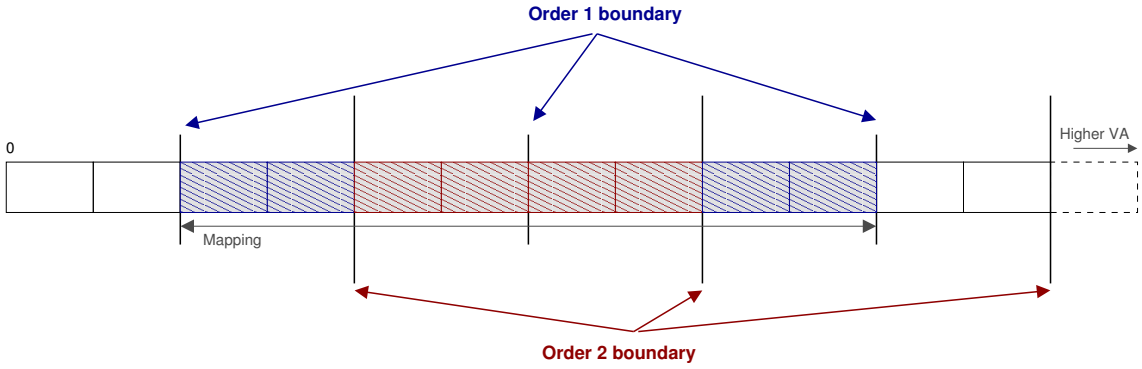


Figure 2.6: Shimizu and Takatori [ST03] size pages on boundaries, choosing the largest size possible within a mapping. Above we see the 8-page mapping is covered by two large pages of order 1, and one superpage of order 2.

on a region of memory. Since protection is enforced by the TLB at the page level, a reduced page-size will be required to maintain correct behaviour. Demotion also allows for graceful handling of insufficient contiguous physical memory and may allow optimisations such as avoiding I/O overheads by splitting large pages when they are evicted to swap. Unlike dynamic approaches, static approaches do not allow for arbitrary growth of a region into a larger page.

Subramanian et al. [SMPR98] implemented multiple page-size support for the HP-UX operating system. As illustrated in Figure 2.5, a *page-size hint* is suggested by an administrator and added to the attributes of a binary executable. This information is stored in the *pregions* (similar to a Linux *vma*, see Figure 2.4) and can be used to select a page size on fault. HP-UX will attempt to fulfil this hint unless there is insufficient contiguous memory or the system is coming under memory pressure. The hinting scheme is also supplemented by transparent hinting mechanisms; for example, heap *pregions* which grow to a large size in small increments are tracked and will have their hints upgraded. Thus an `sbrk` call may receive more memory than is requested (16 KiB instead of 4 Kib, for example) if it has been detected as growing. Hints can also be downgraded under memory pressure to avoid wastage via internal fragmentation.

Rather than modify all virtual memory structures to handle multiple page-sizes, large pages are defined as a group of contiguous base page-size translations. Using this scheme reduces modifications required to the virtual memory layers and is often referred to as *translation replication* (a similar approach was taken with IRIX [GS98]).

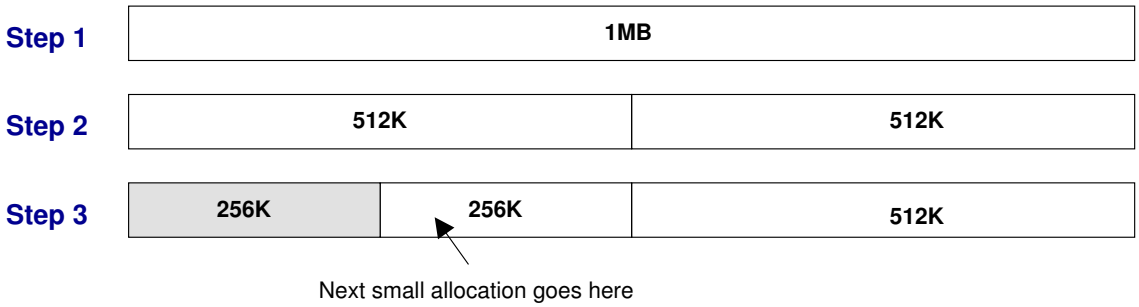


Figure 2.7: The buddy allocator reduces fragmentation by packing allocations close together. Above illustrates the process of allocating 256 KiB from a 1 MiB chunk.

Shimizu and Takatori [ST03] propose a *transparent* multiple page-size implementation for Linux on Alpha, Sparc64 and IA-32. Their approach lets the kernel provide naturally-aligned large pages to cover a given mapping at `mmap` time, as illustrated in Figure 2.6. The frames are marked with their page size when the *virtual memory* is allocated by the system. This means until a page is faulted in, there is no memory backing it. At page-fault time a suitably large region of contiguous memory is requested from the standard Linux memory allocator. If sufficient contiguous memory cannot be found when a page is faulted the large page will be recursively divided and a smaller allocation attempted, until the large page allocation has finally been broken down to the base page-size.

2.4.3 Dynamic Policies

With a dynamic approach, the operating system is able to grow an allocation into a larger page. This policy implies a fundamental change to OS memory management — one of the prime motivators for mapping virtual frames to physical frames is that a contiguous virtual region does not require a contiguous physical region of memory. Below we identify and discuss some of these issues surrounding implementation of a dynamic approach.

Room to grow

A traditional physical memory allocation scheme is the *buddy allocator* [Knu97] which reduces fragmentation by packing allocations together. For example, as illustrated in Figure 2.7, two small 256 KiB allocations will be placed adjacent, leaving contiguous space for further allocations. The disadvantage of this approach is it leaves little room for the smaller allocation to grow. Copying the entire allocation into a larger region is generally not practical because of time overheads and secondary effects such as cache reload.

Keeping a contiguous region of physical memory for a small allocation to grow into implies a *reservation* of a larger amount of memory. Navarro [Nav04] proposed reservation methodologies, but encountered several problems. Firstly, operating systems often use *pinned* or *wired* pages which can not be unmapped and consequently can be the cause of large amounts of external fragmentation. A solution is to modify the system to allocate pinned pages only from a particular known region. Secondly, as mentioned in Section 2.4.1, unused memory is used for disk cache. If the system does not allow reserved memory to be used as cache, overall performance decreases. However, if the reserved memory is used it can cause additional overheads; for example an allocation growing into a dirty cache region needs to wait for the data to be flushed to disk before it may continue.

A background *contiguity daemon* can be run to reclaim contiguity by migrating existing allocations. This is done in IRIX [GS98] with varying levels of aggression depending on the current contiguity requirements. Navarro also proposes algorithms for a contiguity daemon, and shows off-line contiguity reclamation is practical.

Complexity

Dynamic approaches have the most work to do deciding when to instantiate or promote a large page, so they are particularly sensitive to the complexity issues discussed in Section 2.3.2. As mentioned, Romer [ROKB95] and Fang el al. [FZC+01] concluded that simple, low overhead counting schemes (rather than more complex trace-back or probabilistic counting) provided for the best promotion decisions.

Navarro [Nav04] found that an acceptable methodology for dynamic large-page management of the four page-sizes supported by the Alpha processor introduced significant overhead when implemented on Itanium, which can support over 10 different page sizes. He suggested a revised model without these limitations, but it requires the kernel keep a *splay tree* of virtual pages present in the process; something Linux does not provide.

Optimal page-size

To ensure optimal performance it is desirable to allow the operating system to choose from as many different page sizes as the underlying architecture provides. For one benchmark, Navarro found a slowdown of 47% when a process was not able to access the largest page-size supported. Flexibility is also required; Subramanian et al. [SMPR98] found that the largest page-size was not always optimal.

2.5 Conclusions

The analysis of prior work presented in this chapter suggests the following goals for effective large-page support:

- Page size should be transparent to the application programmer. This both avoids programmers requiring knowledge of virtual memory internals and maintains API and ABI consistency.
- Overheads of large-page management need to be kept to a minimum. Online promotion techniques are expensive and a potential bottleneck; therefore a static approach is probably most appropriate [Nav04, ROKB95, FZC+01].
- Large scale modification of the complex Linux virtual memory layers is unlikely to be palatable or practical. Translation replication has been shown to successfully retrofit existing systems and appears the most suitable implementation approach [SMPR98, GS98, ST03].
- Itanium provides a wide range of page size choice which should not be artificially limited by algorithms or data structures which do not scale with the number of available page sizes [Nav04, SMPR98].
- Contiguity can be considered offline [GS98, Nav04].

Thus this thesis focuses on the implementation of a transparent, low-overhead multiple page-size infrastructure for Itanium Linux. The approach is most closely modelled on that of Shimizu and Takatori.

Chapter 3

Itanium MMU

This chapter presents an overview of the Itanium memory management unit (MMU) design and introduces the alternative forms of hardware page-table walker (HPW) provided by the architecture. We then examine how Linux currently supports large pages and how this fits with the architecture.

Section 3.5 introduces the Itanium address space layout. Section 3.2 introduces the available HPW options and Section 3.3 and Section 3.4 explain the operation of the two available modes. Section 3.5 concludes with a discussion of existing Linux large-page support.

3.1 Address Spaces

The main goal of virtual memory is for each running process to have a private and unique view of the virtual address space. The simplest way for the operating system to achieve this is to empty or *flush* all translation entries stored in the TLB when a *context switch* activates a new process (and hence address space). As the newly activated process accesses its virtual pages the OS will be invoked to populate the TLB with appropriate translations to physical frames.

There is, however, a significant performance penalty incurred when transitioning to the OS to resolve a TLB miss. Flushing the TLB on each context switch exacerbates this penalty since it removes any potential ability to reuse translations if the same processes runs regularly and intervening processes have not caused capacity misses (leading to potential eviction of older translations). Thus a common enhancement is to tag each TLB entry with an *address space ID* (ASID). By combining the ASID of the currently running process with the ASID recorded for each translation entry, the TLB can ensure only entries for the active address space are matched, removing the need to constantly empty the TLB.

Programmers often use *threads* to allow execution contexts to share an address space. Each thread has the same ASID and hence shares TLB entries, leading to increased performance. However, a single ASID prevents the TLB from enforcing protection; sharing becomes an “all or nothing” approach. To share even a few bytes, threads must forgo all protection from each other.

The Itanium MMU considers these problems and provides the ability to share an address space (and hence translation entries) at a much lower granularity whilst maintaining protection. The Itanium divides the 64-bit address space up into 8 *regions*, as illustrated in Figure 3.2. Each process has eight 24-bit *region registers* as part of its state, which each hold a *region ID* (RID) for each of the eight regions of the process

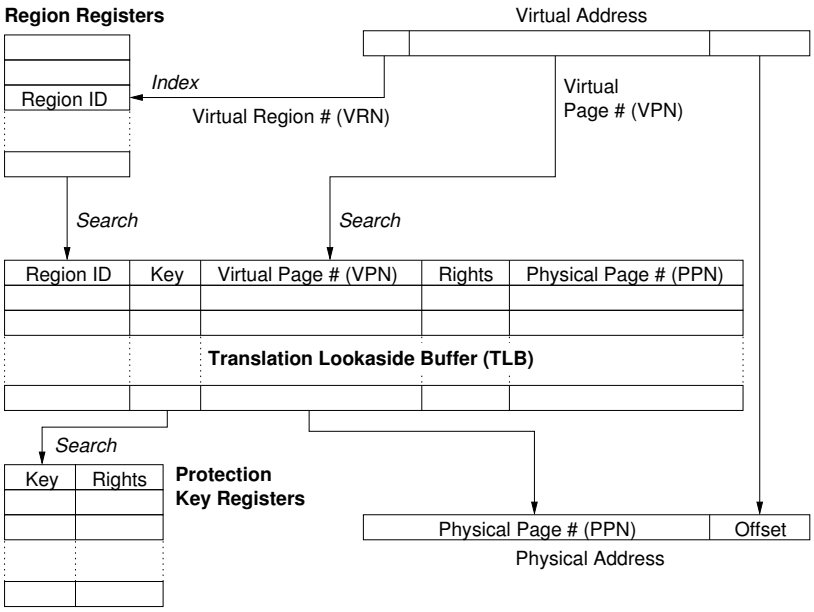


Figure 3.1: A view of the Itanium translation process [GCC+05, ME02].

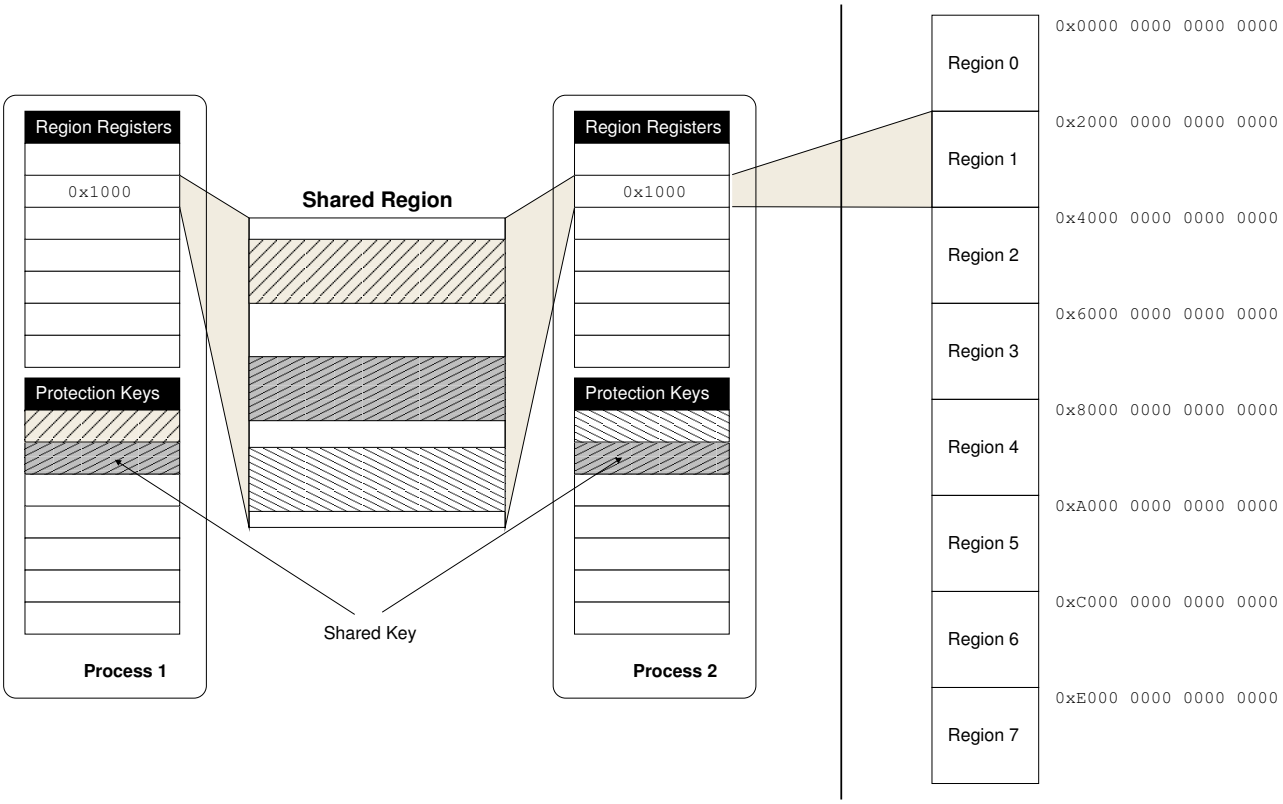


Figure 3.2: Itanium regions and protection keys. In this example the processes alias region 1. Each process has a private mapping and they share a key for another.

address space. TLB translations are tagged with the RID and thus will only match if the process also holds this RID, as illustrated in Figure 3.1.

Further to this, the top three bits (the region bits) are not considered in virtual address translation. Therefore, if two processes share a RID (i.e., hold the same value in one of their region registers) then they have an aliased view of that region. For example, if process-A holds RID `0x100` in region-register 3 and process-B holds the same RID `0x100` in region-register 5 then process-A, region 3 is aliased to process-B, region 5. This limited sharing means both processes receive the benefits of shared TLB entries without having to grant access to their entire address space.

3.1.1 Protection Keys

To allow for even finer grained sharing, each TLB entry on the Itanium is also tagged with a *protection key*. Each process has an additional number of *protection key registers* under operating-system control.

When a series of pages is to be shared (e.g., code for a shared system library), each page is tagged with a unique key and the OS grants any processes allowed to access the pages that key. When a page is referenced the TLB will check the key associated with the translation entry against the keys the process holds in its protection key registers, allowing the access if the key is present or otherwise raising a *protection* fault to the operating system.

The key can also enforce permissions; for example, one process may have a key which grants write permissions and another may have a read-only key. This allows for sharing of translation entries in a much wider range of situations with granularity right down to a single-page level, leading to large potential improvements in TLB performance [CWH03].

3.2 Hardware Page-Table Walker

Switching context to the OS when resolving a TLB miss adds significant overhead to the fault processing path. To combat this, Itanium allows the option of using built-in hardware to read the page-table and automatically load virtual-to-physical translations into the TLB. The hardware page-table walker (HPW) avoids the expensive transition to the OS, but requires translations to be in a fixed format suitable for the hardware to understand.

The Itanium HPW is referred to in Intel's documentation as the *virtually hashed page-table walker* or VHPT walker, for reasons which should become clear. Itanium gives developers the option of two mutually exclusive HPW implementations; one based on a virtual linear page-table and the other based on a hash table. Below we examine the implementation, advantages and disadvantages of each, especially with regard to large-page support.

It should be noted it is possible to operate with no hardware page-table walker; in this case each TLB miss is resolved by the OS. The performance impact of disabling the HPW is so considerable it is very unlikely any benefit could be gained from doing so [CWH03].

3.3 Virtual Linear Page-Table

The virtual linear page-table implementation is referred to in documentation as the *short format virtually hashed page-table* (SF-VHPT). It is the default HPW model used by Linux on Itanium.

3.3.1 Linear Page-Table

A linear page-table consists of a contiguous array of virtual-to-physical translations for an address space. To locate the target translation, the virtual page number is simply used as an offset from the linear page-table base (e.g., virtual-page-0 is entry-0, virtual-page-1 is entry-1, and so on).

Since every page must be accounted for, whether in use or not, a physically linear page-table is impractical with a 64-bit address space. Consider a 64-bit address space divided into (generously large) 64 KiB pages creates $\frac{2^{64}}{2^{16}} = 2^{52}$ pages to be managed; assuming each page requires an 8-byte translation entry a total of $\frac{2^{52}}{2^3} = 2^{49}$ or 512 GiB of contiguous memory is required for the page table of each process!

3.3.2 Hierarchical Page-Table

The usual solution is a multi-level or hierarchical page-table, where the bits comprising the virtual page number are used as an index into intermediate levels of the page-table. This is illustrated for a three-level page-table in Figure 3.3.

Empty regions of the virtual address space simply do not exist in the hierarchical page-table. Compared to a linear page-table, for the (realistic) case of a tightly-clustered and sparsely-filled address space, relatively little space is wasted in overheads. The major disadvantage is the multiple memory references required for lookup.

3.3.3 Virtual Linear Page-Table

With a 64-bit address space, even a 512 GiB linear table identified in Section 3.3.1 takes only 0.003% of the 16-exabytes available. Thus a *virtual linear page-table* (VLPT) can be created in a contiguous area of *virtual* address space.

Just as for a physically linear page-table, on a TLB miss the hardware uses the virtual page number to offset from the page-table base. If this entry is valid, the translation is read and inserted directly into the TLB. However, with a VLPT the address of the translation entry is itself a virtual address and thus there is the possibility that the virtual page which it resides in is not present in the TLB. In this case a *nested fault* is raised to the operating system. The software must then correct this fault by mapping the page holding the translation entry into the VLPT.

This process can be made quite straight forward if the operating system keeps a hierarchical page-table. The leaf page of a hierarchical page-table holds translation entries for a virtually contiguous region of addresses and can thus be mapped by the TLB to create the VLPT as described in Figure 3.4.

For reference, Appendix A.1 shows a flowchart of the Linux fault handling code when operating with the SF-VHPT.

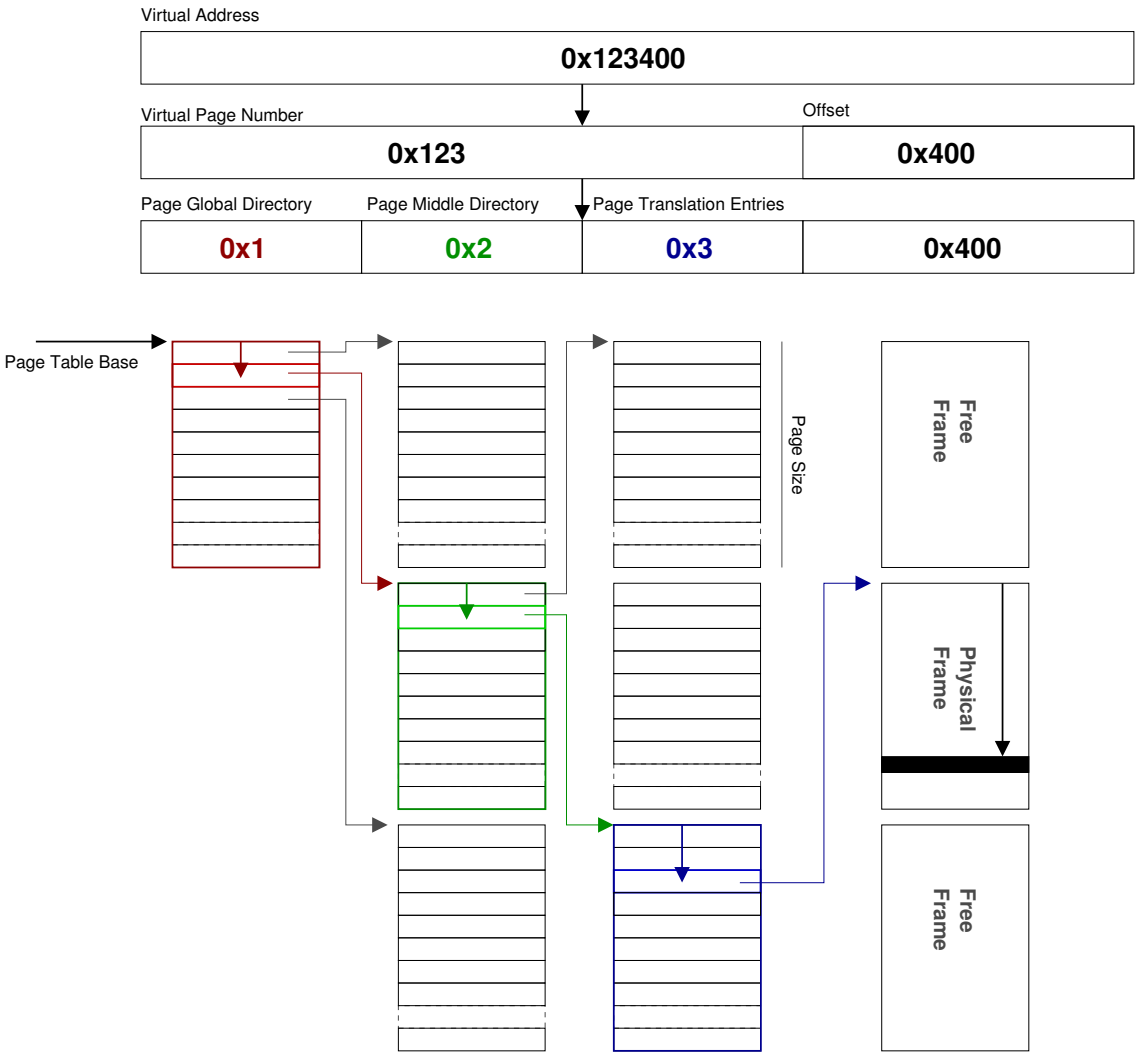


Figure 3.3: A three level hierarchical page table.

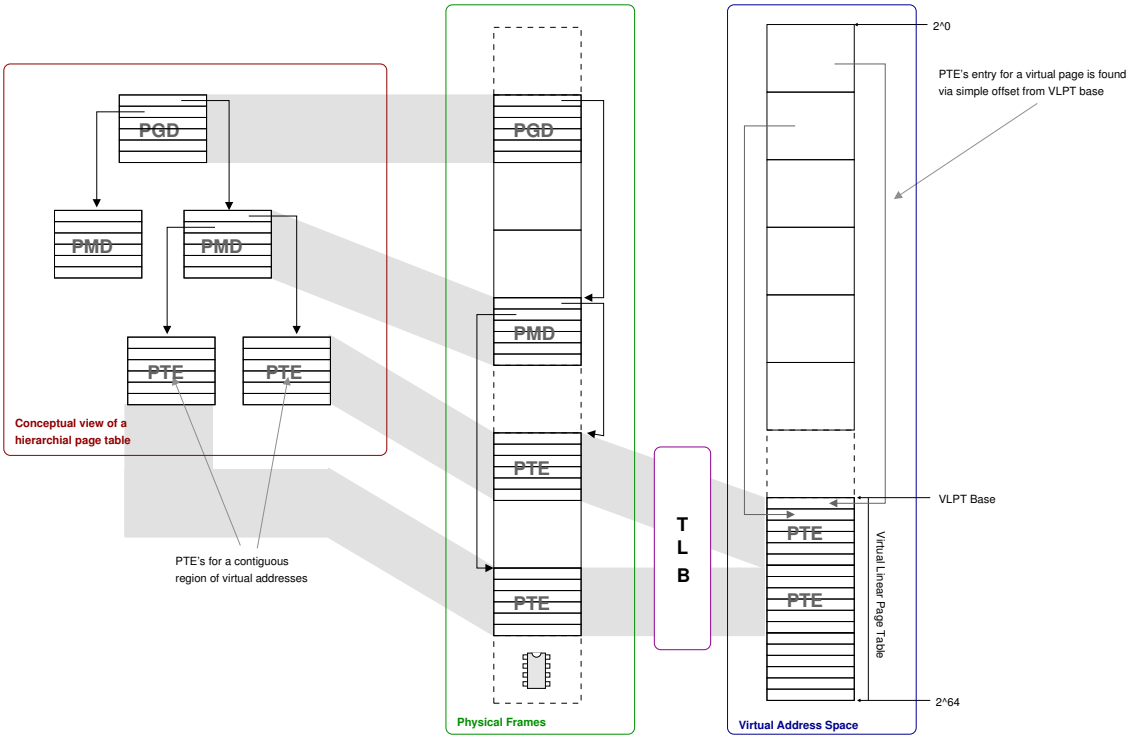


Figure 3.4: Virtual linear page table operation. The virtual page number gives the offset into the virtual linear page-table. This access may raise a *nested fault*; if so the operating system can map the appropriate leaf frame of the hierarchical page-table.

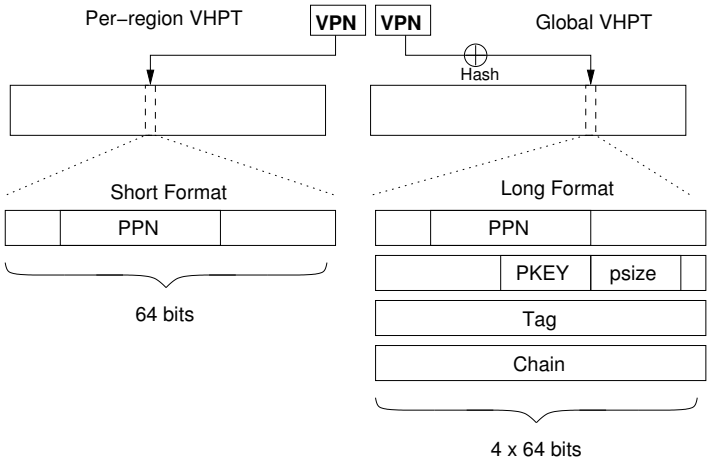


Figure 3.5: Itanium translation formats [GCC+05]

3.3.4 Discussion

The major advantage of a VLPT occurs when an application makes repeated or contiguous accesses to memory. Consider that for a walk of virtually contiguous memory, the first fault will map a page full of translation entries into the virtual linear page-table. A subsequent access to the next virtual page will require the next translation entry to be loaded into the TLB, which is now available in the VLPT and thus loaded very quickly and without invoking the operating system. Overall, this will be an advantage if the cost of the initial nested fault is amortised over subsequent HPW hits.

The major drawback is that the VLPT now requires TLB entries which causes an increase on TLB pressure. Since each address space requires its own page table the overheads become greater as the system becomes more active. However, any increase in TLB capacity misses should be more than regained in lower refill costs from the efficient hardware walker. Note that a pathological case could skip over `page_size` ÷ `translation_size` entries, causing repeated nested faults, but this is a very unlikely access pattern.

The hardware walker expects translation entries in a specific format as illustrated on the left of Figure 3.5. The VLPT requires translations in the so-called 8-byte *short format*. If the operating system is to use its page-table as backing for the VLPT (as in Figure 3.4) it must use this translation format. The architecture describes a limited number of bits in this format as ignored and thus available for use by software, but significant modification is not possible.

A linear page-table is premised on the idea of a fixed page size. Multiple page-size support is problematic since it means the translation for a given virtual page is no longer at a constant offset. To combat this, each of the 8-regions of the address space (Figure 3.2) has a separate VLPT which only maps addresses for that region. A default page-size can be given for each region (indeed, with Linux HugeTLB, discussed below, one region is dedicated to larger pages). However, page sizes can not be mixed within a region. Exact implementation details are discussed further in Chapter 5.

3.4 Hash Page-Table

Using TLB entries in an effort to reduce TLB refill costs, as done with the SF-VHPT, may or may not be an effective tradeoff. Itanium also implements a *hashed page-table* with the potential to lower TLB overheads. In this scheme, the processor *hashes* a virtual address to find an offset into a contiguous table.

The previously described physically linear page-table can be considered a hash page-table with a *perfect* hash function which will never produce a collision. However, as explained, this requires an impractical tradeoff of huge areas of contiguous physical memory. However, constraining the memory requirements of the page table raises the possibility of collisions when two virtual addresses hash to the same offset. Colliding translations require a *chain* pointer to build a linked-list of alternative possible entries. To distinguish which entry in the linked-list is the correct one requires a *tag* derived from the incoming virtual address.

The extra information required for each translation entry gives rise to the moniker *long-format* VHPT (LF-VHPT). Translation entries grow to 32-bytes as illustrated on the right hand side of Figure 3.5.

The hash function is illustrated below [Cha], where `PAGE_SHIFT` is the number of offset bits for the default page size of the region.

```
offset = ((virtual_address >> PAGE_SHIFT) ^ RID) & (VHPT_ENTRIES-1)
```

The hash table is *global*, hence the addition of the RID to avoid the same virtual address in two processes colliding. Each processor keeps its own hash table (SMP issues are discussed in Section 4.7).

For reference, Appendix A.2 shows a flowchart of the Linux fault handling code when operating with the LF-VHPT HPW.

3.4.1 Discussion

The main advantage of this approach is the global hash table can be pinned with a single TLB entry. Since all processes share the table it should scale better than the SF-VHPT, where each process requires increasing numbers of TLB entries for VLPT pages. However, the larger entries are less cache friendly; consider we can fit four 8-byte short-format entries for every 32-byte long-format entry. The very large caches on the Itanium processor may help mitigate this impact, however.

One advantage of the SF-VHPT is that the operating system can keep translations in a hierarchical page-table and, as long as the hardware translation format is maintained, can map leaf pages directly to the VLPT. With the LF-VHPT the OS must either use the hash table as the primary source of translation entries or otherwise keep the hash table as a cache of its own translation information. Keeping the LF-VHPT hash table as a cache is somewhat suboptimal because of increased overheads on time critical fault paths, however advantages are gained from the table requiring only a single TLB entry. These tradeoffs are examined further in Chapter 4.

The global nature of the table means RID allocation is important in avoiding hash collisions. Linux assigns each processes address space a unique, monotonically increasing *context* identifier. This is implemented on Itanium with a direct mapping between the context number and the RIDs which make up a processes address space; for example the address space with context `0x20` has a Region 1 RID of `0x21`, Region 2 RID of `0x22` and so forth. The combination of highly aliased UNIX address spaces created by `fork()` and high locality of reference produced by the hash function (Section 3.4) can produce regular patterns of collisions. As a theoretical example, if VPN `0x1000` was accessed by RID 0x100 and then VPN `0x1001` was accessed by a freshly forked process with RID `0x101`, both would refer to a hash-table offset of `0x1100`.

The solution is to re-distribute the lower bits of the 24-bit RID to reduce the contiguity of RID allocations without breaking the Linux rule of a monotonically increasing context identifier. Many other schemes could be considered if this type of contention becomes apparent in a running system.

Large-page support is still an issue with the hash page-table. The extra room in the long format translation entry provides for an explicit *page size* field, unlike the short format entry which specifies it must take its page size from the default size set for the region. This means that in LF-VHPT mode the HPW will load a translation into the TLB with an arbitrary page size.

However, there is still the issue of not knowing the page size when hashing the virtual address. The hash function described in Section 3.4 shifts out the offset bits from the virtual page number, taking the page size from the default for the region. If using the hash table as a primary source of translation data, this would mean each sub-page of a large page would require an entry in the table. If using the table as a cache it reduces the hit rate, since accessing the same large page but on a different sub-page would require instantiating another translation entry.

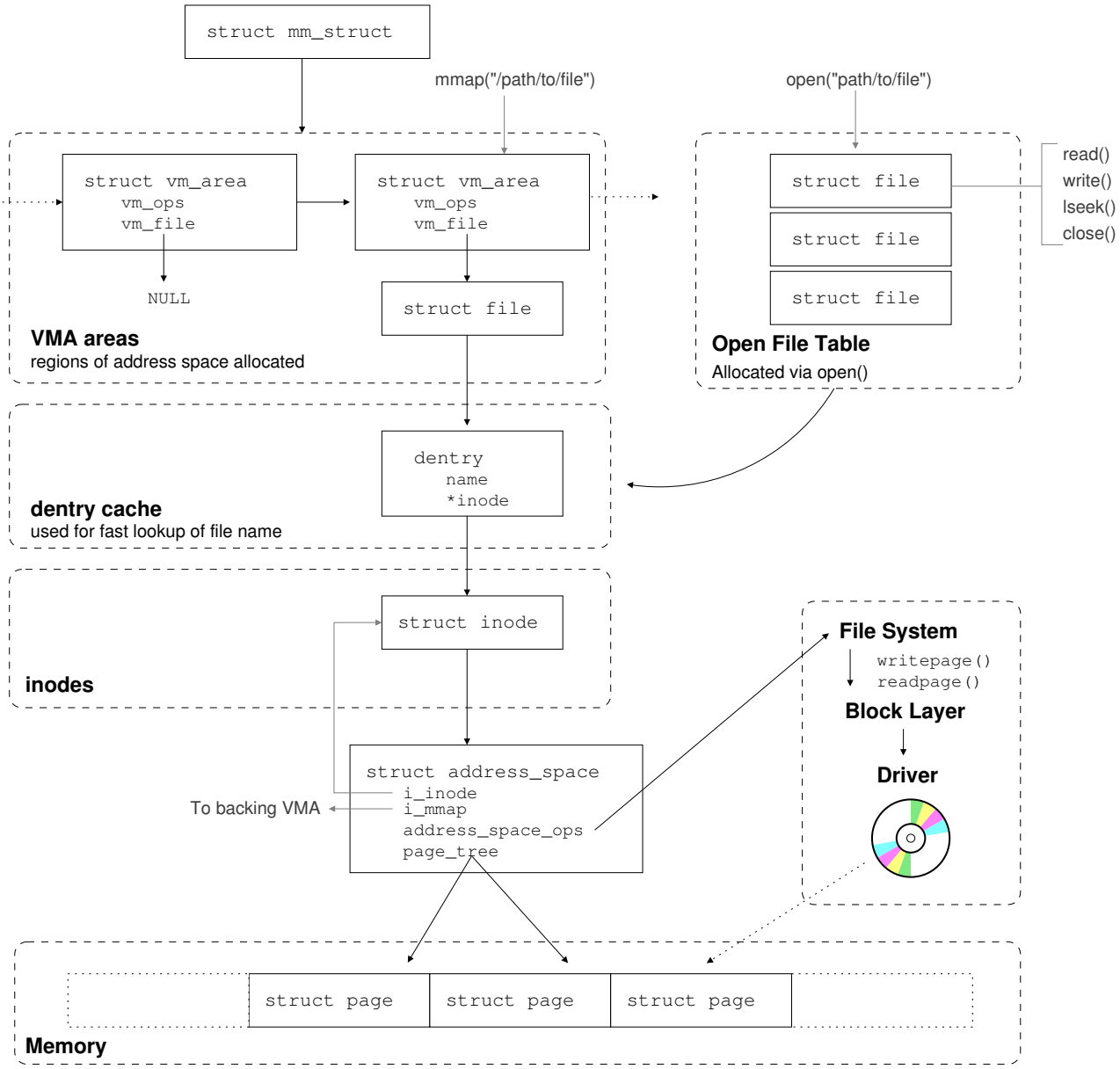


Figure 3.6: Overview of Linux VFS layers

3.5 HugeTLB

Linux currently provides support for large pages via the *HugeTLB* mechanism. As mentioned in Section 2.4.1 this is a global scheme which allocates static large pages from reserved memory. Below we introduce the design and discuss the Itanium implementation.

3.5.1 Linux VFS

Linux is an extremely portable and extensible operating system and relies on good use of abstraction to allow the many supported architectures, drivers and kernel subsystems to work seamlessly with each other. An understanding of the HugeTLB infrastructure starts with an overview of one of the most important abstractions in the kernel, the *virtual file-system* (VFS).

As with traditional UNIX, the VFS layer is the abstraction layer between userspace file handles and the underlying file-systems responsible for storing data on disk. The Linux implementation is illustrated in Figure 3.6. The basic file data structure is the `struct file`; using the standard `open()` system call adds a `struct file` into the processes open file table. Each `struct file` has an associated set of function pointers to *operations* such as `read()`, `write()` and `close()`, which are used by the system calls of the same name. Although they may be overridden for special file types (e.g., to provide direct access to hardware for device files, etc.) by default they are connected to generic functions which interact with the disk cache and underlying *inode*.

The `inode` is a low level representation of a file on disk. It contains information such as the file size, owner and reference counts for hard links. Each `inode` is associated with an `address_space` which acts as an "MMU for inodes" [Gor04]. It describes the link between a file and memory, holding information such as a quick-list of all memory pages currently allocated to this `inode`, any memory pages that are currently *dirty* (out of sync with the backing store) and a number of reverse pointers to aid lookups. The most important field is the `address_space_ops` which is a list of function pointers provided by the underlying file-system. The `writepage()` and `readpage()` function pointers in this structure are connected to those provided by the file-system the *inode* resides within. These functions ultimately provide the method to move data to and from disk.

Linux includes an intermediary *dentry cache* to avoid having to traverse the directory hierarchy (and hence access disk) each time a file is opened. Each `struct file` is associated with a single `struct dentry` in the cache. Each entry in the `dentry` cache points to the underlying `inode`. Whilst a file will only ever point to one `dentry`, a `dentry` may have multiple parents due to hard links.

`mmap`

Memory mapped (`mmap`ed) files are implemented within the VFS hierarchy. A processes virtual addresses space consists of a linked list of *virtual memory areas* (VMA) illustrated on the top-left of Figure 3.6 (see also Figure 2.4). Each region of the address space is assigned to, and described by, a separate `struct vma_area`. When backed by a file the VMA will have its `vm_file` pointer filled in with a pointer to the `struct file` which backs the area. A VMA without this pointer is backed by RAM and is termed *anonymous*.

Unlike the linear behaviour ensured by accessing a file via a file descriptor, a memory mapped file needs to handle a request for any page at any time. To facilitate this, each VMA has a `struct vm_ops` which contains a `fault()` function pointer (in many older kernel versions this was called `nopage()`). The process of mapping a page from disk is illustrated in Figure 3.7. When there is a TLB miss, the virtual memory subsystem finds the VMA where the faulting address lies. If that VMA is backed by a file then its registered `fault()` function is called, which is required to return a pointer to the page the data can now be found in. This can then be mapped into the TLB and the access retried.

3.5.2 Shared Memory

A `mmap` may be either *private* or *shared*. A private `mmap` implies the memory should be *copy-on-write* across `fork()` and any changes to the memory mapped area should not be reflected in the backing file. Conversely, shared mappings ensure the opposite; `fork()` must create aliased copies with changes propagated to the backing file.

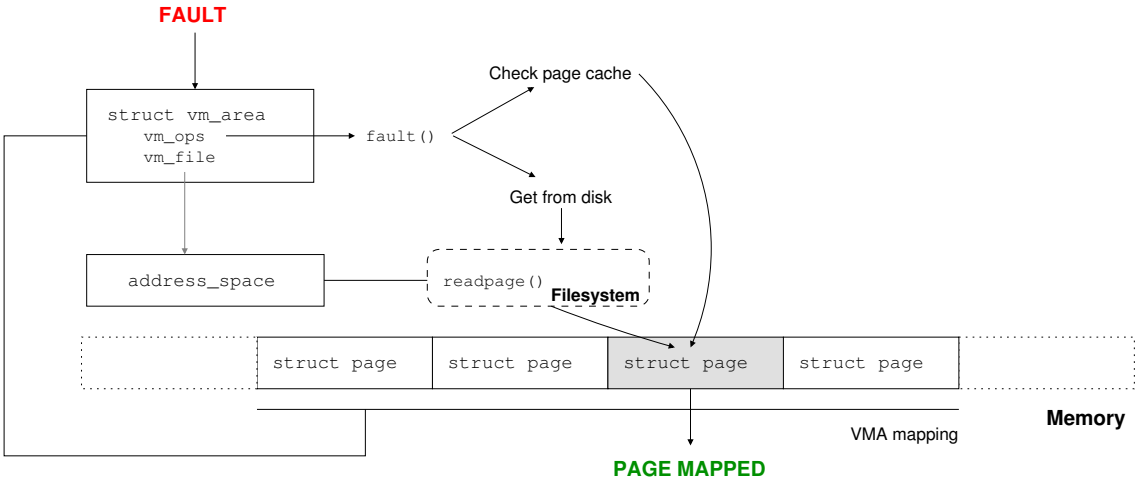


Figure 3.7: `vm_ops fault()` overview

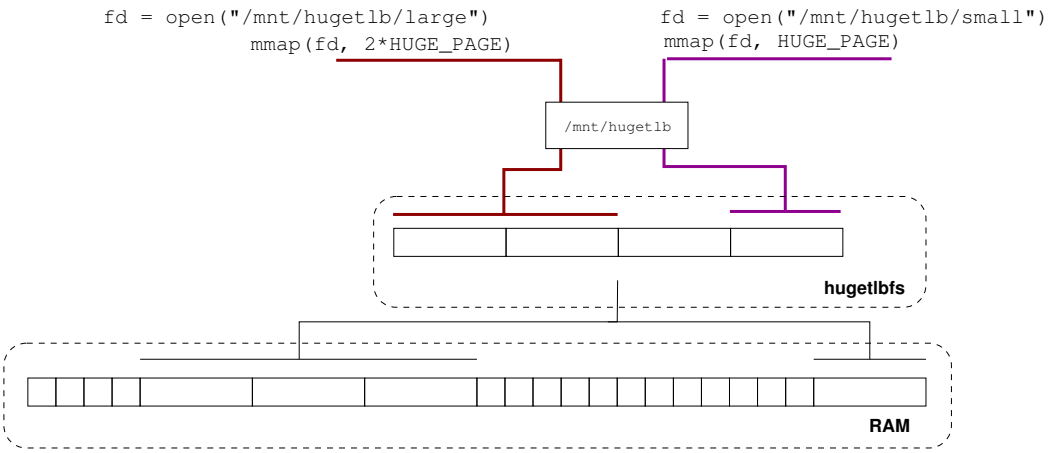


Figure 3.8: Illustration of HugeTLB file-system

Private anonymous memory is therefore handled in the same manner as any other private memory allocated to the process (such as that allocated by `brk()`). Supporting separate code paths for anonymous and file-backed shared memory has the potential to greatly complicate the `mmap` model presented in Figure 3.6. To avoid this, Linux maintains the file system abstraction for shared anonymous memory by creating a file system over RAM termed *shmfs*. The "blocks" that `shmfs` allocates are pages of physical memory, just as a traditional file-system manages blocks on disk. This abstraction greatly simplifies the `mmap` operation by hiding the details such as page allocation, interaction with the disk cache, swapping in/out data and general accounting in an encapsulated file-system (for a full discussion of the file-system operation, see Gorman [Gor04]).

It also significantly simplifies System V shared memory. A `shmget()` call is internally translated into a shared anonymous `mmap` on the `shmfs` and the System V shared memory implementation need only keep an index between the shared memory key and the memory mapped area.

3.5.3 HugeTLB

The file-system abstraction is also maintained with large page support. HugeTLB is implemented as another file-system over physical memory at the same level as `shmfs`. However, rather than allocating base pages of memory from the general system pool, it allocates from large pages preallocated by the administrator. These pages are then available via two mechanisms; either the `mmap` of a file mounted on a special virtual file-system, as illustrated in Figure 3.8, or via a flag when creating System V shared memory.

Using a file-system makes the allocation of large pages architecture independent. However, each architecture implementation will need to ensure it loads larger TLB entries to cover the larger pages. On Itanium this is achieved by reserving one region (Section 3.1) exclusively for large pages. By default this region is set to use 256 MiB pages, but can be changed by the administrator. TLB miss handlers are modified slightly to check if the incoming faulting address lies in the HugeTLB region and, if so, ensure the TLB is set with the correct page-size for the translation. Since the SF-VHPT HPW is region based its effectiveness is not reduced for the HugeTLB region.

Benefits

The advantages provided by HugeTLB memory will vary depending on the TLB overhead a program experiences. Database workloads are known to often exhibit TLB intensive behaviour and therefore one of the most important users of the HugeTLB architecture is the Oracle database. A large part of Oracle performance is dependent on the *shared global area* (SGA), a region of System V shared memory statically allocated upon initialisation. This area is then used to hold buffers and facilitate communication between the numerous processes involved in querying the database. The SGA size is set by the administrator; ideally the entire database will be able to be served from the SGA, with the general rule being a larger SGA results in higher performance.

The large and static SGA is a perfect use-case for the Linux HugeTLB model. At boot time (via a kernel command line) or very early after boot the administrator can request a reservation of large pages sufficient to cover the SGA. If the database is running on a dedicated machine (which is often the case in a production environment) this reservation can be a very large percent of the total available memory. Oracle will then claim this memory when it starts.

The `orabm` benchmark is based upon the TPC-C benchmark [Tra07] which is designed to simulate a high traffic OLTP environment. The benchmark simulates an order-entry environment of terminal operators accessing warehouse information and reports the number of *transactions per second* (TPS), higher being better. The working set is relatively small (several hundred megabytes) and restricted to read-only operations, meaning the benchmark is memory and CPU bound. It is useful to remove I/O since the latency involved in getting to disk may pollute the results and obscure any TLB effects. The benchmark is run against the Oracle 10g database server on a 900Mhz Itanium2 with 2 GiB of RAM.

Figure 3.9 shows the benefits of using HugeTLB. Each point is an average of four benchmark runs, and overall an approximate 5% increase in TPS is observed when using HugeTLB. In this case the SGA is mapped into a 1 GiB region using four 256 MiB pages, which we would assume would essentially eliminate TLB misses. These results confirm anecdotal evidence of more varied workloads achieving performance improvements of 5-8%.

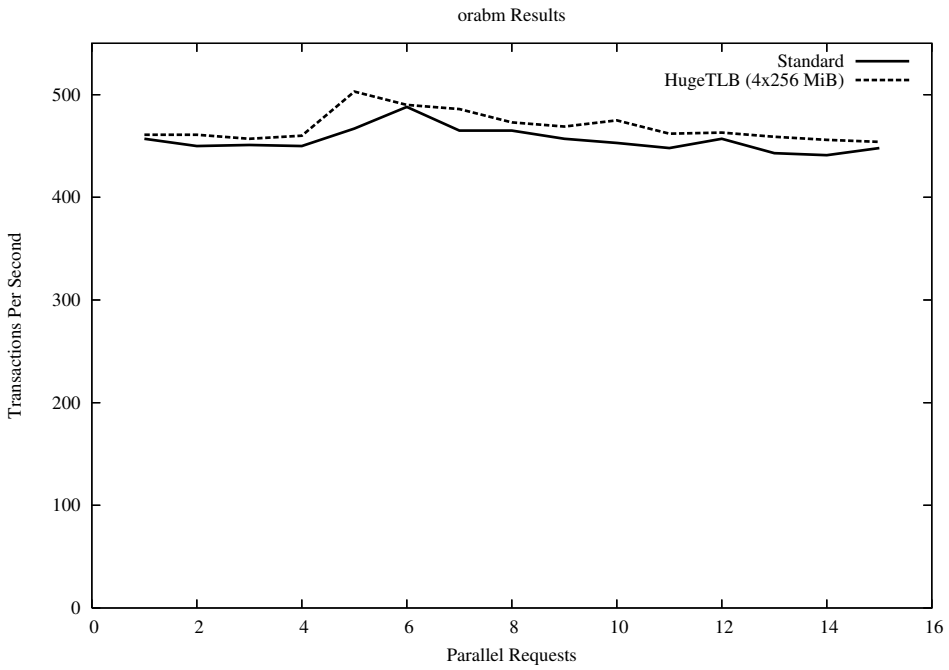


Figure 3.9: Results of the `orabm` Oracle benchmark.

3.5.4 Discussion

Whilst highly effective in specific situations, using HugeTLB requires significant intervention from knowledgeable administrators and programmers. Intermediate steps of abstracting the interface behind a shared library wrapper are still under development [GL]. Whilst having the advantage of requiring less intervention, such approaches are still tied to the HugeTLB implementation and thus will not be able to grant access to the full range of page sizes available on an architecture such as Itanium.

Although this work does not consider shared anonymous mappings, we believe it can interface with proposed developments to the Linux memory model (see Section 5.6.2 for a full discussion).

3.6 Conclusion

We have described the operation of the two forms of HPW on Itanium and discussed some of the challenges of integrating large-page support within them. A detailed examination of the existing Linux HugeTLB system is also presented. The remainder of this thesis is a discussion of various strategies for enabling transparent multiple size large-page support for Linux within these constraints.

Chapter 4

LF-VHPT Evaluation

This chapter presents a detailed analysis of the long-format VHPT (LF-VHPT) hardware page-table walker (HPW). As described in Chapter 3, the LF-VHPT promises advantages for large-page support via its ability to load translation entries with differing page sizes. Below we examine the operation of the LF-VHPT in a static page-size system to gauge its relative performance against the short-format VHPT (SF-VHPT).

Section 4.3 shows results of micro-benchmarks crafted to exercise extreme cases of the two forms of HPW. Section 4.4 presents results of the CPU and memory intensive SPEC benchmark suite and Section 4.5 shows the results of the LMBench suite, a series of micro-benchmarks designed to exercise specific subsystems of the operating system. Section 4.6 then presents results of benchmarks testing more varied single and multi-process workloads. Section 4.8 then concludes.

4.1 Prior Work

The LF-VHPT implementation was originally implemented by Matthew Chapman [CWH03]. Maintenance and enhancement of the work was undertaken by both Darren Williams and myself at various times.

4.2 Test Environment

All benchmarks are run on a 1.5 GHz Itanium2 (“Madison”) in a HP rx2600 server. The processor has 16 KiB, 4-way associative split (separate instruction and data) L1 caches, 256 KiB 8-way associative unified L2 cache and 6 MiB of 12-way associative unified L3 cache.

The processor has separate 32-entry L1 instruction and data TLBs, however these only support a fixed 4 KiB page size and hence do not respond to larger page sizes. The L2 instruction and data TLBs have 128 entries and support a the full complement of page sizes as shown in Table 4.1.

4 KiB	8 KiB	16 KiB	64 KiB	256 KiB	1 MiB	4 MiB	16 MiB	64 MiB	256 MiB	1 GiB	4 GiB

Table 4.1: Page sizes supported by the Itanium 2 Processor [Int00].

```
1  #define LOOPS 3
2  #define AREAS 512
3  #define PAGE_SIZE 16384
4
5  #define MMAP(i) ((0x2000000UL * i) + (i * PAGE_SIZE))
6
7  char *memareas[AREAS];
8
9  int main(void)
10 {
11   int i,j;
12   for (i=0; i < AREAS; i++)
13     memareas[i] = mmap((void*)MMAP(i), PAGE_SIZE,
14                        PROT_READ|PROT_WRITE,
15                        MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
16
17   for (i=0; i < LOOPS; i++)
18     for(j = 0; j < AREAS; j++)
19       memareas[j][0] = 'a';
20
21   return 0;
22 }
```

Figure 4.1: Benchmark to stress HPW

The system has 2 GiB of RAM and uses the HP zx1 chip-set. The base kernel was a standard Linux 2.6.18 kernel, with benchmarks using the Itanium Linux default base page-size of 16 KiB unless otherwise specified.

4.3 Extremes

Below we use micro-benchmarks designed to stress particular extreme situations to gain insight into the operation of the HPW.

4.3.1 Sparse access

To initially validate the operation of the LF-VHPT HPW we wish to analyse a situation with significant TLB capacity misses. This is achieved with a micro-benchmark touching a large number of individual pages on 32 MiB boundaries as presented in Figure 4.1.

As identified in Section 3.3.4, this is a worst-case scenario for the SF-VHPT HPW. With 16 KiB pages and 8-byte translation entries each page of the VLPT covers $\frac{16384}{8} \times 16384 = 32\mathrm{MiB}$ of virtual address space and therefore each access the benchmark program makes will cause a nested fault. This results in the SF-VHPT HPW being not being able to load any translations from the virtual-linear page-table at all.

When using the LF-VHPT HPW the translation will be instantiated in the hash table on the first loop and subsequently retrieved by the HPW for two of the three passes of the test loop. Profiling reveals this to be the case, with ≈1024 HPW inserts recorded. The only caveat is the need to perturb the pages across 2 GiB boundaries (line 5). The test system utilised a 4 MiB hash table and therefore, with 16 KiB pages, the hash

Metric	Triggered By	SF-VHPT	LF-VHPT
L2DTLB_MISSES	L2 TLB miss	65619	65619
DTLB_INSERTS_HPW	HPW insert	65570	19

Table 4.2: TLB misses and HPW inserts for contiguous walk of 1 GiB region

function presented in Section 3.4 will alias 2 GiB regions. Not perturbing the pages would result in hash table collisions.

Further loops are satisfied by the LF-VHPT hash table. To make the test measurable the number of loops was increased to 100,000 and the results showed a large runtime difference; SF-VHPT runs in 4.42 seconds whilst LF-VHPT almost halves this at 2.80 seconds.

Although this benchmark is clearly contrived it provides validation and an upper bound estimate of the benefits of LF-VHPT for a sparse and repeated access pattern.

4.3.2 Linear Access

In contrast to this, Figure 4.2 shows the cycles-per-fault with a single process walking linearly over a `mlock`ed 1 GiB region. This represents the opposite scenario of a best-case for SF-VHPT and a worst-case for LF-VHPT. This benchmark is useful as it allows for an examination of the cost of faults.

The no-VHPT case is provided for a baseline comparison in Figure 4.2a. It represents the cost of transitioning to the OS and walking the page table.

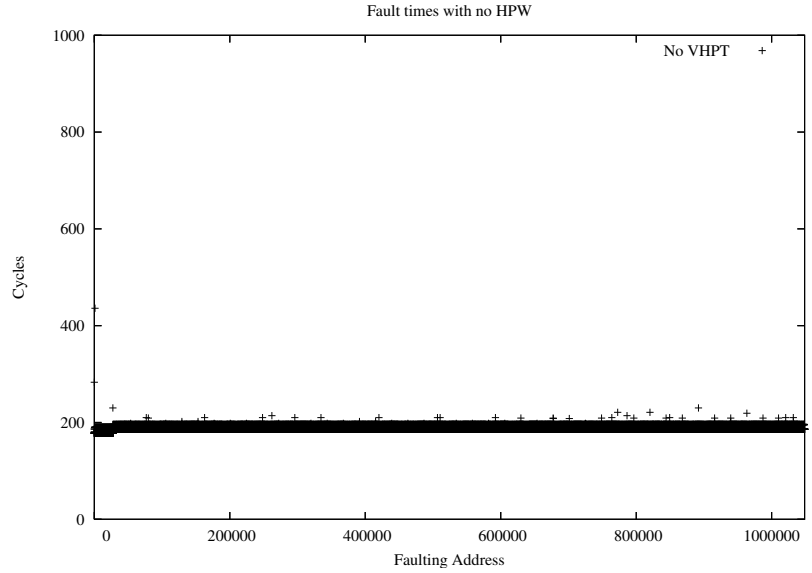
The SF-VHPT (Figure 4.2b) shows a spike for the first page of every 32 MiB region as the nested fault is taken. As expected this peaks to around the constant level for the no-VHPT case (i.e., Figure 4.2a) since at this point both operations are essentially the same — traverse the page tables and insert a mapping. We expect $\frac{1\ GiB}{16\ KiB} = 65,546$ TLB misses overall, with most being covered by the HPW. These estimates correlate with the counter results presented in Table 4.2.

The LF-VHPT (Figure 4.2c) shows a large variation in the cost of faults and results in an average run-time 4.4% longer than the SF-VHPT test. On each page-fault control is returned to the operating system which walks the page table and inserts the translation into both the TLB and hash table. Therefore we expect the baseline time to be similar to the no-VHPT case. Closer inspection reveals cache access as the cause of the very haphazard looking result. Figure 4.3, a zoomed-in extract of Figure 4.2c, shows a regular spike in access time every fourth fault, corresponding to the LF-VHPT handler accessing another cache line (e.g., each LF-VHPT entry is 32 bytes, so four entries fit in one of the Itanium's 128-byte cache lines). Prior work has highlighted the importance of maximising cache line affinity of adjacent translations [Elp99]. Although the linear access pattern does exploit this locality property the penalties for accessing fresh lines are unavoidable.

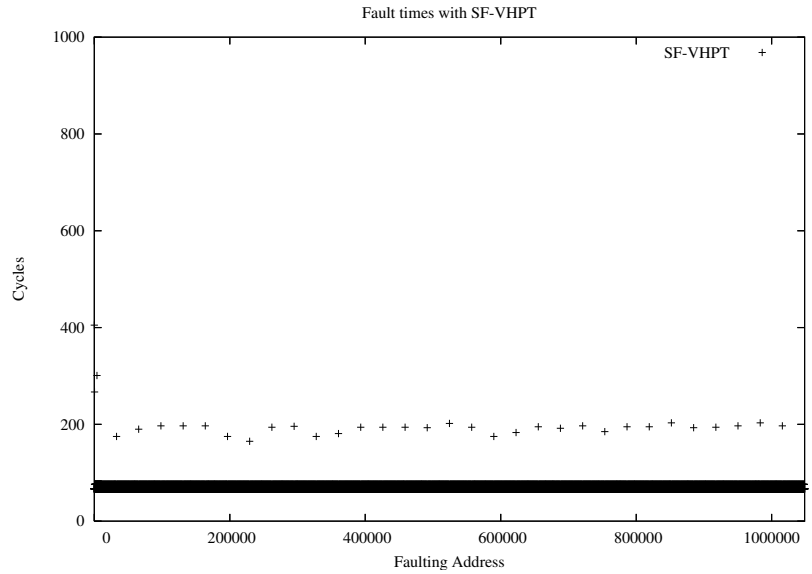
As shown in Table 4.2, by only accessing the pages once the HPW is never given a chance to reclaim the extra cycles spent instantiating the hash table.

Micro-architectural analysis

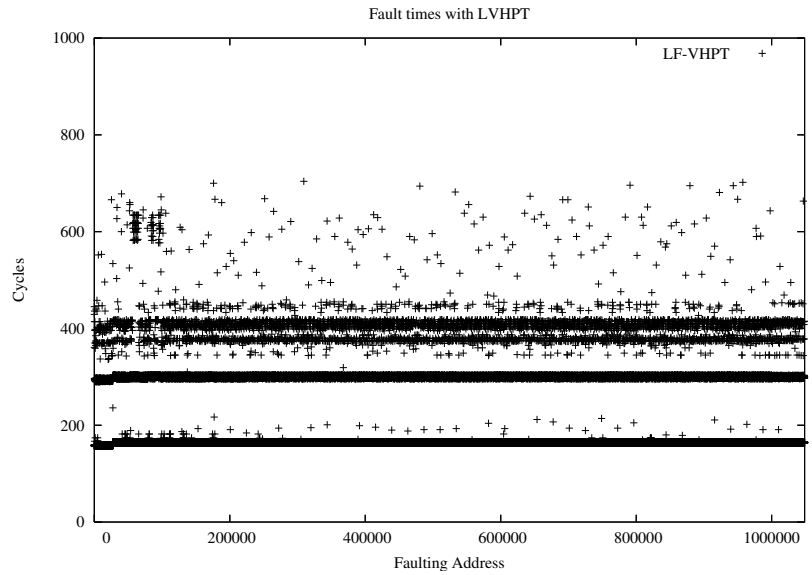
The Itanium processor has a performance management unit (PMU) which is able to attribute cycles to various events within the processor. These events are organised in a hierarchy where upper layers are



(a) No VHPT



(b) Short Format



(c) Long Format

Figure 4.2: Cycles per miss when faulting linearly over a 1 GiB region (Section 4.3.2).

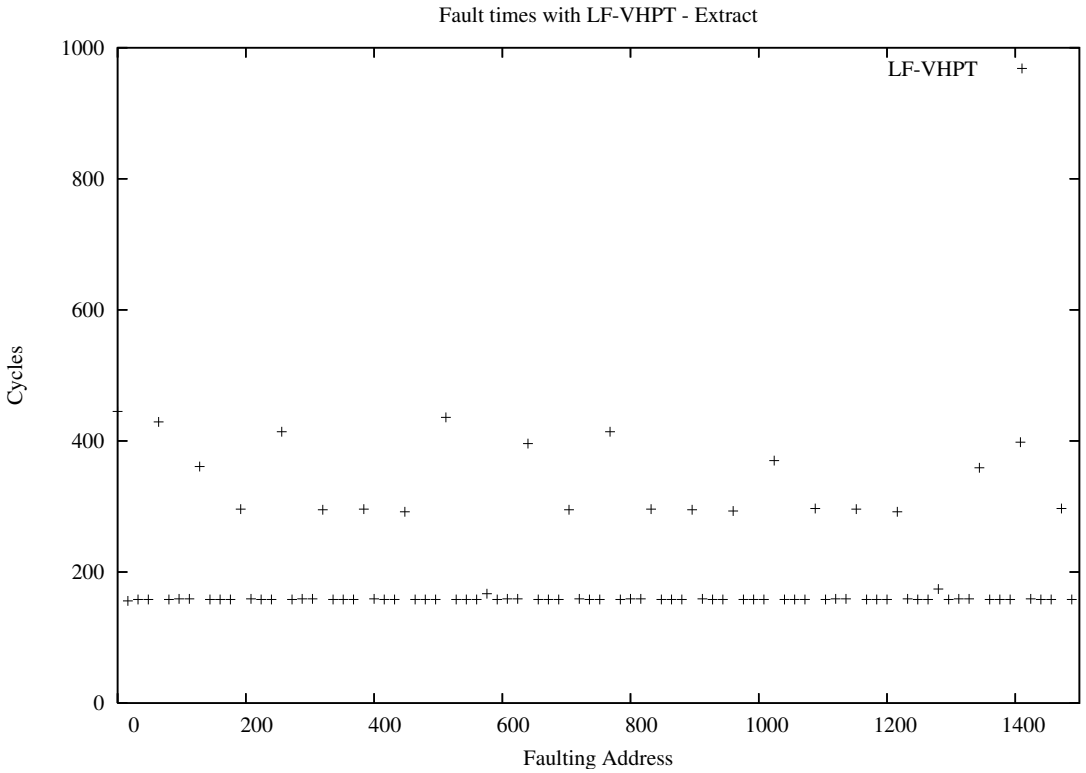


Figure 4.3: Extract of Figure 4.2c

Event	Triggered by	Difference from SF-VHPT
BE_L1D_FPU_BUBBLE.L1D_HPW	waiting for HPW	212.10%
BE_L1D_FPU_BUBBLE.L1D_DCURECIR	memory contention	79.45%
BE_L1D_FPU_BUBBLE.L1D_L2BPRESS	lack of L1D ↔ L2D bandwidth	-54.44%
BE_L1D_FPU_BUBBLE.L1D_TLB	L2 → L1 TLB transfer	5.60%

Table 4.3: LF-VHPT cycles accounted to Itanium events for a contiguous walk of 1 GiB of memory, relative to SF-VHPT cycles.

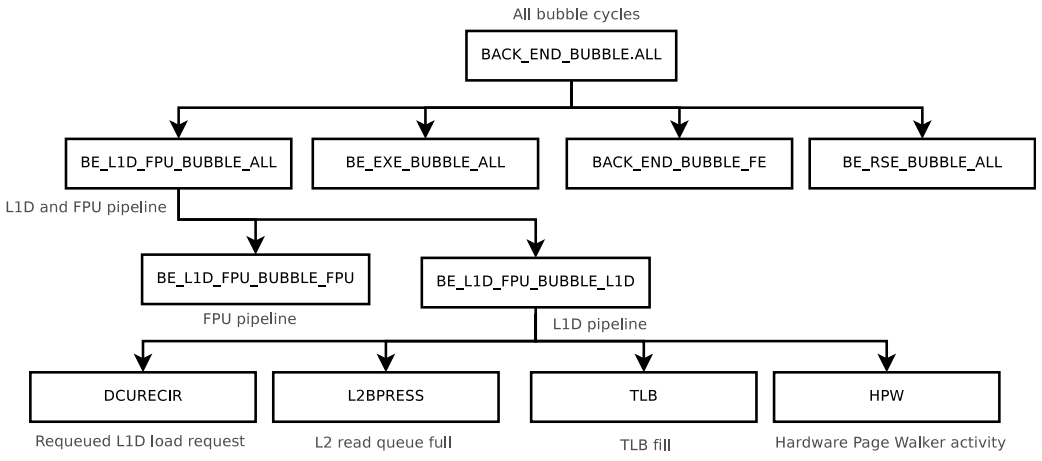


Figure 4.4: Hierarchy of Itanium back-end bubble performance management counters

cumulative of their child events; this allows drilling-down to more specific events to better analyse results. Figure 4.4 illustrates the breakdown of "bubble" events; i.e., cycles where the processor is not retiring instructions. This figure focuses on the breakdown of the processor "L1D pipeline" which parents a range of events relating to TLB, HPW and cache performance. Table 4.3 shows the cycle times attributed to L1D stall events for the SF-VHPT and LF-VHPT.

The major increase in running time for the LF-VHPT can be attributed to the `L1D_HPW` event, which measures the amount of time spent waiting for the hardware page-table walker. On each fault the HPW must check if the translation is available for insertion; the large increase in cycles spent waiting reflects the greater time the 32-byte long-format entries take to access. The variability of the hash table data in the memory hierarchy contributes to the widely differing timing results.

The `L1D_DCURECIR` (data cache unit recirculation) event can be caused as a secondary effect of other pipeline events or can be symptomatic of memory contention; the L1 cache does not queue load events so any that cannot be satisfied immediately must be *recirculated*. This event is by its nature non-specific, but we note that architecture manuals suggest time attributed to the HPW is accumulated in `L1D_DCURECIR` and `L1D_HPW` events. The LF-VHPT will have a less regular memory access pattern which leads to greater variability in cache wait times.

Conversely the `L1D_L2BPRESS` event is triggered when the L2 caches 32-entry out-of-order "*OzQ*" queue becomes full and the L1 cache is asked to stop sending data. The higher count for the SF-VHPT is probably due to the regularity of access by the HPW when using SF-VHPT; since the benchmark proceeds linearly the HPW can very quickly access adjacent translation entries which puts high demands on cache bandwidth.

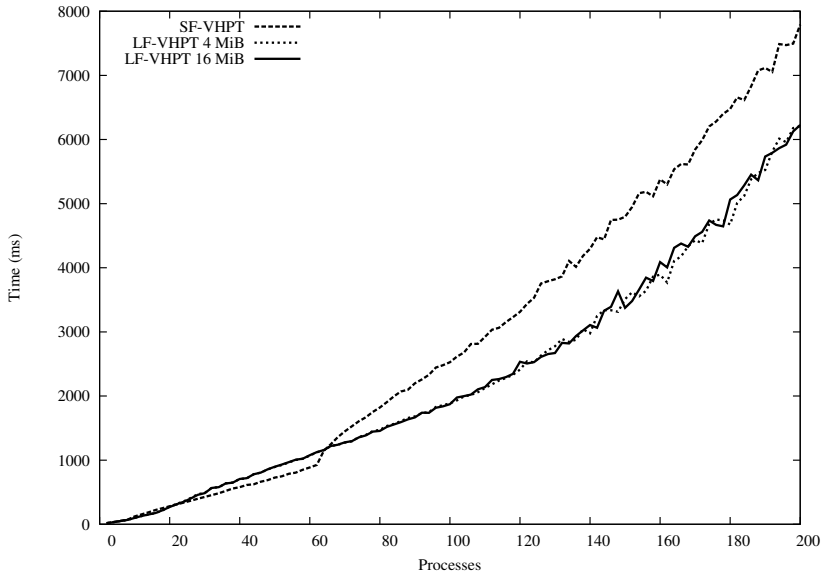
`L1D_TLB` reflects time spent waiting for transfers from the L2 TLB to the L1 TLB. Although Table 4.2 shows us that the number of faults is the same, the LF-VHPT has fewer chances to reuse L1 TLB entries and thus spends slightly more time waiting to fill the top-level TLB.

4.3.3 High Contention

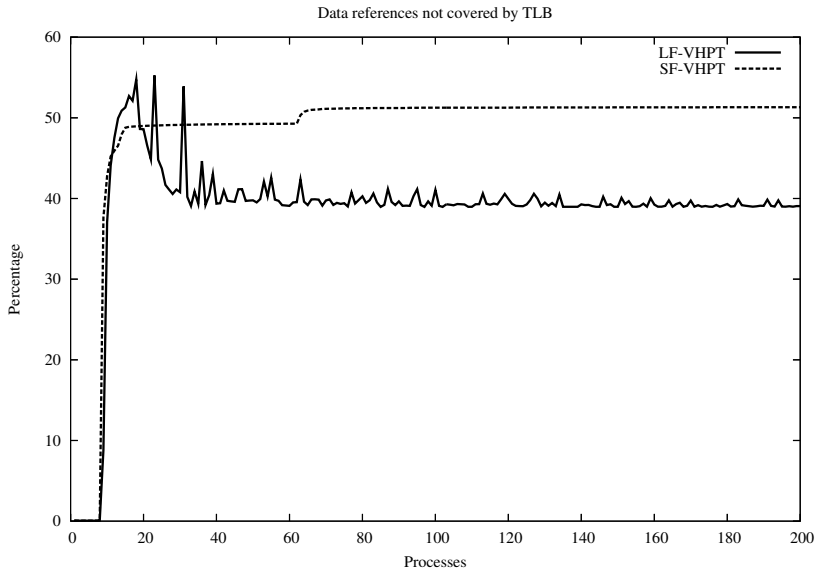
Previous micro-benchmarks have considered a single process case. To contrast these benchmarks, Figure 4.5 shows results of an increasing number of processes walking a small `mmap`ed region. Each process repeatedly accesses a single cache line on 10 contiguous virtual pages in an effort to make as many page faults as possible. This is expected to be a good case for the LF-VHPT due to the HPW covering the repeated TLB misses and the greater number of entries available to the benchmark processes.

Overall time results in Figure 4.5a show that LF-VHPT does scale better as more processes compete for TLB entries. This effect is seen more clearly in Figure 4.5b where the extra TLB entries available when the LF-VHPT HPW is enabled is reflected by a much lower percentage of data references causing a TLB miss.

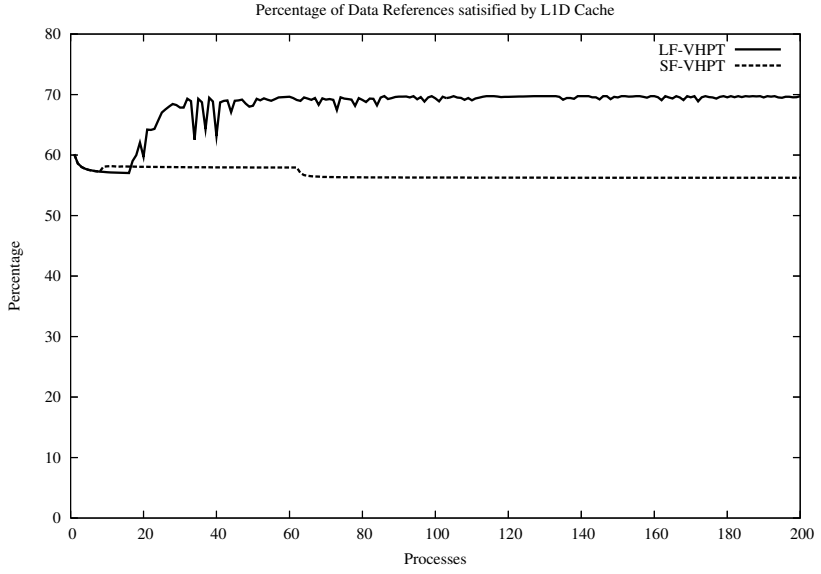
The SF-VHPT shows a marked degradation at 64 processes and the inflection point is also prominent in the cache performance figures presented in Figure 4.5c. A "back of the envelope" calculation suggests this is a reasonable point where page table data overwhelms L1 capacity; consider the 16 KiB L1 cache is divided into 64-byte lines, meaning $\frac{16384}{64} = 256$ possible lines. If each process has one cache line for data and three cache lines for translations (one each for the top level, middle level and current translation entry) at around 64 processes we would expect the cache to start taking capacity misses (e.g., $64 \times 4 = 256$). Further experiments show that this inflection point can be made to move with different combinations of base page-size. The very different cache profile of the LF-VHPT avoids this behaviour.



(a) Execution time (lower is better)



(b) Percentage of data references not covered by the TLB (lower is better)



(c) Percentage of data references covered by cache (higher is better)

Figure 4.5: Micro-benchmark with multiple processes walking a small 10 page (160 KiB) region.

One of the major concerns of LF-VHPT is decreased cache utilisation from the larger entries, however for this micro-benchmark LF-VHPT has a *higher* cache utilisation than the SF-VHPT. This is probably because the hash table makes effective use of the entire cache line (as illustrated in Figure 4.3), unlike the SF-VHPT which brings in a considerable amount of unneeded data as it traverses page tables.

4.3.4 Analysis

Best and worse-case microbenchmark scenarios highlight the relative strengths and weaknesses of both formats of HPW. However, even in the extreme cases the overall performance impact of either choice remains minimal. This is because a TLB miss is, even in the best case, expensive — architecture manuals state at least 25 cycles without cache misses or other microarchitectural hazards. From these results we can conclude that even worst-case LF-VHPT performance should remain acceptable.

4.4 SPEC

The SPEC CPU2000 suite is the industry standardised CPU intensive benchmark suite which stresses a system's processor, memory subsystem and compiler [Hen00]. Since the suite is primarily designed for CPU evaluation many of the SPEC benchmarks have a small working set and are of limited use in analysing virtual memory behaviour. However, some of the tests do exhibit interesting behaviour which we analyse below.

4.4.1 SPEC TLB behaviour

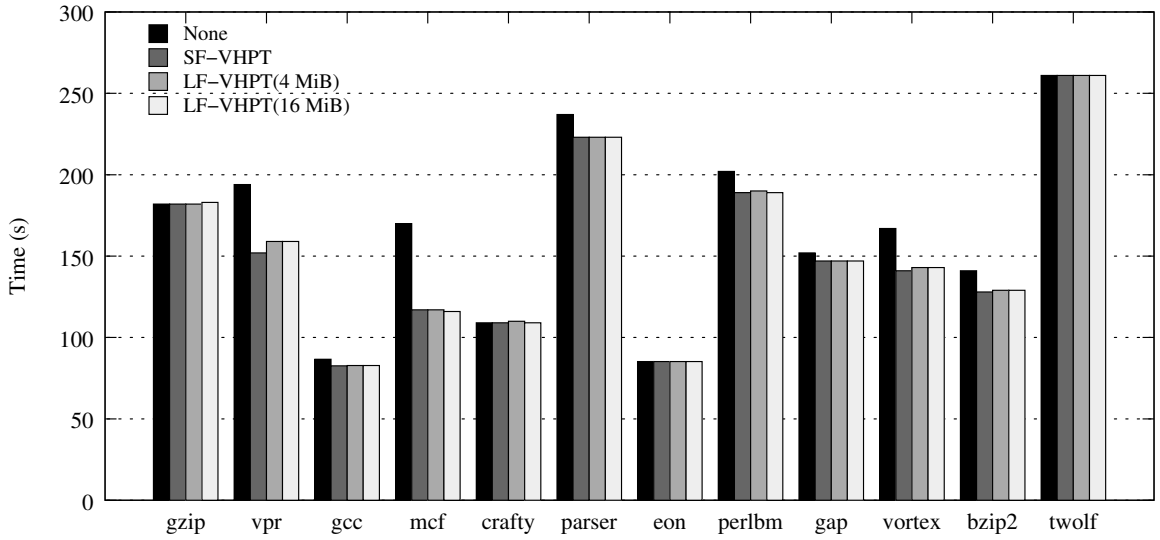
Kandiraju and Sivasubramaniam [KS02] produced an analysis of the TLB behaviour of the SPEC benchmark suite. On their system they found `vpr`, `twolf` and `mcf` were the most sensitive to increased TLB associativity and size. They also modelled the temporal separation (time between TLB misses) and spatial access patterns (range of virtual addresses accessed) of the benchmarks. Results showed that of all the benchmarks `vpr` and `twolf` have the greatest temporal separation between TLB faults and spatial profiles show `mcf` having sporadic long periods of sequential access with `vpr` and `twolf` exhibiting an essentially random pattern of virtual address access.

4.4.2 Results

Run times for SPEC CINT2000 with SF-VHPT and LF-VHPT with varying hash table size are shown in Figure 4.6.

Overall, results show disabling the hardware walker has no effect on four tests (`gzip`, `crafty`, `eon` and `twolf`) and only small performance reductions for others (less than 5% for `gcc` and `gap` and less than 7% for `parser` and `perlbm`). This suggests that for these tests TLB coverage is sufficient and the cost of refilling TLB entries is not limiting the performance of the benchmark. Consequently we expect no benefit from either form of HPW. Overall the results are consistent with Kandiraju and Sivasubramaniam [KS02] — we see the largest penalties with `vpr` and `mcf` which were identified as sensitive to TLB behaviour.

Traditional wisdom states translation hash tables should be large enough to cover translations for physical memory three times, e.g., for the benchmark machine with 2 GiB RAM and 16 KiB pages this is 12 MiB:



(a) Time (shorter is better)

Benchmark	Speed-up		
	SF-VHPT	LF-VHPT(4 MiB)	LF-VHPT(16 MiB)
gzip	1.000	1.000	0.995
vpr	1.276	1.220	1.220
gcc	1.048	1.046	1.046
mcf	1.453	1.453	1.466
crafty	1.000	0.991	1.000
parser	1.063	1.063	1.063
eon	1.000	1.000	1.000
perl	1.069	1.063	1.069
gap	1.034	1.034	1.034
vortex	1.184	1.168	1.168
bzip2	1.102	1.093	1.093
twolf	1.000	1.000	1.000

(b) Speed-up compared to no HPW

Figure 4.6: SPEC CINT2000 results. Values were constant over 3 runs.

	Routing			Placement		
	None	Short	Long	None	Short	Long
Time (s)	71.724	71.803	76.586	122.678	80.901	82.80
% DR[a] satisfied by						
TLB	97.88	98.69	97.94	97.75	97.65	97.68
HPW	0.00	0.00	0.00	0.00	2.34	2.25
Software	2.12	1.31	2.06	2.25	0.01	0.07
total	*100%*	*100%*	*100%*	*100%*	*100%*	*100%*
% L2 TLB miss by HPW[b]	0.00	0.01	0.00	0.00	99.73	96.95
% DR hit in cache	26.95	27.12	27.11	11.12	10.27	10.33
% DR to cache [c]	46.74	46.74	46.74	49.01	49.84	49.83

[a]Data References; e.g., a memory operation issued into the pipeline
[b]i.e., what percentage of TLB misses were then covered by the HPW
[c]data memory read references issued into memory pipeline to be serviced by L1D. Includes integer and RSE loads, but does not include floating-point, VHPT loads or semaphore operations.

Table 4.4: Detailed performance analysis of `vpr`

$$\begin{aligned} hash\_size &= \frac{3 \times memory}{page\_size} \times lvhpt\_entry\_size \\ &= \frac{3 \times 2 \times 2^{30}}{16 \times 2^{10}} \times (4 \times 8) \\ &= 12\text{MiB} \end{aligned}$$

The LF-VHPT hash table is pinned with a single TLB entry and therefore must be a power-of-two size supported by the processor; this results in rounding up to a default 16 MiB table for our benchmark system. The timings from Figure 4.6 show that for the single-threaded SPEC benchmarks no disadvantage is incurred when artificially decreasing to a smaller 4 MiB table size, suggesting hash table collisions are not a significant factor.

4.4.3 Analysis of `vpr`

Of all the SPEC results the LF-VHPT HPW results for `vpr` compare least favourably to its SF-VHPT counterpart. This test simulates code used for creating FPGA chips with a "routing" and a "placement" phase, which are run as separate sequential processes. TLB and *d*-cache statistics from a sample run are shown in Table 4.4.

Routing The routing phase shows high TLB coverage (i.e. high percentage of data references covered by the TLB) but very low HPW usage, suggesting a core amount of frequently accessed data with occasional random accesses requiring a new translation which are very infrequently reused.

In their studies of the SPEC suite Kandiraju and Sivasubramaniam [KS02] show `vpr` has a 98.36% hit rate for a large monolithic TLB, but when modelled on a multi-level TLB show a 43.4% miss rate in the L2 TLB (they only show results for the overall test, rather than the separate components as shown above). This agrees with the hypothesis that most of the references are covered by the smaller L1 TLB and occasional references are largely outside the coverage of even the larger lower translation levels.

A small and dense working set has advantages for the SF-VHPT, since with 64 KiB pages one TLB entry given up to map a page of the virtual linear array covers $\frac{64\ KiB}{8} = 8192$ potential translations. Although the overall percentage of TLB misses which result in HPW hits for the SF-VHPT is only 0.01%, raw figures show around 5 times more hits which can be attributed to this effect (SF-VHPT has 38618 HPW hits, LF-VHPT has 6774 hits).

The results show that cache effects are negligible, with each version maintaining a very similar hit rate. This suggests cache effects of the different HPW formats are negligible, with cache references being dominated by application (rather than OS) data.

Placement The placement phase shows significant benefits from enabling both forms of the hardware-walker and illustrates the high cost of taking software faults. The effects of "free" mapping of adjacent translations with the SF-VHPT are again apparent in a slightly higher HPW coverage of TLB misses. The extra costs of keeping the LF-VHPT table are amortised over the higher HPW hit rate.

This cache exhibits slightly higher cache hits than the no-VHPT case as expected, since it does not have the extra pollution of VHPT traffic. However, it is interesting to note neither long or short-format has a particular advantage in cache hit performance.

4.4.4 Conclusions

SPEC benchmarks are primarily designed for CPU evaluation, thus many of the tests have a small working set and do not stress the MMU and its support structures (as shown by those tests that have little or no penalty despite no hardware-walker support).

A detailed analysis of the worst-performing LF-VHPT test (Section 4.4.3) showed how a particular component of the test exhibiting adverse behaviour was largely responsible for the performance difference. Cache utilisation did not play a role in the overall results, probably because the relatively large caches on the test system absorb the effects of the HPW overhead.

Overall, the overheads presented by using LF-VHPT are minimal and we can conclude that there is potential to make up the small difference through improved efficiencies when loading large pages.

4.5 LMBench

LMBench [MS96] is a suite of micro-benchmarks, each designed to evaluate specific components of a system. Many of the included benchmarks stress the virtual memory subsystem and are thus useful for examining HPW choice. The full complement of results are presented in Appendix B, but here we discuss only significant results.

4.5.1 Latencies

The latency component of the benchmarks exhibits the biggest difference between SF-VHPT and LF-VHPT performance, as illustrated in Table 4.5.

Test	None	LF-VHPT
`execve`	3.0%	2.3%
`fork`	5.4%	4.8%
`mmap`	14.2%	26.4%

Table 4.5: Decrease in performance over SF-VHPT for various LMBench latency tests

Data References % covered by	SF-VHPT	no-VHPT	LF-VHPT
TLB	99.99%	99.99%	99.99%
HPW	0.01%	0.00%	0.01%
SW	0.00%	0.01%	0.00%

Table 4.6: TLB statistics for the LMbench `mmap` read bandwidth micro-benchmark.

The `mmap` latency test makes a single walk of a linear region of memory to establish the overheads of setting up mappings. This is the ideal case for the SF-VHPT and it is therefore expected to perform well. The additional latency increase of the LF-VHPT over the no-VHPT case is also expected, as LF-VHPT requires additional time to instantiate the entry in the separate hash table.

4.5.2 Bandwidth

The second range of benchmarks focuses on bandwidth of various system operations. In general these tests exercise memory in a linear and regular fashion so are best suited to a SF-VHPT, however the tests maintain a small working set so the difference is modest. As evidence of this, results in Appendix B show all but two of the no-VHPT and LF-VHPT results remain within 1% of the SF-VHPT result (the two outliers, being the Unix and `pipe` test, differ by less than 2% and the raw difference in results is within one standard deviation).

Detailed TLB usage for the `mmap` read bandwidth micro-benchmark presented in Table 4.6 illustrates how the choice of HPW has little effect on the overall results. In all cases the vast percentage of the data references performed by the are covered by the L1 and L2 TLB. We see such high coverage ratios because the test reads in very small increments (i.e., bytes, where page size is 16 KiB), hence the proportion of total data references to misses is very high.

Since the LF-VHPT will only be effective for pages that are reused, we can assume that the misses covered in this case are for commonly reused test framework code rather than the core test loop. We consequently see the the SF-VHPT covers more of the TLB faults, but since software faults are required for less than 0.01% of total data references the result is negligible.

In summary, results show LF-VHPT has a less than 1% overhead for bandwidth related micro-benchmarks.

4.5.3 Context Switching

The final component of the LMBench suite tests the overhead of context switching. This range of micro-benchmarks should be very TLB intensive due to the high number of address spaces competing for TLB entries and thus we expect better results from the more TLB-friendly LF-VHPT.

The numeric results from Appendix B are presented in Figure 4.7. In general the graph is highly uniform, suggesting that, similar to bandwidth micro-benchmarks (Section 4.5.2), the overall differences are

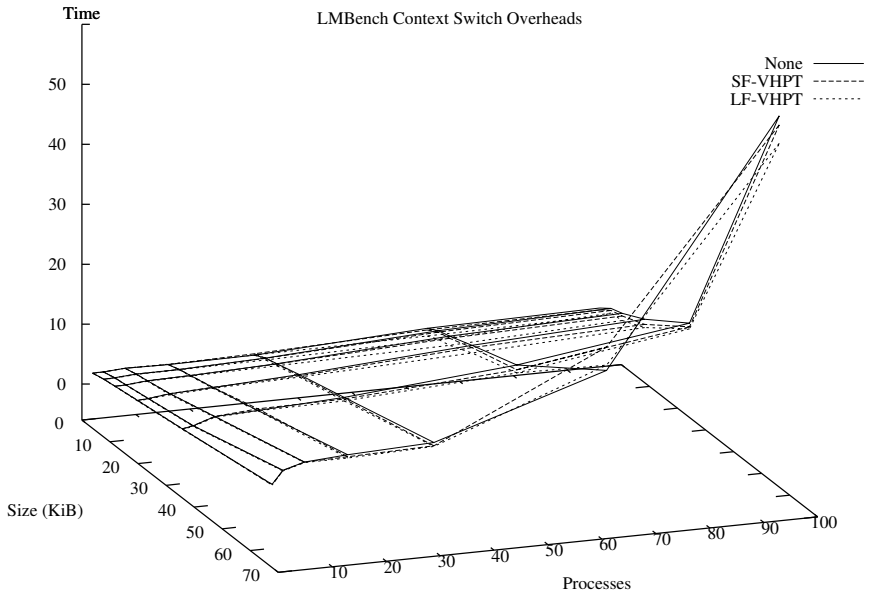


Figure 4.7: Graphical view of LMBench context switching (Appendix B). Results are generally consistent independent of HPW for low numbers of processes with small sizes, but diverge with greater number of larger processes (smaller is better). The LF-VHPT has an advantage (best illustrated in the most extreme case to the lower right) due to inducing less TLB overheads.

negligible.

However, towards the upper-right end of the scale with many large processes some differentiation is evident. This is greatest at the peak of 96 processes of 64 KiB each; here we see the LF-VHPT with the lowest context switch time due to the dual combination of more available TLB space and ability to hardware fill misses. The no-VHPT case is slightly slower, but still beats the SF-VHPT case, again due to more available TLB entries. SF-VHPT suffers at the extreme as each processes becomes active and pollutes the TLB with extra entries to map page tables.

4.5.4 Conclusions

It is difficult to draw general conclusions from micro-benchmarks as they tend to exercise very specific and occasionally pathologic cases. Latency tests (Section 4.5.1) showed the worst results for LF-VHPT but under circumstances known to be adverse. Bandwidth tests (Section 4.5.2) showed SF-VHPT has a measurable but close to insignificant ($<$1%) advantage over LF-VHPT. Conversely, TLB intensive context switching benchmarks (Section 4.5.3) show a small advantage to LF-VHPT in the most extreme cases.

4.6 System Performance

The following benchmarks are of a more general nature and are used to gain insight into overall system behaviour rather than the more memory subsystem intensive benchmarks presented previously.

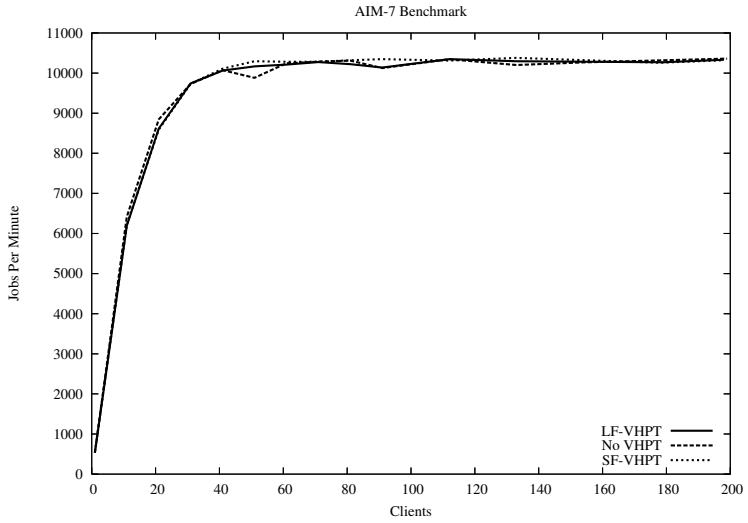


Figure 4.8: Results of the OSDL AIM-7 benchmark from 1 to 200 clients.

4.6.1 AIM

The OSDL AIM-7 benchmark [AIM] is a system-wide benchmark which runs an increasing number of tasks simulating various user workloads. For a well-behaved system we expect to see a sharp increase in jobs per minute (JPM) throughput as the system comes to full load, then a long plateau and eventual decrease in throughput as the system goes into overload.

Figure 4.8 shows OSDL AIM-7 results ranging from 1 to 200 clients. Small variations in the results are accounted for by the design of the benchmark to have clients randomly select their workload profile (CPU intensive, disk intensive, etc.).

Overall the choice of hardware page-table walker, or indeed even turning it on at all, has little impact on overall throughput. A very slight trend of higher throughput for SF-VHPT can be perceived, but raw results are well within one standard deviation.

The ambiguous result is an accurate reflection of performance characteristics on a busy, varied workload multi-user system. The benchmark does have a hardware walker friendly component of work searching and sorting large datasets, but the CPU and disk intensive workloads perturb the system far too much for the gains to be recognised.

4.6.2 Kernel Compile

A typical developer task is compiling a kernel, which involves extracting a compressed archive and many repeated small compilation tasks. The overall process is not expected to be limited by TLB coverage but will be largely CPU limited (and to some extent I/O limited, but ample memory for disk cache largely avoids this). The test does however provide insight into costs of typical address space life-cycle. The build process consists of compiling slightly over 1000 object files which are linked into the final binary image. The results presented in Table 4.7 are an average of 3 runs building a 2.6.18 Linux kernel.

The results show that for the CPU and memory intensive decompression and extraction phase either HPW is approximately an equal improvement. For the build phase the LF-VHPT has a slight penalty, as we might expect due to overheads in maintaining the hash table.

HPW	Extract	Build
None	18.41	708.89
SF-VHPT	15.55	693.33
LF-VHPT	15.60	696.43

Table 4.7: Time (in seconds) for a kernel build benchmark.

Kernel	Indicative SPECweb99
SF-VHPT	94
LF-VHPT	96

Table 4.8: SPECweb99 results

4.6.3 SPECweb99

SPECweb99 [SPE] performs a load test of a web server. For the benchmarking server we used Apache 2 with 5 worker processes. The dynamic content portion of the test was handled by the provided PERL CGI script, but to avoid constantly forking the PERL interpreter was run under FastCGI. The benchmark starts a number of client processes, each requesting a fixed proportion of static and dynamic data from the target machine. The SPECweb99 result is based on the maximum number of connections the server can maintain at given throughput level (≈300Kb/s). The benchmark was produced with a relaxed version of officially required SPECweb99 conditions (run time, etc.) and with an untuned Apache server, but all parameters were constant between the two tests.

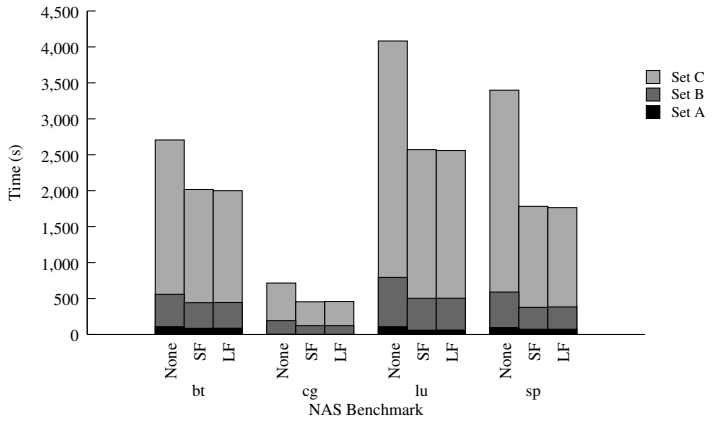
This situation is similar to the high-contention case presented in Section 4.3.3 since the long running Apache processes are each trying to service very similar requests as fast as possible.

As shown in Table 4.8 the overall benefit is small but noticeable, representing an extra two clients sustained at the required rate.

4.6.4 NAS

The Numerical Aerodynamic Simulation (NAS) parallel application benchmarks [BBB+91] are a series of tests designed for performance evaluation of highly parallel machines. The benchmarks represent typical scientific applications; they are long-running, have large working-sets and are processor and memory intensive. For this particular case they are run in a non-parallel fashion and artificially limited to a single processor to best analyse the interactions with the HPW. To reduce the time taken this benchmark was run on a faster 1.5GHz machine with the same specifications as previously mentioned. Tests were compiled with the Intel `ifort` compiler with the highest level of optimisation.

Results are illustrated in Figure 4.9. The various tests represent common fluid-dynamics problems and are run with an increasingly large dataset (from the smallest `A` through `C`). We see that over the considerable life-span of the longest running tests both LF-VHPT and SF-VHPT offer considerable benefits and differ only slightly in the speed-up achieved. The slight deviations can be attributed to interaction between the aggressive optimisation of the compiler (most importantly cache pre-fetching requests) and the different cache profiles presented by by SF-VHPT and LF-VHPT.



Benchmark	Speed-up SF-VHPT	Speed-up LF-VHPT
bt.A	1.285	1.249
bt.B	1.263	1.259
bt.C	1.363	1.381
cg.A	1.892	1.874
cg.B	1.571	1.561
cg.C	1.573	1.563
lu.A	1.830	1.809
lu.B	1.549	1.547
lu.C	1.590	1.599
sp.A	1.316	1.312
sp.B	1.623	1.596
sp.C	1.999	2.033

Figure 4.9: Results of selected NAS benchmark tests for datasets A, B and C. `bt` = Block Triangular, `cg` = Conjugate gradient, `lu` = LU solver, `sp` = Pentadiagonal solver. Speed-up is relative to no-HPW.

4.7 Scaling

Efficient multi-processor and non-uniform memory architecture (NUMA) support is currently future work. A discussion of the many possible implementation strategies and potential effects follows.

4.7.1 Multi-processor

There are two fundamental options for the LF-VHPT hash table — either giving each CPU a private hash table or sharing a single hash table between all CPUs.

A private hash table for each CPU eliminates concurrency concerns when updating the hash table. The OS can keep track of which CPUs the process has run on; flushing unmapped entries therefore involves visiting only the relevant hash tables.

The hardware implementation allows for a shared table by reserving the top bit of the LF-VHPT tag field (Figure 3.5) as a *valid* bit which signals to the HPW if the translation can be inserted. Atomic instructions can be used to disable this bit when the system is updating the other fields of an entry. The implementation must also ensure correct memory fencing operations are followed to ensure the update is seen globally.

Each model therefore has tradeoffs which is somewhat influenced by the affinity of processes to a given CPU. A CPU private hash table can avoid the costs of atomic updates, serialisation and cache coherency and is appropriate for a process with high affinity. On the other hand, a constantly migrating process may benefit from being able to retrieve translations from cache rather than needing the OS to instantiate them in the CPU private hash table.

Threads create further complications by sharing the same address space across multiple CPUs. Multiple tables may produce greater pollution but may also reduce expensive cache coherence traffic. It is difficult to predict *a priori* the overall impact in this situation.

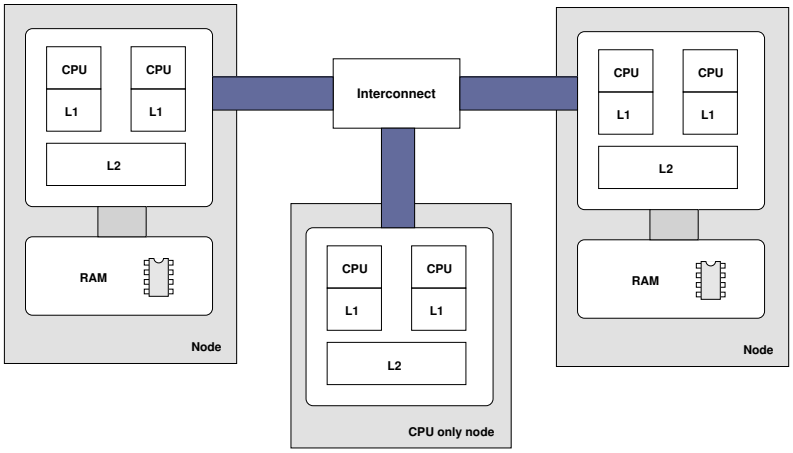


Figure 4.10: Illustration of a NUMA system

4.7.2 NUMA

A NUMA system is a collection of interconnected nodes which present a shared global memory as illustrated in Figure 4.10. For each node, local memory access is preferred to avoid the cost of traversing the interconnect. The interconnect is usually modelled as a crossbar for smaller systems or a hypercube for larger ones.

It is not uncommon for a large NUMA Itanium machine to contain more than 1TiB of physical memory. Given rules of thumb that hash table size should be enough to cover 3-times physical memory, with 16 KiB pages this equates to $\frac{3\times 2^{40}}{16{,}384} \times 4 = 768\mathrm{MiB}$ of hash table per-processor. Current NUMA nodes often have 4 physical CPU sockets which, with multi-core packages, offer potentially 16 or more processors. Apart from the tremendous cost of allocating this memory, the large size makes it quite likely the hash tables will overflow the node-local memory and incur remote access penalties.

However, a shared hash table would also not seem like a practical solution for a component as critical to overall performance as translation performance. A un-replicated, globally shared table is by its nature node-local to only one node and all others will experience increased latency accessing it; on a large system this penalty may be so great as to make walking the quite possibly node-local OS page-tables cheaper. There would also be increased cache coherency traffic to keep the replicated data in the many processor caches coherent. Sharing access to such a highly utilised data structure is likely to lead to heavy contention and delays from locking and/or serialisation.

There are also many implementation challenges posed by large commercial hardware such as the ability to handle hot-plugged memory, CPUs and nodes. Another complicating factor is *compute only* nodes which have no node-local memory.

The most plausible solution would be for the hash table to cover translations only for node-local memory. This constrains the per-processor hash table to a practical size which will reside in node-local memory; that is if a translation describes a remote address it is not entered into the hash table. With current versions of Linux there is an implementation challenge to avoid large overheads when determining if an address is node-local. Existing memory layout information provided for Linux is not easily accessed from the fast-path fault handlers and would require a transition from assembly to C. Relocating the information would require constraints on its ability to cause faults when accessed, necessitating it be in pinned per-CPU data. It may also be possible to constrain addresses based upon platform specific knowledge about possible

physical memory location.

4.8 Conclusion

The overheads of LF-VHPT over the current SF-VHPT implementation are at worst small and under extreme TLB load show a small performance improvement. While the large data caches appear to ameliorate concerns over larger translation entries with LF-VHPT the implementation does suffer from not being the primary source of translation information, leading to higher latencies when resolving faults. The LF-VHPT poses particular scalability challenges, especially when combined with a NUMA system.

Chapter 5

Large-Page Implementation

This chapter describes the implementation of a transparent large-page infrastructure for Linux. Section 5.1 gives an overview of the Itanium fault handling process. Section 5.2 describes the modifications required to transparently support large pages and Section 5.3 gives details of the translation replication scheme. Section 5.4 and Section 5.5 describes the interaction with the Itanium HPW and implementation within Linux respectively. Section 5.6 concludes with some discussion of page allocation.

5.1 Fault Handling Overview

Loading a translation on the Itanium architecture involves four main registers, as illustrated in Figure 5.1. On a TLB miss the hardware pre-loads the interrupt fault address (IFA) register with the faulting address (vpn) and the interrupt insertion register (ITIR) with the default page size (ps) information as taken from the page-size set in the region register of the region the faulting virtual address lies within. The software fault handler then loads a general-purpose register with translation information from the operating system page tables. This register (GR in Figure 5.1) describes the physical page number (ppn), access rights (ar), privilege level (pl), the present (p), dirty (d) and accessed bits (a) and memory attributes (ma) such as cache policy. Finally, the appropriate region register (RR) for the faulting address gives the region id (rid) for the translation.

Once these four registers contain appropriate information, the processor is instructed by software to insert the translation into the TLB.

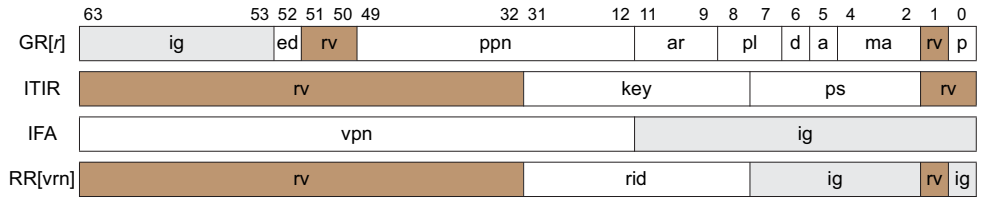


Figure 5.1: Registers used for translation insertion (replicated from [Int00]). See Section 5.1 for field descriptions.

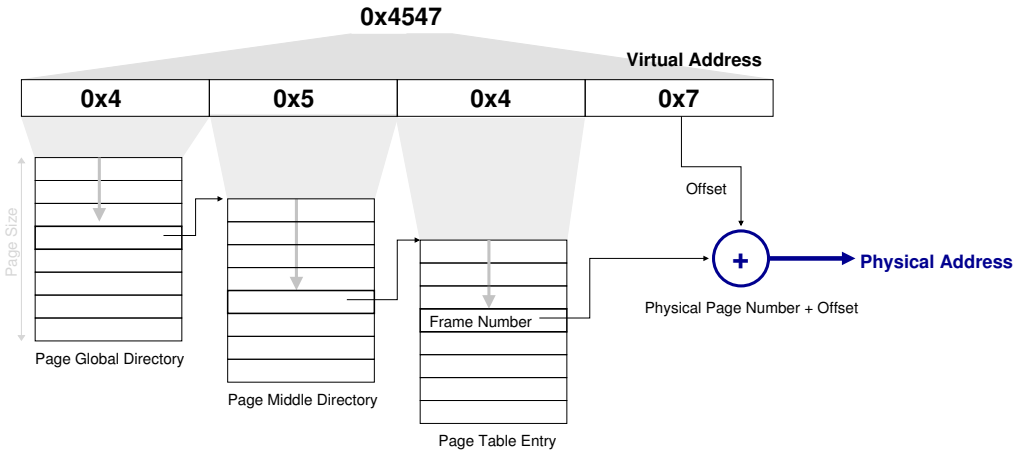


Figure 5.2: Three level page-table operation. Bits from the virtual address give an index into a level table. Upper levels provide a pointer to the next level table, whilst the leaf entries point to a physical page.

5.1.1 Page Table Operation

Linux uses a hierarchical multi-level page-table as shown in Figure 5.2. In standard operation each level directory is one page in size and each entry is a 64-bit pointer. In unmodified Itanium Linux, the leaf entries (which describe the virtual-physical translation information) mirror the GR format from Figure 5.1. This allows sharing of the translation entries between Linux and the SF-VHPT HPW and also facilitates quick fault handling since the GR register can be loaded directly from the leaf entry.

5.2 Page Size Modifications

As described in Section 5.1, at fault time the hardware loads the ITIR register with the preferred page size for the region the faulting virtual address lies within. Unmodified Linux supports only a single page size per region, so this value is always correct. For example, the HugeTLB infrastructure discussed in Section 3.5 marks a particular region as having a larger page size and therefore any faults in this region are known to be backed by this larger page size. To support multiple page sizes *within* a region the operating system must maintain a record of the relevant page size for any given virtual address and "reset" the ITIR with this information before TLB insertion.

5.2.1 Page Table Modification

The first option for maintaining page sizes for translations is a complete re-implementation of the Linux page table with data structures especially designed for multiple page-size support. Re-engineering is likely to be able to optimise fault handling; for example a guarded page-table has been shown to be a high performance alternative [LE95]. Unfortunately, current Linux implementations "open-code" the page table abstraction, and thus the page-table implementation is not sheltered by an easily modifiable API. Consequently re-implementation would require an unpalatable amount of change.

A translation replication scheme (as discussed in Section 2.4.2) is an appealing alternative. Under this scheme translation (leaf) entries of the standard hierarchical page-table are modified to hold the size of the page the translation entry currently resides within. The disadvantage of this scheme is wasted storage

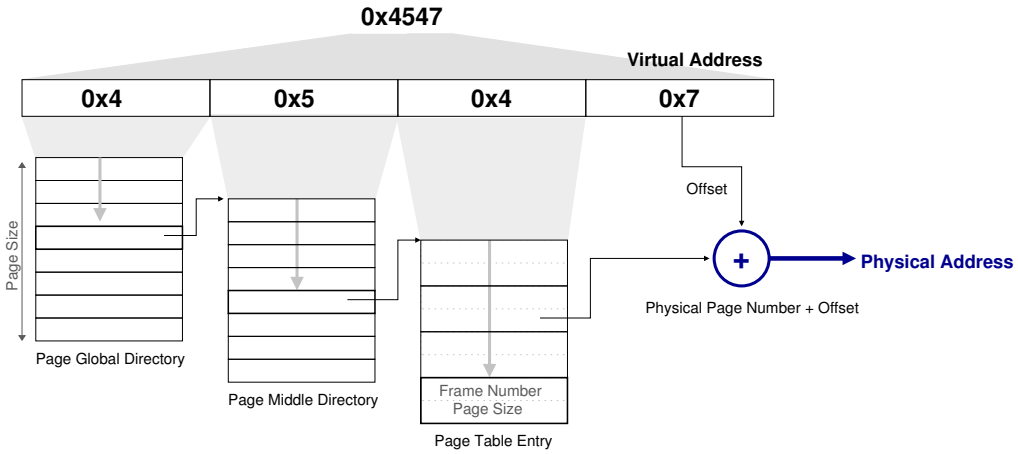


Figure 5.3: Example of double-size translation entries. In contrast to Figure 5.3, note the leaf level has been expanded with an extra word to facilitate storage of extra information such as page size or protection key information.

overheads from the replicated translation entries, but this is offset by the significant advantage of reduced modification of the OS virtual memory code. This approach was taken in the Shimizu and Takatori [ST03] work, possibly inspired by similar approaches in HP-UX [SMPR98].

5.2.2 Translation Modification

To maintain the existing core page-table structure an implementation must find room to keep additional page size information. Although the top ten "ignored" bits of the existing translation entry (Figure 5.1) are sufficient to store the page size, additional features such as protection keys require even more space which can be gained only by increasing the size of a translation entry.

The smallest possible increase in translation entry size is a CPU word, which effectively doubles the existing size. With an additional word in each translation entry the ITIR register can be "shadowed" and directly loaded by software, rather than having to move the page-size information into the ITIR via a separate general purpose register. This gives the potential to save cycles on the fault-path. This scheme is illustrated in Figure 5.3.

A negative impact of doubling the translation size is a reduction of available virtual address space by one bit. Consider that an unmodified 16 KiB (2^{14}) page-size system with three levels and 64-bit (8 byte) pointers can address a total of 128TiB of virtual address (Equation 5.1).

$$\left(\frac{2^{14}}{8}\right)\left(\frac{2^{14}}{8}\right)\left(\frac{2^{14}}{8}\right)2^{14} = 2^{47} = 128\text{TiB} \tag{5.1}$$

Using double-size 128-bit (16-byte) entries for leaf entries thus allows one less bit and reduces the available virtual address space to a more modest 64TiB. However, this limitation can be exceeded by adding an additional fourth level to the page table, which is already an optional feature of unmodified Itanium Linux. With 16 KiB pages this adds another full 14-bits to the virtual address space, giving a total of 1EiB (Equation 5.2).

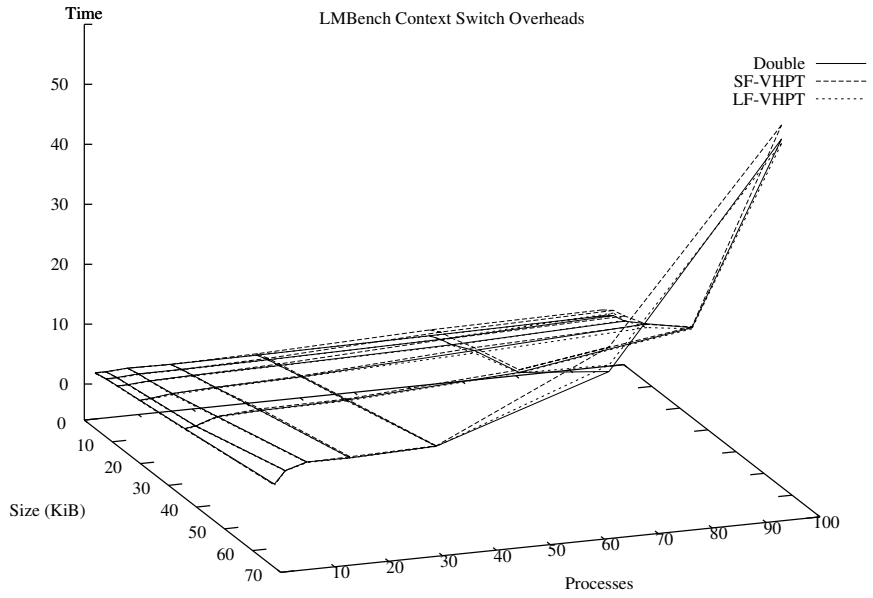


Figure 5.4: Graphical view of LMBench context switching with double-size translation entries (Appendix B). Lower is better.

$$\left(\frac{2^{14}}{8}\right)\left(\frac{2^{14}}{8}\right)\left(\frac{2^{14}}{8}\right)\left(\frac{2^{14}}{16}\right)2^{14} = 2^{60} = 1\text{EiB} \tag{5.2}$$

Dedicating more space to page tables increases their cache footprint, which could potentially have an adverse effect on overall performance.

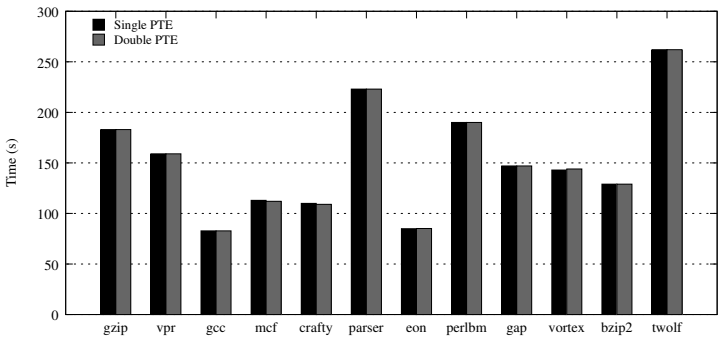
Analysis

Double-word translation entries were implemented as an extension to the LF-VHPT patches.

LMBench The LMBench micro-benchmarks (presented in full in Appendix B) make precise measurements of small effects and thus illustrate the specific costs of double-size translation entries. For the processor and process tests the slow-down is a noticeable but small 1–2%. The context switching benchmarks show a larger penalty which can be attributed to the larger page tables. The results show a higher standard deviation (and thus less uniformity) which suggests the larger page tables are contributing to cache pollution.

Figure 5.4 illustrates the results in the manner of Figure 4.7. For the larger context switching tests the graph shows the double-word translation kernel (based on LF-VHPT) still maintains its advantage over the SF-VHPT, but overall performance remains extremely close to LF-VHPT.

SPEC SPEC CPU2000 results (Figure 5.5) are indistinguishable from an unmodified LF-VHPT single-word translation based system. The primary performance concern was higher cache pollution from the extra word in each translation entry. Although there is some indication of this on the microbenchmark results the overall effect appears not to be significant enough to perturb results on these longer running and more intensive tests.



Test	Speed-up
gzip	1.000
vpr	1.000
gcc	1.000
mcf	1.009
crafty	1.009
parser	1.000
eon	0.998
perl	1.000
gap	1.000
vortex	0.993
bzip2	1.000
twolf	1.000

Figure 5.5: Double-word translation entry SPEC CPU2000 results. Speed-up is relative to single-word PTE entries.

Discussion

Initial investigation suggests doubling of translation entries could be done with a generally small overhead. This approach would be required for advanced features such as protection keys (Section 3.1.1) but, as described above, not strictly necessary for supporting storage of differing page sizes. Undertaking this approach for large-page support means decreased maintainability and increased divergence from base Linux code and thus the alternative approach of reusing the available reserved bits appears to be a better solution.

5.3 Translation Replication

As described in Section 5.2.1, replication of translation entries provides a minimally invasive method of providing transparent large-page support in Linux. An overview is presented in Figure 5.6. The operating system keeps translations for each sub-page of a large-page mapping, but the hardware (i.e., the TLB and HPW) treats the mapping as a single large page.

To illustrate the methodology, consider an `mmap` of anonymous memory. Linux implements anonymous `mmap` "lazily" by initially filling out blank (not-present) translation entries in the page table and only backing virtual pages with physical pages on reference. It is therefore possible to mark each of the not-present translation entries with a size (based on the size of the `mmap`) during the initial setup phase of the `mmap` call. The second part of the operation therefore requires checking the allocated page-size when handling a translation fault. At this time a request for contiguous memory to back the large page is made, and each of the entries marked as present. This large-page translation is then loaded into the TLB.

This implementation has a number of advantages:

1. The OS retains the concept of each base page having a single translation entry. OS virtual memory code needs to be modified to understand that an operation on a single translation entry may need to be propagated to all sub-pages of a large page, but these changes are practical and self contained.

2. The existing highly efficient fault-path is, for the most part, retained. Alternative schemes often require additional checks as the page table is walked to determine page size; e.g. if the upper levels

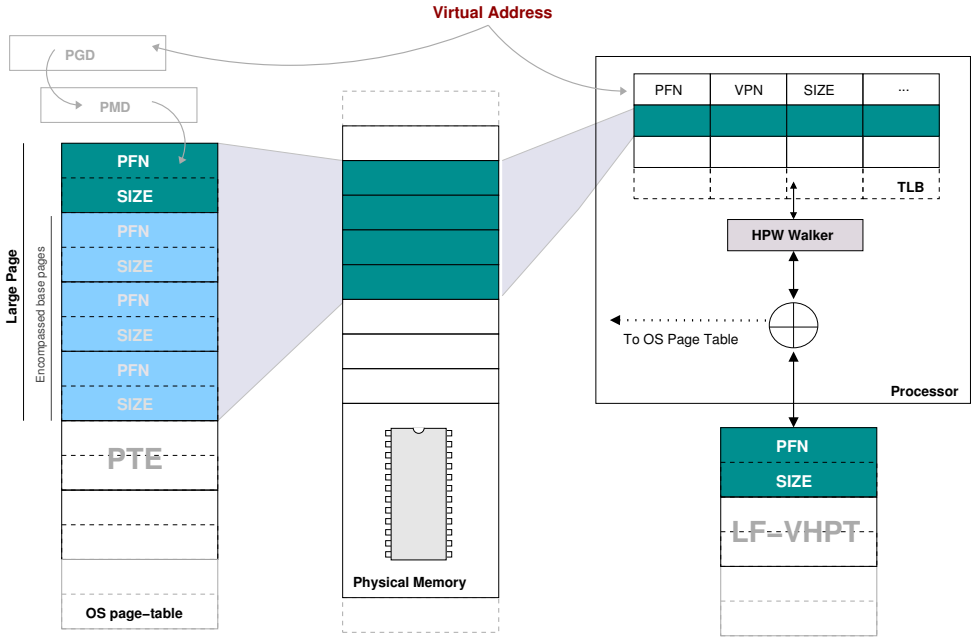


Figure 5.6: Translation replication. Translation entries are expanded to include a page size; thus the OS page tables keep a record of each sub-page. When translations are loaded to the TLB, hardware treats the translation as a single large page.

can be either a pointer to a lower level *or* a translation entry additional checks are required when walking the page table.

3. Physical memory allocation is abstracted from the large page implementation. Having sufficient free contiguous memory is required for efficient large-page support, however keeping free memory contiguous can be can considered separately. This abstraction means policies for allocation, avoiding fragmentation, etc. are not needlessly tied to large-page implementation concerns. This is discussed further in Section 5.6.

4. In the worst case scenario of no contiguous memory being available the failure case is straightforward. The large page can be split recursively until enough contiguous memory is found, or it will reach the termination case of being split to the base page-size.

5.4 Hardware Interaction

5.4.1 Page-Table Walker

A full introduction to the Itanium HPW has been given in Chapter 3. Below we examine the challenges presented by the interaction of the two forms of HPW and large-page support.

Short-Format VHPT

As described in Section 3.3.4, the SF-VHPT reserves the top of a region's address space for a virtual linear page-table. On fault the hardware can check for a translation entry by simply dividing the faulting virtual

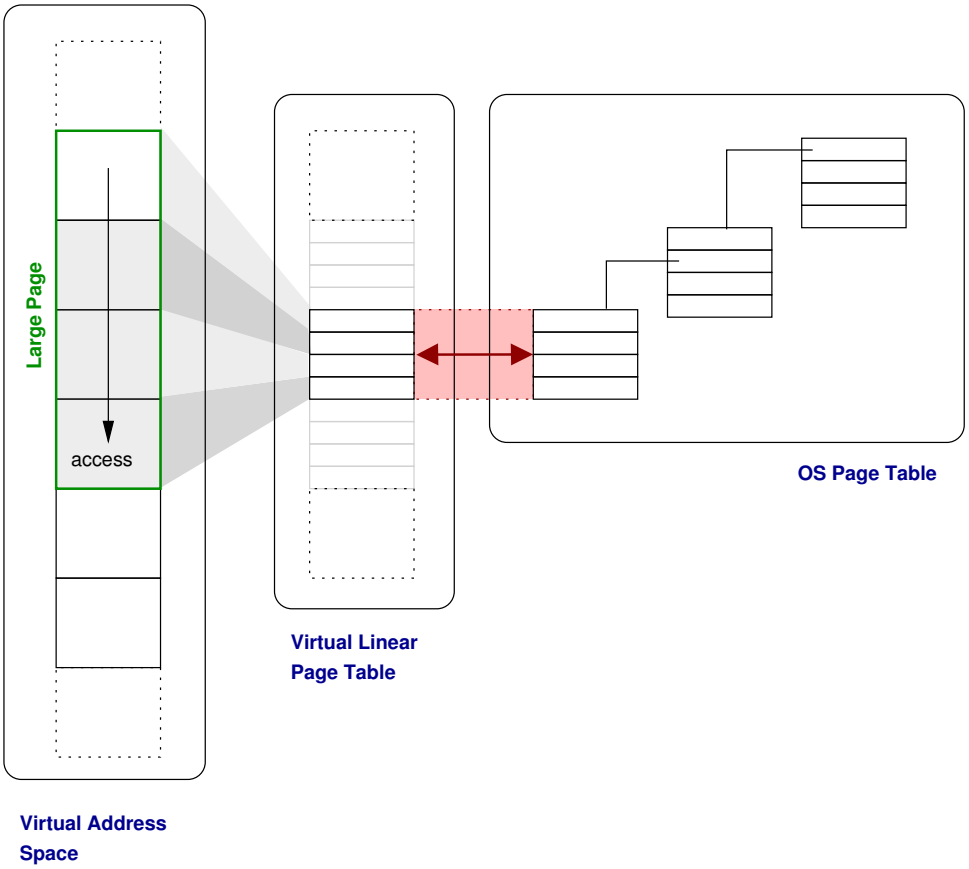


Figure 5.7: When using the SF-VHPT a TLB entry is required to map the translation entries from the operating system page table to the virtual linear page-table, illustrated by the double-headed arrow. It is expected this translation will provide quick access to neighbouring translation entries when walking linearly through the address space. However, using a large page means the virtual linear page-table is under utilised, increasing the relative cost of keeping the page mapped. In this example, without a large page the virtual linear page-table TLB entry gets used four times; with the large page it gets used only once.

address by the page size of the region and checking this offset in the virtual linear page-table. In Linux the virtual linear page-table maps back to the operating system page tables directly.

The SF-VHPT HPW provides no support for loading a translation with a page size different from the page size of the faulting region. Thus the only solution is to keep the underlying OS translation entries for large pages in an invalid state to avoid the HPW inserting them automatically. The SF-VHPT therefore introduces a number of potential inefficiencies when using multiple page sizes:

1. If the page of the virtual linear page-table does not have a TLB entry mapping it to the underlying OS page-tables the fault handler will map one. The penalty of taking an extra TLB entry is assumed to be recovered by allowing quick access to nearby translation entries on the assumption that faults will exhibit high spatial locality. This assumption may not be true when using a large page, because accesses which previously may have covered several pages may now be covered by a single translation. This means the extra TLB entry is under-utilised and increases its marginal cost.
2. If the page of the virtual linear page-table is mapped the hardware will be "fooled" into not loading a large-page translation because there is no way to communicate the page size. The processor has wasted time checking for a translation that it will not be able to fill.

One advantage is that the virtual linear page-table can still be used by the OS fault handler. As described in Section 5.5.5, the OS fault handler still attempts to access the translation entry via the virtual linear page table using the `tpa` instruction. The OS accessing the virtual linear array has two possible outcomes:

1. The virtual linear array page is not mapped, causing a *nested fault*. This fault will cause the virtual linear page to be mapped, requiring a walk of the OS multi-level page table to find the correct page table leaf page. After this is complete the original translation can be inserted.
2. The translation entry is correctly accessed through the virtual linear page-table.

Thus whenever a nested fault is avoided, so is an expensive walk of the OS multi-level page table. When inserting the translation, unlike the hardware walker the OS code is able to examine the entry to establish if it is a large-page and set the TLB accordingly.

Long-Format VHPT

When in long-format mode, the hardware checks a per-processor hash table filled with translation information on fault. Each hash table entry in the LF-VHPT is four words (as-per Figure 5.1) and contains page size information. This allows the hardware fault handler to load translations with any page size.

As described in Section 3.4 the hash function locates the entry partially based on the base page-size of the region the faulting address is within. Therefore, in order to implement complete hardware reloading support each sub-page of a large page requires an entry in the hash table as illustrated in Figure 5.8. This suggests two solutions.

Firstly, the implementation could fill every possible hash table slot for a large page with translation information. This means that no matter where a fault happens, the hardware can load translation information without software intervention. On a single processor, the main concern is increasing cache pollution (and possibly increasing hash collisions) by writing to the hash table for many entries that are unlikely to be used. Since the hash table is kept per-processor we might also consider replicating the entry for the other processors in the system to cover the case of the process migration. This would seem unlikely to improve performance; it will incur more cache pollution (which is already a concern due to

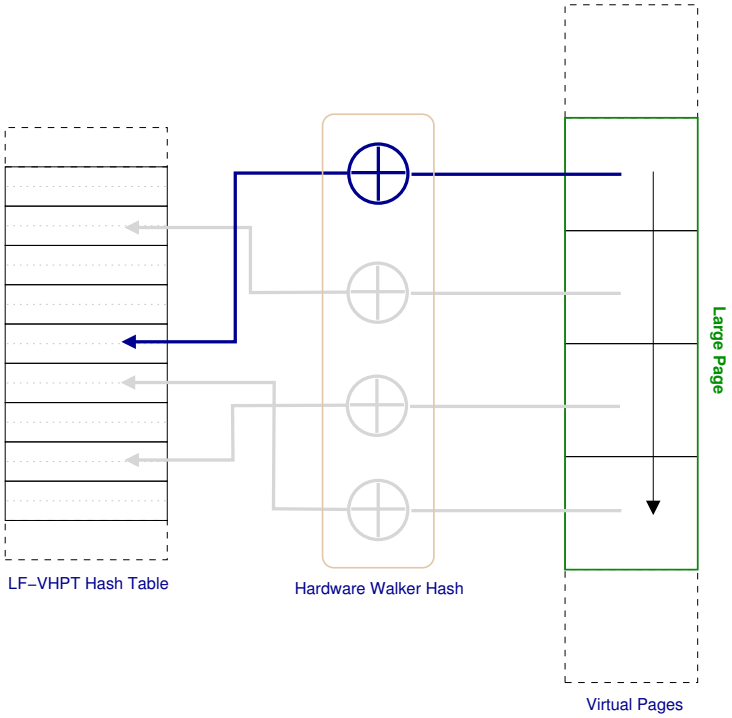


Figure 5.8: Sub-pages of a large page will hash to different entries in the long-format VHPT. Sub-page size is dependent on the region of the faulting address, so can not be modified dynamically. Filling out the extra entries in the hash table will increase cache pollution and, if access proceeds in a linear fashion will not provide any performance benefits, as sub-pages are not likely to be faulted on.

larger entries), increase the fault handling time as more data is written to memory, increase potential for lock contention and will not scale as more processors are added. Additionally, hash table memory may not be node local in a NUMA system. The additional penalties for reading or writing to non-local memory may be larger than simply handling the fault via the OS.

Thus the second option is to simply fill in only the first base-page of a large-page in the hash table of the faulting processor on the assumption access will generally be localised and linear. For the usual case this would seem to be a suitable approach, however if an application accesses data in a more haphazard fashion or the TLB entry for the large page is evicted during processing this will require falling back to slower software faults. In this case, as with the invalid short-format entries, there is also extra overhead from the hardware probing for a translation that can not be filled. Our implementation takes this approach.

Although the hardware can load translations from the hash table directly, populating the hash table requires traversing the OS page tables as described in Section 5.5.5. The LF-VHPT does not have the advantage of being able to “short-cut” access to the translation entries via the virtual linear page-table afforded to the SF-VHPT.

Comparison summary

A summary of the advantages and disadvantages for each form of HPW are presented below.

HPW Model	Advantage	Disadvantage
SF-VHPT	The SF-VHPT has no ability to load anything but the standard base page-size	Minimal modifications are required to keep large page translation entries in an invalid state so they are handled by software.
	The OS software fault handler can use the virtual-linear page-table to quickly access translations	Large page entries must be disabled in the virtual-linear page-table, but the HPW has no knowledge of this and will still waste time checking them
LF-VHPT	Single pinned entry for translation hash table provides for more TLB entries.	OS must maintain translation hash table separately from internal page tables.
	HPW can directly load translation entries from hash table with an arbitrary page size	OS fault handler must always do a full page-table walk to locate translation entry

5.4.2 Alignment

On almost all commercial hardware, translations must be *naturally aligned*; i.e. the virtual address modulo the page size must be equal to zero. Any mapping naturally aligned is also naturally aligned for a smaller page-size boundary (e.g., a 1 MiB naturally aligned page is also naturally aligned for 512 KiB, 256 KiB, 128 KiB and so on). To provide maximum possibilities for instantiating large pages, choosing placements on large natural alignment boundaries is useful.

This can become a major constraint on a 32-bit system such as x86 where aligning to a 4 MiB page size can rapidly lead to running out of virtual address space. On a 64-bit platform such as Itanium the problem is less pronounced, but making the address space more sparse by spreading mappings out is sub-optimal because it reduces spatial locality, which the page tables and HPW are designed to take advantage of.

5.5 Implementation Details

5.5.1 Prior Work

The generic (non-architecture dependent) large-page support is heavily based upon the work by Shimizu and Takatori [ST03]. I re-implemented their work on current kernels, originally using the Itanium with LF-VHPT as a base. As I finished moving the work to the SF-VHPT I became aware of simultaneous work by John Marvin [Mar]. Since the code shared the same parent, in an attempt to reduce duplication the final work is based on a merge with his published patch.

5.5.2 Scope

The scope of the implementation is limited to the `mmap` of anonymous memory. File-backed memory maps do not gain benefits from large pages, meaning for this implementation program code is not mapped with large pages (this also means instruction TLB performance is unaffected). However, memory requested via the standard `malloc` call is allocated from a pool of anonymous mapped memory, meaning it is mapped with large pages.

5.5.3 Tuning Parameters

Although the implementation is transparent to system users, there are still a number of useful tuning parameters. These parameters are created by the large-page implementation, exported via `/proc/sys/` and can be tuned with `sysctl(8)`.

Page size selection A bitmap provides the ability to restrict page order (and hence page size). Bit 1 of this bitmap represents validity of an order-1 page, bit 2 an order-2 page and so forth.

Alignment padding As mentioned in Section 5.4.2, aligning mappings on larger boundaries can help with alignment of larger pages.

The code achieves this by padding the distance between allocations, as explained in Section 5.5.4.

Statistics The implementation also provides statistics for pages pre-allocated in virtual space, large pages faulted in and large pages unable to be satisfied due to insufficient contiguous memory.

5.5.4 Initial `mmap`

A call graph of the functions to implement `mmap` is presented in Figure 5.9. Functions in grey are unchanged, the other functions contain minimal changes as described below.

- The user makes a `mmap` system call, requesting a region of memory. The current implementation handles only `MAP_ANONYMOUS` (i.e. memory backed, rather than file backed) requests.
- `arch_get_unmapped_area` is called to find a region of virtual memory for the new mapping.
- Linux implements `mmap` with an indirection through the `mmap2` system call which requires the mapping size specified in pages, rather than bytes. Calling `mmap2` directly can be useful on some systems when implementing large file support.
- `do_mmap_pgoff` is called to find a region of virtual memory for this request. It calls into the architecture-specific function `arch_get_unmapped_area`.
- `arch_get_unmapped_area` walks the processes VMA list (see Figure2.4) to find an appropriate virtual location for the new allocation. During this loop, the code is modified to pad from the end of any previous allocations to a specified alignment (kept in the `/proc` tunable variable `superpage_vm_align`). This ensures all starting addresses will be aligned on a given boundary.
- After choosing an appropriate address `do_mmap_pgoff` registers a VMA to describe the region as taken.
- Finally the new function `make_ptes_present` is called to mark all base page-size translation entries within the mapped region with a page size; for example mapping a 4 MiB region would require marking each of the 256 16 KiB sub-pages. This function attempts to find the largest pages possible to map the region based on its location and alignment restraints.

Figure 5.9: `mmap` call chain graph

5.5.5 Page Fault

SF-VHPT

With SF-VHPT the architecture can raise the following faults:

`vhpt_miss` is raised when the HPW attempts to access a page of the virtual linear array which is currently not mapped.

On an unmodified system this fault is handled by walking the process page-table to find the translation entry for the original faulting address. Assuming it is valid two entries are inserted into the TLB; the translation information for this fault and a translation for the leaf page is mapped for the virtual linear page-table. When using large pages the process is similar, except extra information about the page size is extracted from the translation entry, and the `itir` register re-loaded with this size.

`[d|i]tlb_miss` is raised when the HPW attempts to load a translation entry but finds it is invalid. The fault handler attempts to access the translation entry via the virtual linear array, a situation which may cause another TLB miss if the virtual linear array page is unmapped. This “double” fault is referred to as a nested fault and is handled by walking the page table as described for `vhpt_miss`. The handler then checks the present bit of the translation, and if not set will then call the OS page fault handler (described in Section 5.5.5).

When using large pages this fault is raised for otherwise valid translations with a non-base page-size, since the HPW has no ability to insert translations with a page size different from the region of the faulting

address. Thus the exit path of the handler is modified to load the ITIR register with page-size information from the translation entry before insertion.

LF-VHPT

The LF-VHPT hash table is kept pinned with a single fixed TLB entry, thus a vhpt_miss or nested TLB fault can not be raised. The [d|i]tlb_miss fault handlers walk the page table to find the appropriate translation entry. If the translation is not present or is invalid, the OS page fault handler is invoked.

Once satisfied, the translation is inserted in the LF-VHPT hash table (hopefully to be used again) and into the TLB. Large-page support requires extracting the page-size information from the translation entry to correctly modify the LF-VHPT entry and load the ITIR register.

OS page-fault handler

If the fault cannot be resolved from the existing page table the OS fault handler is invoked. The major core functions involved with handling a page fault are shown in Figure 5.10. Again, functions with no modifications are in grey.

- ia64_do_page_fault sanity checks the page fault.
- handle_mm_fault is the generic, architecture independent code for handling a page fault.
- handle_pte_fault : decides what "type" of fault has been taken, for example a page swapped to disk, a fault on a region backed by anonymous memory or a fault on a region backed by a file. It then calls the appropriate helper function.
- do_anonymous_page is the helper function for anonymous memory faults (e.g., in a region mmapped as MAP_ANONYMOUS). This function establishes the page size for the fault (as stored in the translation entry).
- alloc_pages requests physical memory, and is passed the order of the page size requested.
- If there is sufficient contiguous free physical memory to satisfy the request a success value is returned and do_anonymous_page marks each of the sub-pages as present in the page table.
- If the allocation cannot be satisfied down_pte_sp is called to divide the large page. This function walks along the sub-page entries of the large page and marks each as the next lower page-size. The request is then retried and the process continued until either the page is allocated or, once the allocation reaches the base page-size, an out of memory error is returned.

Once complete, the page fault handler restarts the faulting process which will attempt to access the address again. The translation entry should now be valid and able to be loaded by either the HPW or the fault handlers.

5.6 Page Allocation

This work does not consider details of page allocation, however below we describe a number of areas under active development.

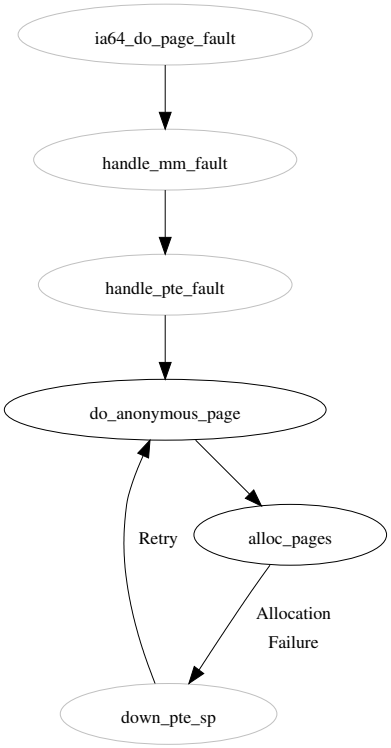


Figure 5.10: Page fault call-chain graph

5.6.1 Defragmentation

Problems stemming from memory fragmentation were previously discussed in Section 2.3.1. A full discussion of techniques is beyond the scope of this report, but better support for large contiguous allocations via defragmentation of memory is currently an area of active research within the Linux community [GW07]. The large-page implementation presented currently interfaces with the page allocation subsystem in only one place; the call to `page_alloc()` illustrated in Figure 5.10. Advanced defragmentation support would give this call a greater possibility of succeeding, especially when higher-order pages are requested.

5.6.2 Clustering

As described in Section 3.5, the Linux shared memory architecture is embedded within the VFS layers. The assumption of a single, consistent page size is prevalent throughout the design of Linux, particularly within the VFS and page cache API. The ability to remove this assumption would provide for a number of benefits.

Firstly, `shmfs` could be modified to request larger physically contiguous anonymous shared memory allocations. Allocations could then be mapped with larger TLB entries by the same translation replication process as described above. This would avoid the situation of having to reserve large pages as separate pool (HugeTLB) but does require additional controls to avoid excessive fragmentation. Secondly, allowing file systems to transfer larger contiguous blocks into the page cache may provide a significant performance boost.

Modifications to these layers have been proposed but have so far not achieved wide support. The general approach is some form of *clustering* [Irw03], where base pages are tracked as part of a larger allocation

(analogous to the translation replication scheme described in Section 5.3). The most recent work by Christoph Lamenter of SGI tags multiple pages as part of a single larger page and modifies several file systems to move data in larger sizes. At this stage the work does not handle the modification of translation entries and fault handlers to instantiate these larger pages in the TLB.

Chapter 6

Large-Page Evaluation

This chapter presents a detailed analysis of the transparent large-page implementation described in Chapter 5.

The analysis follows the outline presented in Chapter 4. Section 6.2 shows the results of micro-benchmarks crafted to exercise extreme cases and thus to highlight best and worse case scenarios. Section 6.3 presents results of the CPU and memory intensive SPEC benchmarks to provide analysis of standard loads. Section 6.4 presents the results of a more varied range of single and multi-process workloads to illustrate the performance under more general conditions. Section 6.5 presents a summary of the benchmark results and Section 6.6 concludes the chapter.

6.1 Test Environment

The test environment is the same as that described in Section 4.2.

6.2 Extremes

The results below are from micro-benchmarks designed to stress the large-page implementation. They are very TLB intensive and hence provide a best case scenario for the implementation.

6.2.1 Matrix Transposition Benchmark

A favourable micro-benchmark for large pages is matrix transposition, which equates to large stride data movements creating high TLB pressure. The process involves copying the rows of a given matrix to the columns of a second matrix, as illustrated in Figure 6.1.

A micro-benchmark was written to read the incoming matrix (`A` in Figure 6.1) in either row-major order (`0,1,2,3,4...`) or column major-order (`0,3,6,1,4...`). Reading in row-major order increases cache hits because of locality of adjacent values, whilst reading in column-major order will present lower cache use and higher cache pollution due to the large stride. Writes therefore go to the second matrix in the

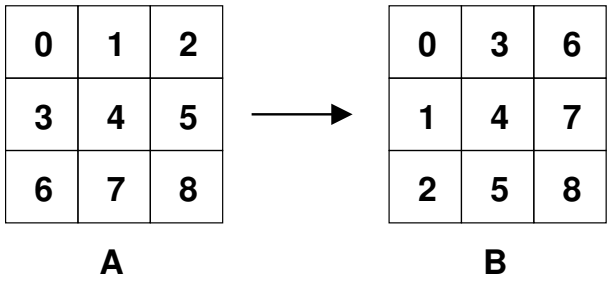


Figure 6.1: Matrix transposition. Rows of matrix `A` are transferred to columns of matrix `B`.

Row Major Order

```
1   for (i = 0; i < dim; i++)
2   {
3     jdim = 0;
4     for (j = 0; j < dim; j++)
5     {
6       a[jdim + i] = b[idim + j];
7       jdim += dim;
8     }
9     idim += dim;
10  }
```

Column major order

```
for (j = 0; j < dim; j++)
{
  idim = 0;
  for (i = 0; i < dim; i++)
  {
    a[jdim + i] = b[idim + j];
    idim += dim;
  }
  jdim += dim;
}
```

Figure 6.2: Pseudo-code for the core of the matrix transposition micro-benchmark

opposite manner, but write-buffering keeps this overhead more consistent. The micro-benchmark was based on one previously used by Shimizu and Takatori [ST03].

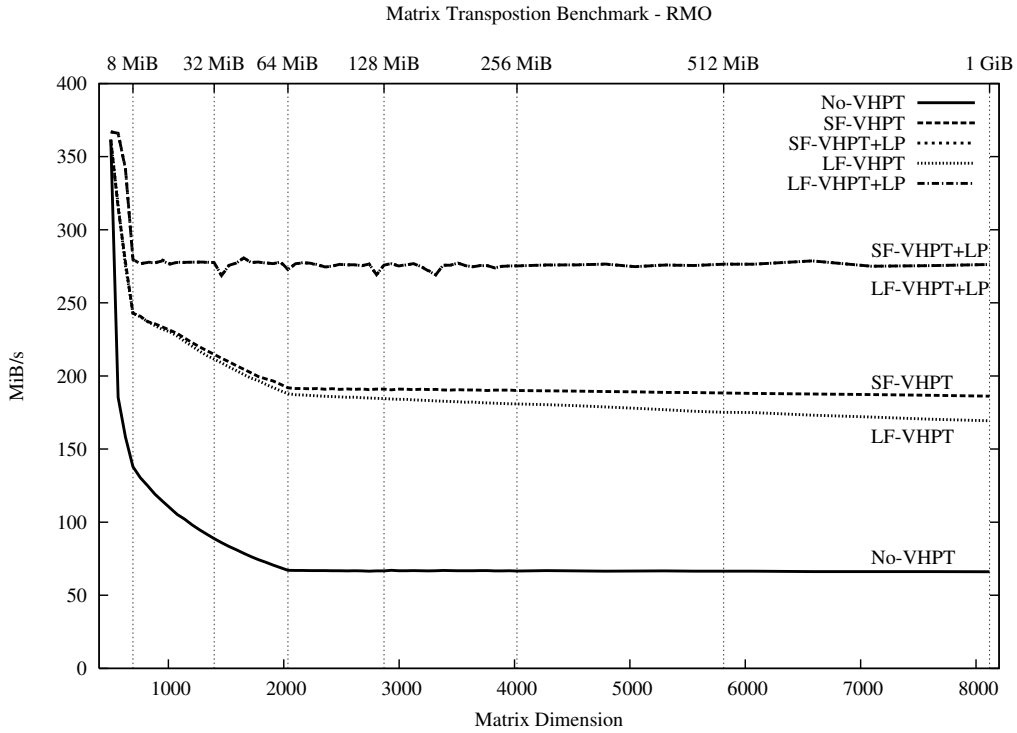
Analysis

Figure 6.3a presents results for increasing matrix size reading in a row-major order, while Figure 6.3b presents the same test when reading in column-major order. A benchmark run consists of gradually increasing the matrix dimensions; the upper axis shows the working-set size at various points.

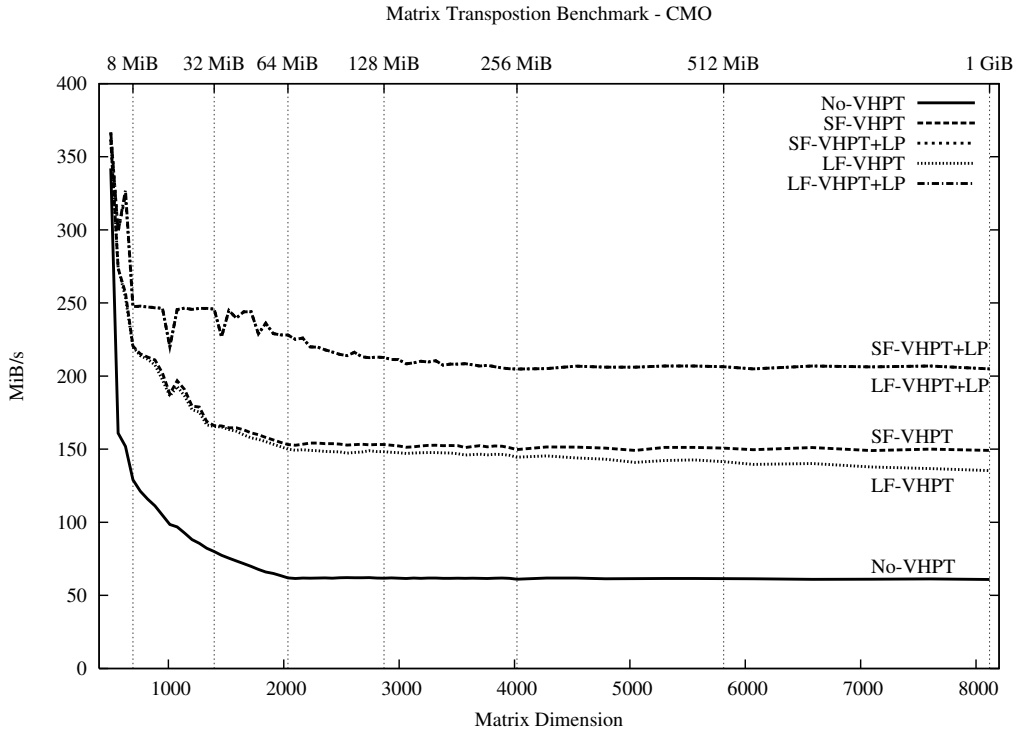
Overall, the graph shapes are as expected. Once the mapped area expands past the 6 MiB of cache on the test processor, performance drops dramatically. As the working set continues to become larger, contention for cache space and TLB entries increases (and performance decreases) until a “saturation point” is reached where performance levels out.

The lower performance of the fixed page-size LF-VHPT implementation compared to the fixed page-size SF-VHPT implementation can be attributed to higher cache latencies; profiling shows the less cache friendly LF-VHPT suffers, on average, almost two cycles greater latency when updating the cache in row major order (line 6 of Figure 6.2). This is to be expected, as HPW traffic is high during the benchmark and the longer LF-VHPT entries pollute the cache more.

Figure 6.3b shows the matrix transposition done in column-major order in order to deliberately perturb the cache. A side-effect of this is closing some of the gap between fixed page-size LF-VHPT and SF-VHPT tests, since the SF-VHPT gets less chance to benefit from its smaller cache footprint. The test also exhibits greater “jitter” before settling; this appears to be caused by particularly bad cache conflicts at particular points causing long cache latencies.



(a) Reading in row-major order



(b) Reading in column-major order

Figure 6.3: Matrix transposition benchmark. The upper axis marks the size in megabytes of the total area mapped (i.e. the beginning and end matrices are half this size each).

Kernel	DTLB hit rate (%)	HPW-satisfied TLB miss (%)
No HPW	78	0
SF-VHPT	72	99.73
LF-VHPT	72	99.93
SF-VHPT+LP	100	56.10
LF-VHPT+LP	100	19.71

Table 6.1: TLB hit rate for a run of the matrix micro-benchmark presented in Figure 6.3a.

Figures 6.3a and 6.3b show that when multiple page size support is enabled, both SF-VHPT and LF-VHPT have almost exactly the same performance characteristics. The kernel can very effectively map large pages for this benchmark as confirmed by the results presented in Table 6.1, which shows TLB misses essentially eliminated with large page support. Therefore, with no TLB misses the result is a reflection of the ability of the hardware to transfer data rather than the effectiveness of any particular translation scheme.

Although this table shows a large percentage difference in HPW efficiency between the LF-VHPT+LP and SF-VHPT+LP kernel, the actual results are negligible; over ≈11 billion data references, of the 3476 TLB misses for the SF-VHPT+LP kernel 1950 were resolved by the HPW, as compared to 679 of the 3445 misses reported by the LF-VHPT+LP kernel. Consequently the overall increase in bandwidth is attributed to the much reduced page fault rate and the consequent reduction in HPW traffic freeing the cache for data transfer.

6.2.2 Tlbcover

Tlbcover is a TLB intensive workload used for system testing at Hewlett Packard's Open Source and Linux Organisation (OSLO), kindly made available for this work by Lee Schermerhorn. The benchmark is derived from a customer code base and makes random transformations of an in-memory table. The results of repeated runs with increasing working sets ranging from 1 MiB to 512 MiB are illustrated in Figure 6.4.

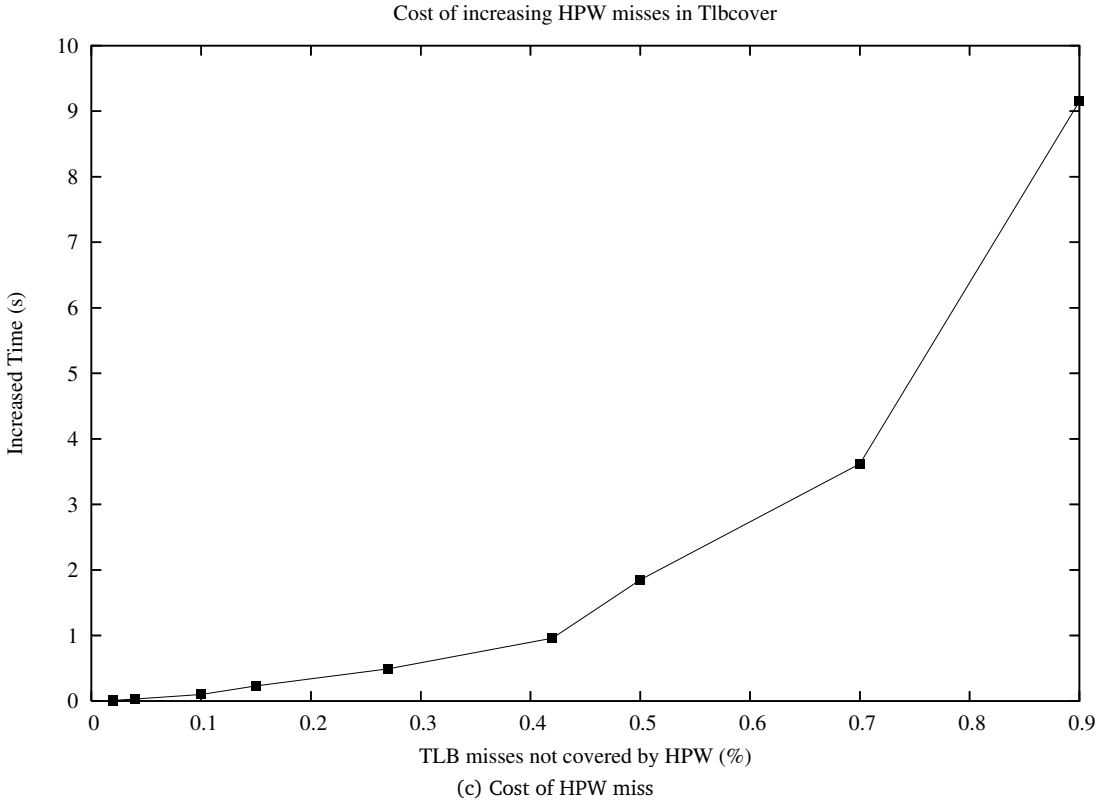
Profiling presented in Figure 6.4a shows increasing TLB pressure as the working set increases. As expected, large-page support kernels effectively remove TLB overheads by mapping the table with large pages. This results in good run time decreases; for the largest working set run time decreased from 114 to 104 seconds, or 8.8%. Despite causing significant TLB stress, the benchmark does not produce significant HPW pressure; results in Figure 6.4b show only a gradual decrease in TLB misses being covered by the SF-VHPT HPW as the highest working sets are reached. Combined with the timing information, Figure 6.4c illustrates increase in run time as the HPW becomes less effective. The non-linear graph illustrates that modest decreases in HPW effectiveness can result in greater than proportional runtime increases.

To analyse the cache effects of this HPW activity the average instructions retired per cache miss results are shown in Figure 6.5. In an attempt to analyse the HPW in the best light possible these results were obtained from a more modern *Montecito* processor with 16 KiB L1 cache, 256 KiB of shared L2 cache and 9 MiB of L3 cache. A higher result indicates more efficient use of the available cache. Figure 6.5a shows that once the L1 cache is overwhelmed an equilibrium is quickly sustained with all tested kernels. On the other hand, the L2 cache figures show a slight advantage in cache efficiency due to the large-page kernels as the higher working sets are reached. With large pages there are no interjections from HPW traffic and almost twice as many instructions are retired for each cache miss. This test serves to highlight the benefits of lower cache pollution from avoiding HPW traffic.

(a) Runtime and DTLB coverage

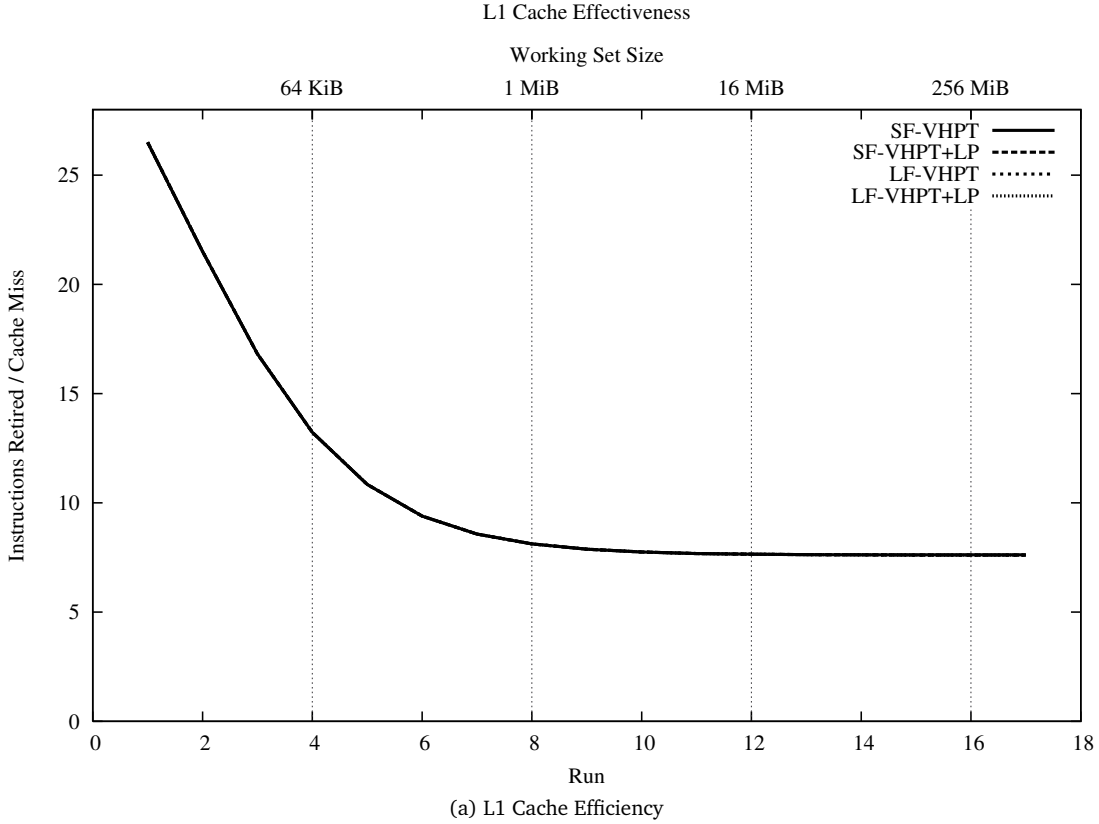
Run	1	2	3	4	5	6	7	8	9	10	11
SF-VHPT HPW %	99.98	99.98	99.98	99.96	99.90	99.85	99.73	99.58	99.50	99.30	99.10

(b) HPW Coverage (% of TLB misses covered by HPW)

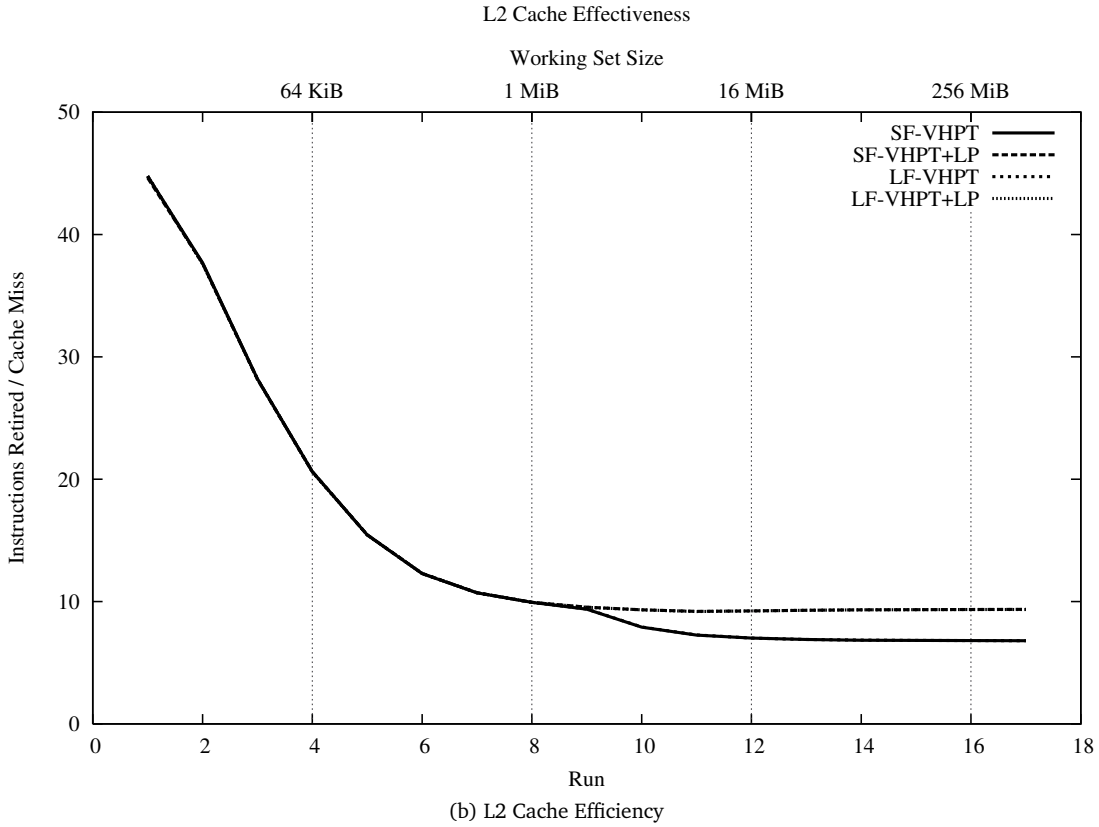


(c) Cost of HPW miss

Figure 6.4: Results of the Tlbcover benchmark



(a) L1 Cache Efficiency



(b) L2 Cache Efficiency

Figure 6.5: Cache efficiency for the Tlbcover benchmark

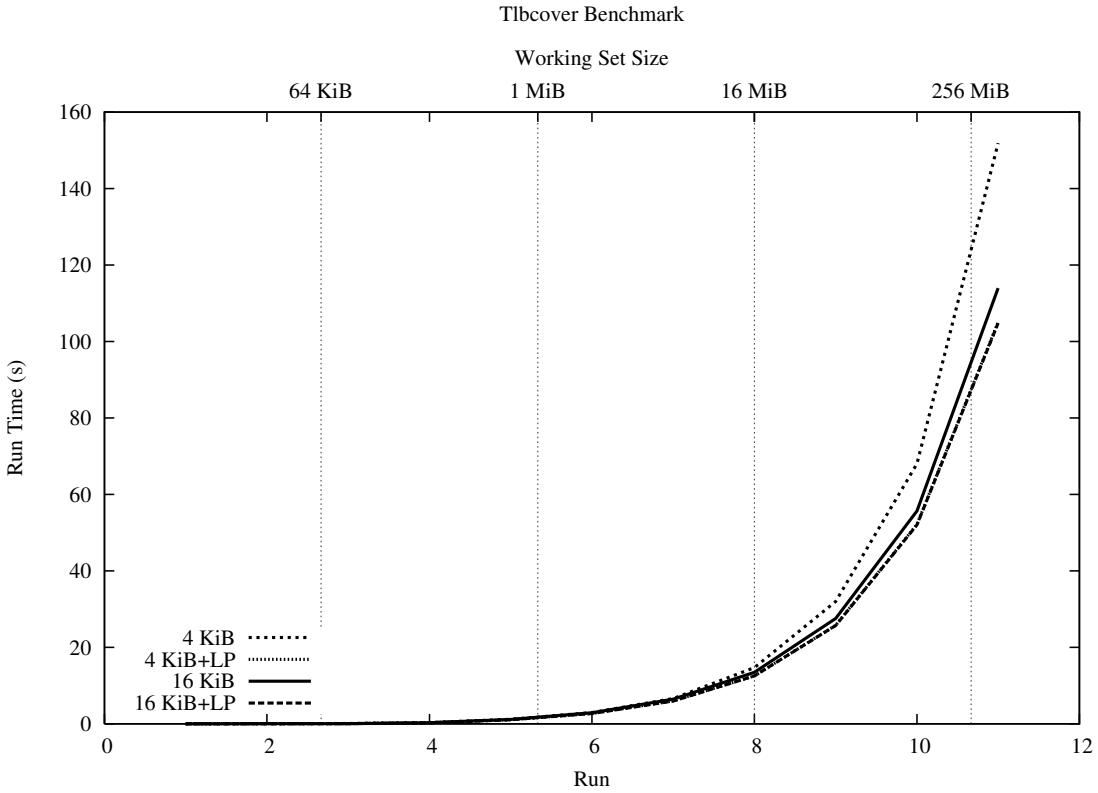


Figure 6.6: Results of Tlbcover with reduced 4 KiB page size. Note the 4 KiB+LP results are overlapped with the 16 KiB+LP results as the same large pages are chosen for both.

Effect of smaller page size

By default, Itanium Linux uses a 16 KiB fixed base page-size. This is large even by contemporary standards; in particular it is considerably larger than the 4 KiB base page-size provided by the ubiquitous IA-32 processor.

Figure 6.6 shows the results of the Tlbcover benchmark run on a SF-VHPT kernel with a reduced 4 KiB base page-size. The reduced page size leads to a considerable increase in runtime as the working set increases, however enabling large-page support results in performance exceeding the 16 KiB base page-size kernel and equal to the 16 KiB kernel with large-page support. The large-page kernel results overlapping is expected since the kernel has the ability to map the same larger pages independent of base page-size.

Table 6.2 shows the relative speed-up afforded by the large-page kernels. The results show that the 4 KiB kernel scales better than the 16 KiB kernel as the working set increases. The results also suggest that a considerable component of the benefits can be realised by statically increasing the base page size. However, Figure 6.7 shows that further increasing the base page-size to 64 KiB brings no further benefits; the additional improvements are from the multi-megabyte pages allowed by the large-page kernels.

6.3 SPEC

The relative performance of a multiple page-size kernel for SF-VHPT and LF-VHPT is shown in Figure 6.8. Performance results are not uniformly better and, in some cases, show a performance decrease. Over the

	Speed-up	
Run	4 KiB+LP	16 KiB+LP
1	1.00	1.00
2	1.14	1.17
3	1.38	1.15
4	1.18	1.12
5	1.12	1.09
6	1.10	1.09
7	1.11	1.08
8	1.17	1.08
9	1.24	1.07
10	1.30	1.07
11	1.45	1.09

Table 6.2: Speed-up relative to base page-size kernels for Tlbcover benchmark

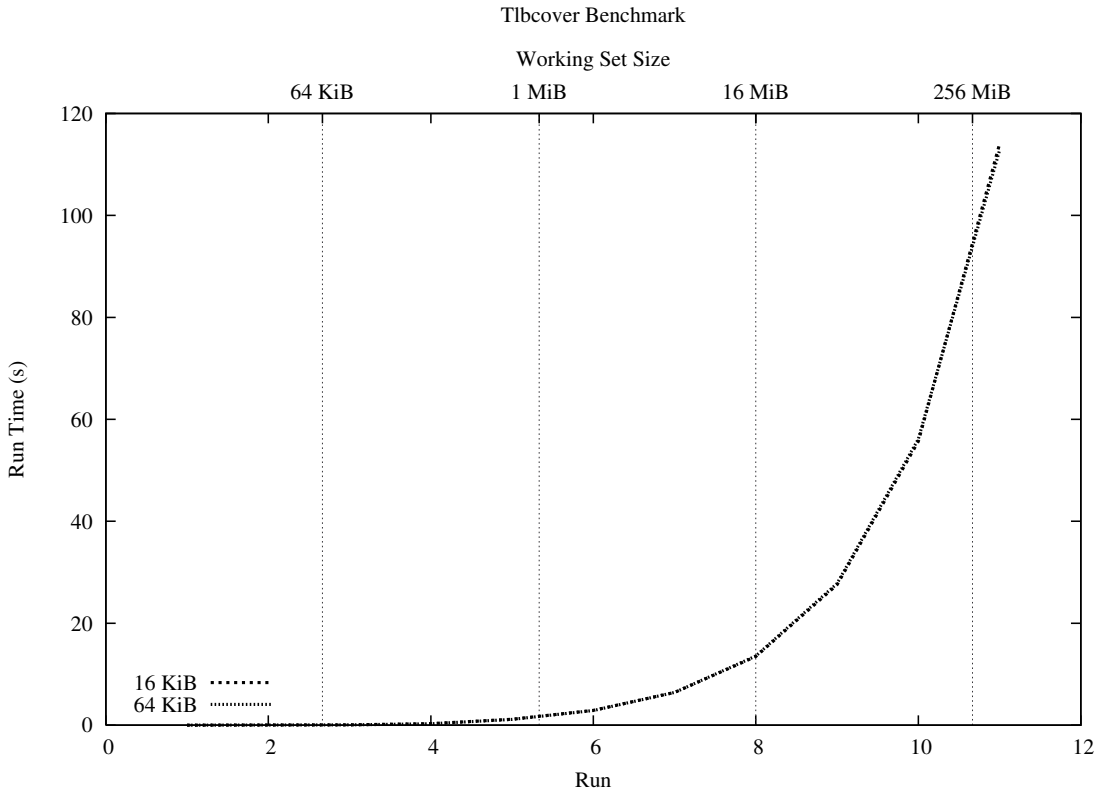
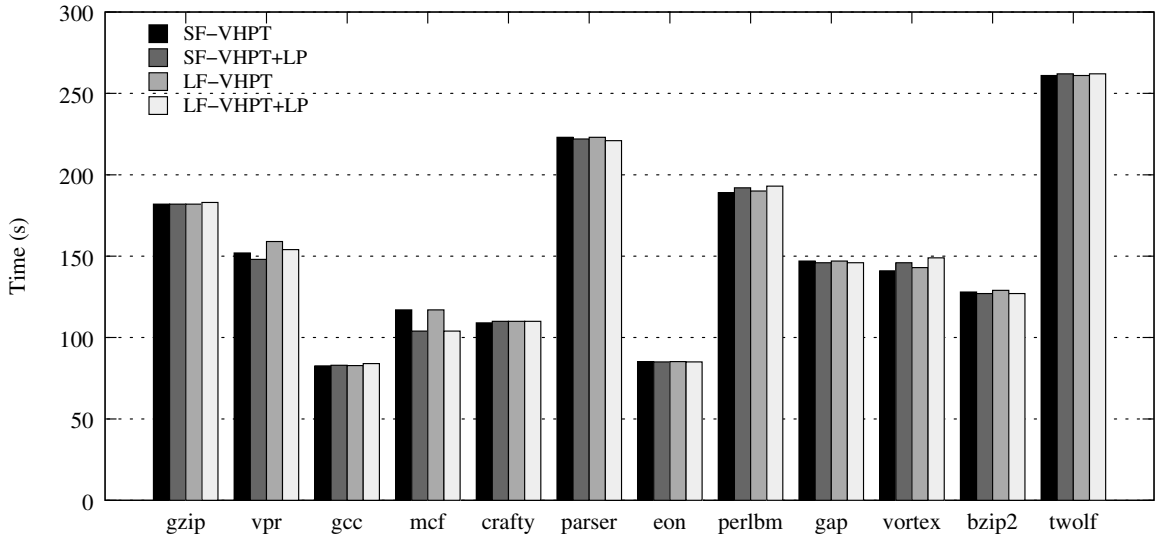


Figure 6.7: Results of Tlbcover with a 64 KiB page size.

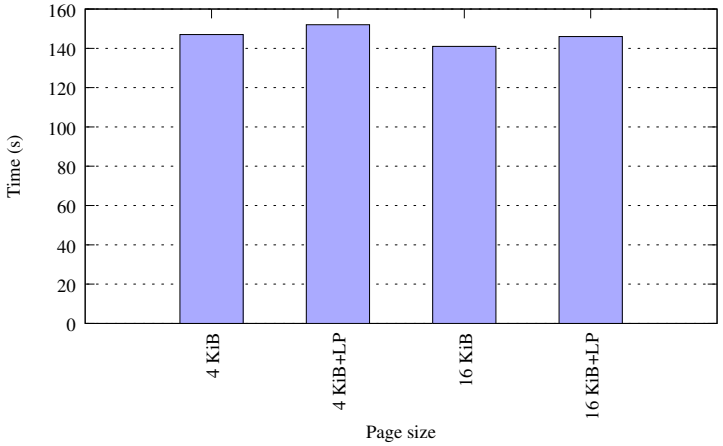


(a) Time

	Speed-up	
Benchmark	SF-VHPT+LP	LF-VHPT+LP
gzip	1.000	0.995
vpr	1.027	1.032
gcc	0.995	0.986
mcf	1.125	1.125
crafty	0.991	1.000
parser	1.005	1.009
eon	1.002	1.002
perl	0.984	0.984
gap	1.007	1.007
vortex	0.966	0.960
bzip2	1.008	1.016
twolf	0.996	0.996

(b) Relative Speed-up

Figure 6.8: SPEC CPU2000 results with large-page support. Speed-up is relative to the same kernel without large-page support.



Page Size	Speed-up
4 KiB+LP	0.967
16 KiB	1.043
16 KiB+LP	1.007

Figure 6.9: Run times for `vortex` with various base page-sizes and large page support. Speed-up is relative to the smallest 4 KiB base page size.

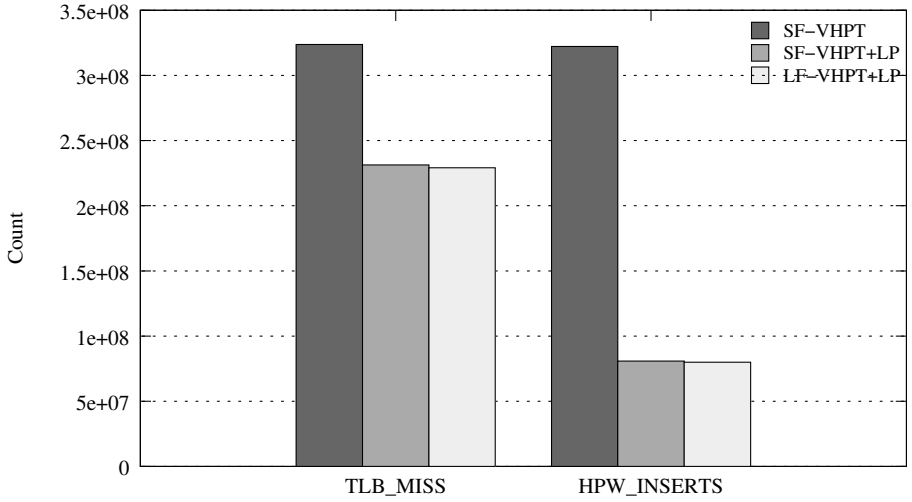


Figure 6.10: TLB misses and HPW inserts for for `vortex` with various base page-sizes and large-page support.

runs there were no demotions of large pages for want of contiguous memory, hence effects of fragmentation are not being felt.

6.3.1 `vortex`

The `vortex` benchmark shows the greatest slowdown when run with large-page support. To initially ascertain the TLB requirements of the benchmark the page size was reduced to 4 KiB as in Figure 6.9. Somewhat as expected the relatively small increase in running time with smaller pages indicates the test is not creating significant TLB pressure. This is confirmed when looking at statistics gained from hardware profiling in Table 6.5 where we can see 99.5% of data references were covered by the TLB. This benchmark therefore is useful to help quantify some of the overheads related to large-page support.

Hardware profiling of TLB operations is presented in Figure 6.10. The left-hand side shows a marked reduction in the number of TLB misses for kernels with large page support enabled, consistent with increased TLB coverage. However, the right-hand side shows a drastic decrease in the proportion of

Kernel	Gradient	Cycles @ 1.5 GHz
SF-VHPT	1.28181×10^{-8}	19.23
SF-VHPT+LP	4.09793×10^{-8}	61.47
LF-VHPT+LP	5.35105×10^{-8}	80.27

Table 6.3: Average cost of a TLB miss for large-page kernels (see Figure 6.11).

Kernel	% misses covered by HPW	Overall miss cost	Breakdown	
			HPW	Software
SF-VHPT	99.53	19	19	<1
SF-VHPT+LP	34.94	61	21	40
LF-VHPT+LP	34.88	80	28	52

Table 6.4: Breakdown of HPW and software fault costs (cycles).

translations inserted by the hardware page-table walker. It therefore appears the slower run time of the benchmark can be attributed to increased TLB refill costs due to the reduced effectiveness of the hardware page-table walker.

6.3.2 TLB Refill Costs

To attempt to quantify the TLB refill costs the benchmark was repeated whilst artificially reducing the number of TLB entries available to it by pinning "fake" entries. This causes an increase in TLB misses and provides data suitable for creating a linear regression. Results are presented in Figure 6.11, where each point represents a benchmark run with one fewer TLB entries available. Taking the gradient of the best-fit line gives an approximation of the extra time taken for each TLB miss, and multiplying by clock speed of the processor estimates the average cycles required for each TLB miss (Table 6.3).

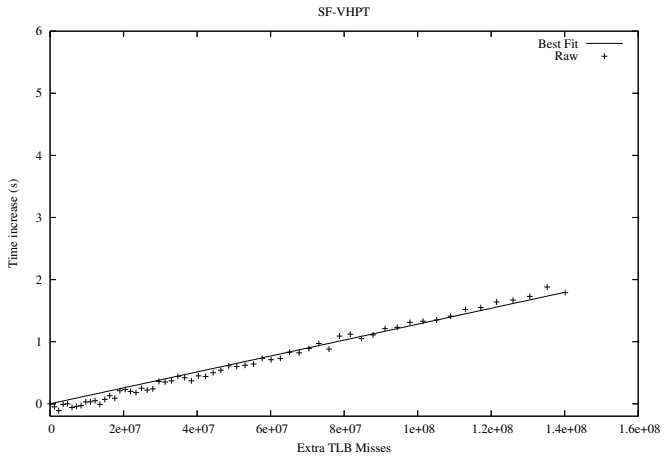
We can combine these results with the results from Figure 6.10 to gain an approximate breakdown of time spent in the HPW and time spent in software, presented in Table 6.4. These results give us some idea of the relative costs summarised in Section 5.4.1.

Due to the extremely high coverage of the SF-VHPT HPW the software fault component is negligible and the result therefore essentially a measure of HPW overhead. As we would expect the SF-VHPT with large page support suggests a very similar overhead; the inability to take advantage of hot cache lines full of translations may lead to the small extra overhead. We see that the LF-VHPT with large-page support runs slower again due to the longer variable cache latencies as described in Section 4.3.2.

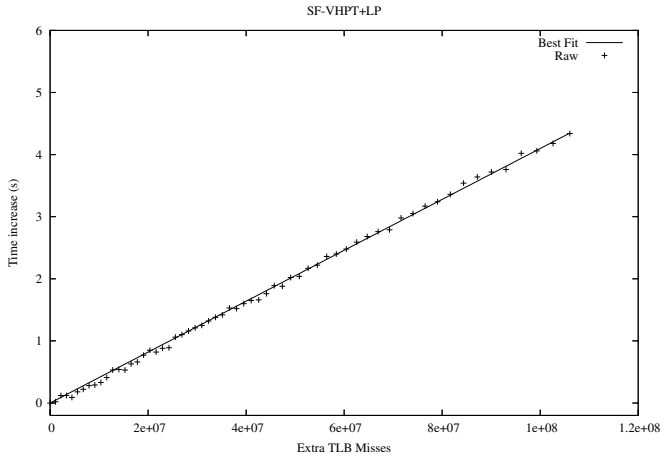
6.3.3 TLB Effectiveness

Given the fault timings derived in Table 6.3, the fraction of cycles which are not used for translation management operations (i.e. are retiring useful computational instructions) can be derived. We will refer to this ratio as "effectiveness"; results given the previously derived TLB miss costs are illustrated in Figure 6.12. The graph asymptotically approaches 100% effectiveness with the steepest gradients below around 1000 cycles, although it takes up to ≈5000 cycles for all three cases to be within 1% of each other.

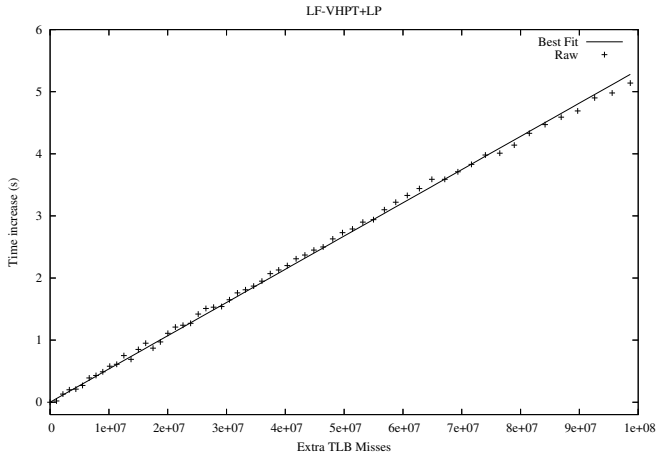
Relative results for `vortex` running on SF-VHPT with and without large-page support are presented in Figure 6.13. Here we see that the reduction in TLB misses with large pages does lead to greater average time between TLB misses. However, the movement is not sufficient to maintain the same effectiveness as



(a) $m = 1.28181 \times 10^{-8}$



(b) $m = 4.09793 \times 10^{-8}$



(c) $m = 5.35105 \times 10^{-8}$

Figure 6.11: TLB Reduction test. Each point represents a run of the benchmark with one fewer TLB slot available. The results were normalised to an unmodified system and a line of best-fit added. The gradient of the line (presented) represents the average cost of a TLB fault.

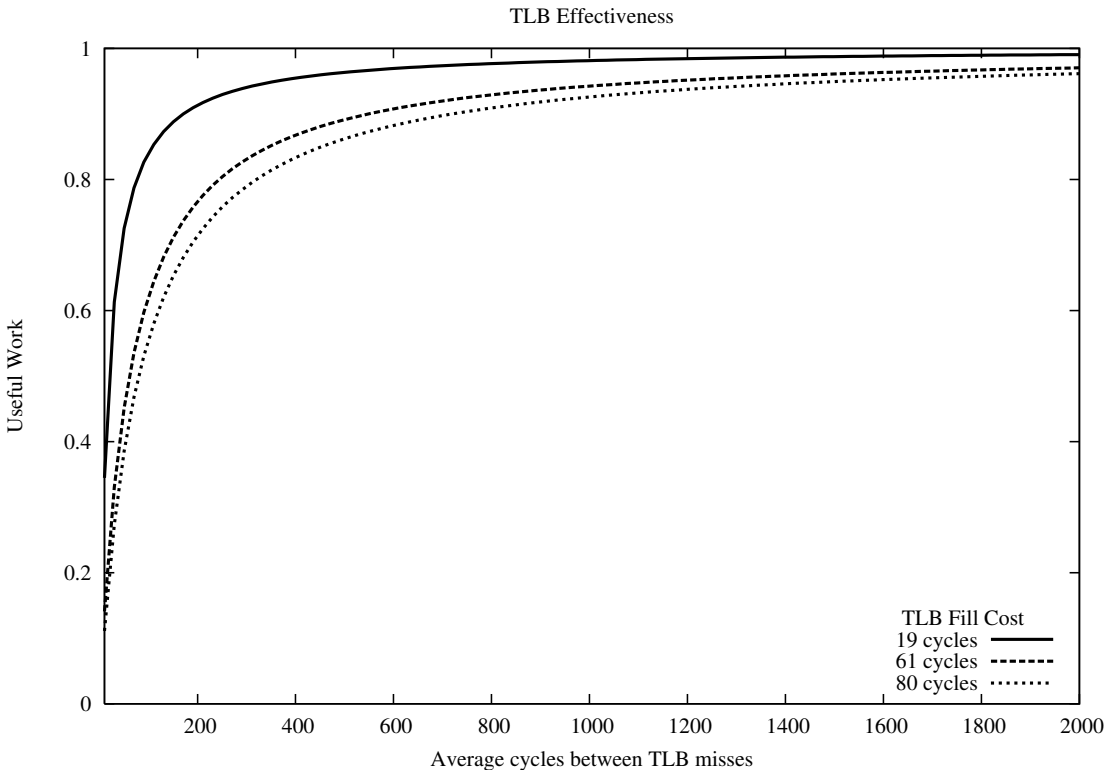


Figure 6.12: TLB effectiveness given various fault latencies (as per Table 6.3)

the plain SF-VHPT with its lower TLB refill time. Therefore, for the SF-VHPT+LP line to reach the same effectiveness (0.97) it would need to increase average cycles between misses to 1972 cycles. Working the results into the parameters of the effectiveness equation (Equation 6.1, below), gaining a sufficient increase would require reducing the number of TLB misses from Figure 6.13 to that derived in Equation 6.2, a cut of a further 46%.

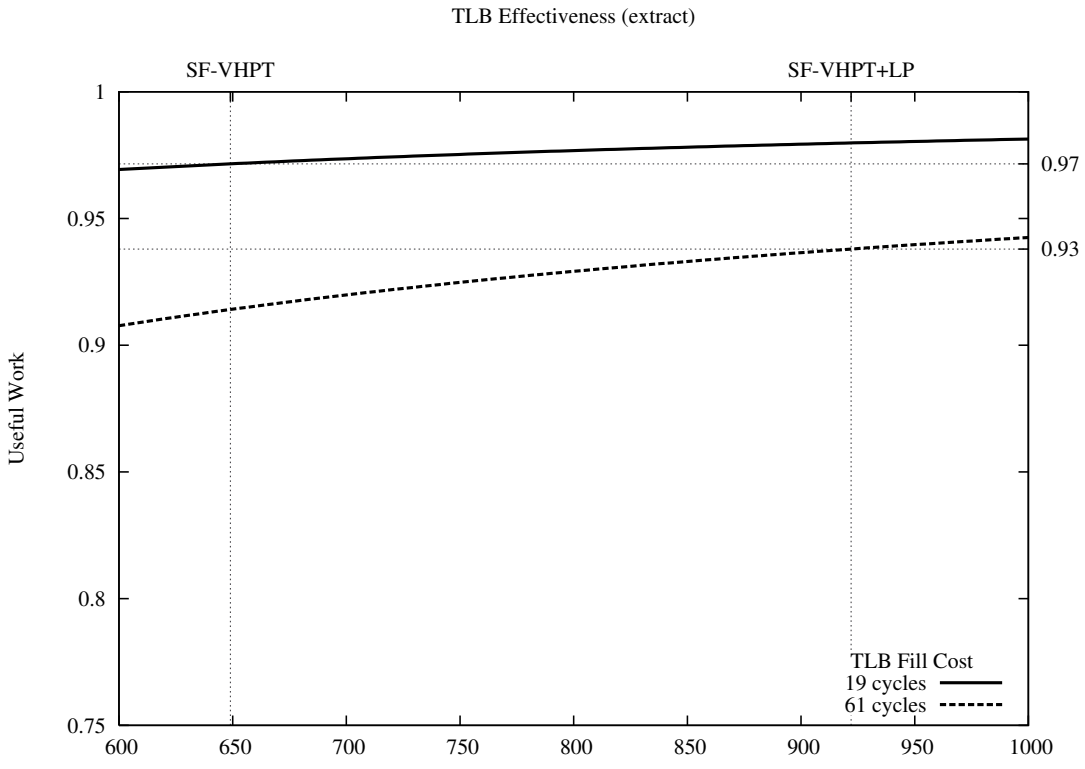
$$effectiveness = \frac{\frac{cycles}{misses}}{\frac{cycles}{misses} + latency} \tag{6.1}$$

$$\begin{aligned} 0.97 &= \frac{\frac{213,254,629,975}{misses}}{\frac{213,254,629,975}{misses} + 61.47} \\ 213,254,629,975 &= 0.97 \times (213,254,629,975 + 61.47 \times misses) \\ misses &\approx 107,296,307 \end{aligned} \tag{6.2}$$

6.3.4 HPW Effectiveness

The results presented in Section 6.3.3 represent the combined behaviour of both the OS TLB refill handlers and the HPW. If the HPW covers more refills the lower overheads will raise the efficiency ratio.

Previous discussion has identified the memory access patterns of an application as important to the effectiveness of the HPW. For example, large strides can cause more nested faults with a virtual linear table and little re-use of entries creates more overhead in managing the LF-VHPT hash table.



Kernel	Benchmark Cycles	TLB Misses	Cycles/Miss
SF-VHPT	210,360,675,290	323,718,623	650
SF-VHPT+LP	213,254,629,975	231,246,443	922

Figure 6.13: TLB effectiveness extract for the `vortex` benchmark (see also Figure 6.12).

Kernel	DTLB hit rate (%)	HPW-satisfied TLB miss (%)
SF-VHPT	99.65	99.53
SF-VHPT+LP	99.75	34.95
LF-VHPT+LP	99.75	34.88

Table 6.5: TLB miss rates for `vortex`. Column 2 shows data references covered by the TLB, column 3 is TLB misses covered by the HPW.

To gain a better idea of how efficient we can expect the HPW to be with this benchmark, we instrumented the kernel to report the virtual address of faults during a benchmark run. The results for `vortex` are illustrated in Figure 6.14a.

The first thing to notice is the small range of the y-axis, which covers ≈25 MiB of address space. This small area is very favourable for the SF-VHPT, since with a 16 KiB page size a single TLB entry can hold enough translation entries to map 32 MiB of virtual address space via the virtual linear page-table.

This measurement is instrumented within the kernel at the point of allocation of physical pages (`do_anonymous_page`) thus repeated accesses to the same virtual address represents a new mapping and implies a `free()` operation in between. We can therefore see that the test creates, accesses and frees many small allocations across its lifespan.

The SF-VHPT with large-page support shows some benefit from the tight address space; the very slightly higher HPW fill rates in comparison to LF-VHPT illustrated in Table 6.5 can be attributed to greater ability to find adjacent mappings.

However, the latency of misses remains three times greater than the original test. We know that faults covered by a non-base page-size cannot be satisfied by the SF-VHPT HPW, hence the reduced HPW coverage in Table 6.5 is to be expected.

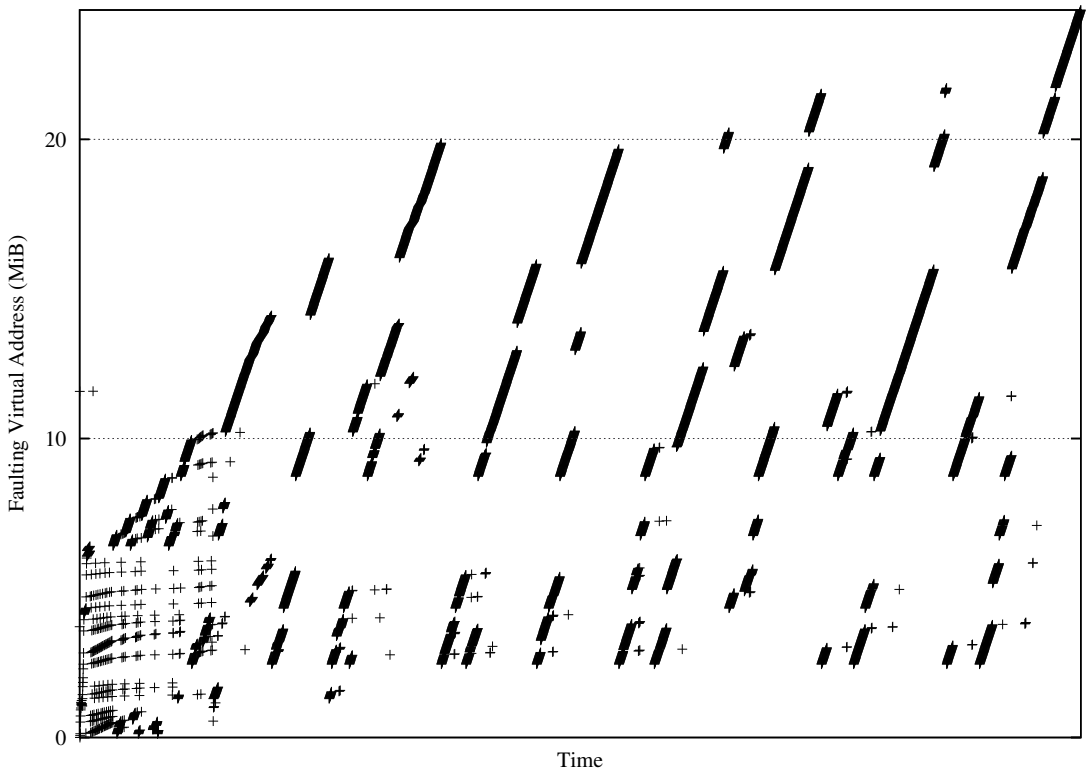
Table 6.6 shows the large page allocations for a single benchmark run as a proportion of all pages allocated. There is a particularly large proportion of order-2 allocations (with 16 KiB pages this works out to a 64 KiB page) with over 80% of the address-space used by the process being mapped with a non-base page-size. With most of the allocations being relatively small and having a short life span, the extra costs of transitioning to a software fault handler are not amortised across the benchmark run, leading to the negative performance impact.

This behaviour is also unfriendly to LF-VHPT. Firstly, the test shows an extra 20 cycles of latency over the SF-VHPT. This is due to the extra page table walks done by the LF-VHPT fault handler; unlike the SF-VHPT which can use the virtual linear page-table mapping to access the translation entries the LF-VHPT handler must traverse the page table to the translation entries on every miss.

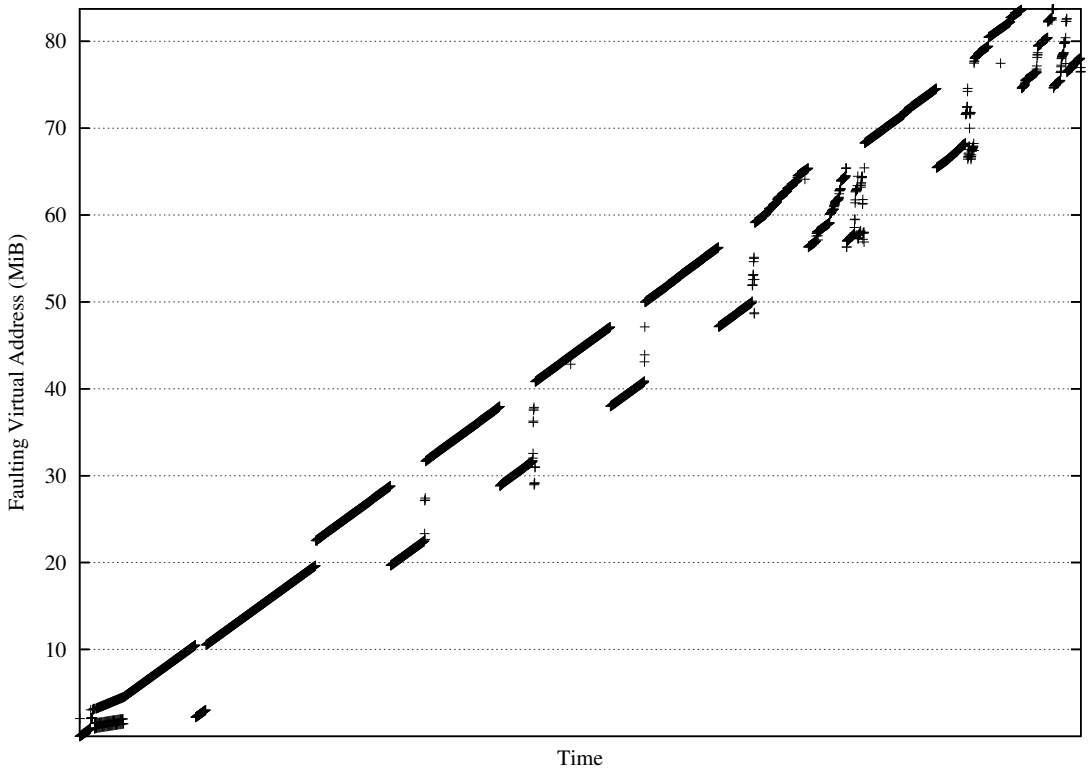
The heavy allocate-use-free cycle is also unfavourable since freeing a mapping invalidates the hash-table entry meaning it is both not reused by the HPW and needs to be frequently re-instantiated. This shows up in Table 6.5 where we see the smaller number of misses the HPW is able to satisfy.

6.3.5 Comparison with `mcf`

It is interesting to compare the negatively impacted `vortex` test with the positively impacted `mcf` benchmark.



(a) Access pattern for `vortex`. This benchmark runs three separate processes each with similar access patterns, only the first process is shown above.



(b) Access pattern for `mcf`

Figure 6.14: Faults recorded in `do_anonymous_page` whilst running two SPEC benchmarks with the HPW disabled. Addresses have been normalised to the lowest virtual address accessed

Order (2^n)	Allocated	%
0	4872	17.53%
2	3374	48.56%
4	545	31.38%
6	11	2.53%

Table 6.6: Large pages allocated (in `do_anonymous_page`) for a run of `vortex` (order-0 is the base page-size of 16 KiB). Column 2 represents the percentage of total address space allocated with pages of order-*n* size.

Kernel	Benchmark Cycles	TLB Misses	Cycles/Miss
SF-VHPT	167,011,979,645	767,849,589	218
SF-VHPT+LP	154,402,232,842	402,959	383,171

Table 6.7: Cycles per TLB miss for `mcf` benchmark

Table 6.7 is analogous to the `vortex` results presented in Figure 6.13. We see that with this benchmark large-page support has been extremely effective at reducing TLB misses, and thus longer refill time falls to insignificance given the improvements.

The access profile, illustrated in Figure 6.14b, gives more insight into why large-page support is so effective. A larger address space range of 85 MiB is covered in a very linear fashion which is ideal for large pages. There is very little repeated access meaning much less cyclic allocate-free behaviour.

Table 6.8 reinforces that the kernel effective at creating large pages and essentially eliminates TLB faults. Table 6.9 shows that the benchmark creates many high-order large pages which further helps to spread the cost of higher latency fault handlers.

6.3.6 Comparison

Several other large-page implementations have used the SPEC CPU suite to analyse their implementation.

Navarro The most direct comparison to prior work can probably be made with Navarro [Nav04] who implemented variable large-page support for Itanium on FreeBSD. Table 6.10 shows a comparison of relative speed-ups for the SPEC suite.

There are a number of points to note about the comparison:

- FreeBSD uses the LF-VHPT HPW by default, although the original work did not document any measurements of interactions with the HPW.
- The Itanium architecture is by its nature very sensitive to compiler techniques. Navarro implies use of the `gcc` compiler which is known for sub-optimal code, especially in older versions. Our benchmarks

Kernel	DTLB hit rate (%)	HPW-satisfied TLB miss (%)
Vanilla	97.21	93.42
SF-VHPT+LP	100.00	0.22
LF-VHPT+LP	100.00	0.09

Table 6.8: TLB miss rates for `mcf`. Column 1 is data references covered by the TLB, column 2 is TLB misses covered by the HPW (100% represents $< 0.001\%$ of data references causing a TLB miss).

Order (2^n)	Allocated	%
0	5497	20.20
2	1028	15.11
4	96	5.65
6	39	9.17
8	17	15.99
10	9	33.87

Table 6.9: Large pages allocated (in `do_anonymous_page`) for a run of `mcf` (order-0 is the base page-size). Column 2 represents the percentage of total address-space used allocated within pages of order-*n* size.

	Speed-up		
Benchmark	Navarro	LF-VHPT+LP	Difference
gzip	1.000	0.995	-0.005
vpr	1.300	1.032	-0.268
gcc	1.012	0.986	-0.026
mcf	1.707	1.125	-0.582
crafty	1.083	1.000	-0.083
parser	1.081	1.009	-0.072
eon	1.000	1.002	+0.002
perl	1.031	0.984	-0.047
gap	1.068	1.007	-0.061
vortex	1.125	0.960	-0.165
bzip2	1.082	1.016	-0.066
twolf	1.137	0.996	-0.141

Table 6.10: Comparison of SPEC2000 benchmark results with Navarro [Nav04] (Page 65). Speed-up is LF-VHPT+LP compared to LF-VHPT.

were built with the current Intel `icc` compiler. This may have some influence on memory access patterns, especially since `icc` does more aggressive pre-fetching of data.

- The test was performed on older Itanium 1 hardware which has significantly smaller caches and a smaller 64 entry L2TLB.

One particularly large difference is the `mcf` test which is improved by almost an additional 60% for Navarro compared to our work. As identified in Section 6.3.5, this test is extremely friendly to large pages and it is possible to essentially eliminate TLB misses for the test.

Table 6.8 shows how our implementation has expanded TLB coverage to 100% and it is fair to assume Navarro's work has achieved this as well (although, as mentioned detailed TLB performance counter results are not presented). Therefore, this suggests Navarro's larger speed-up reported is in the most part due to the poorer performance of the code used for the base comparison.

Visual inspection of the FreeBSD fault handling code would seem to confirm this suspicion. For example, there is very little use of instruction level parallelism or the large register-space available: every instruction of the DTLB fault handler uses its own bundle.

Similar evidence for this hypothesis can be found in the results of the `C4` [Tro] which simulates the well known game *Connect Four*. The essential data structure is a 64 MiB hash table which is traversed by a number of CPU intensive algorithms. Table 6.11a shows that despite a large number of TLB misses, in comparison to the overall data references the misses are small. Further the SF-VHPT HPW is very effective here, although the SF-VHPT+LP essentially eliminates TLB misses.

However, as shown in Table 6.11b, despite a greater reduction in TLB misses performance increase is not

Kernel	SF-VHPT	SFVHPT+LP
TLB Misses	636,254,917	240
as % DR	0.15	0
HPW Inserts	635,071,787	188
as % TLB Misses	99.81	78.33

(a) Hardware counter statistics

Kernel	TLB Miss Reduction (%)	Speedup
Navarro	95.65	1.360
SF-VHPT+LP	100	1.015

(b) Comparison to Navarro

Table 6.11: Detailed results of the Fhourstones benchmark

Benchmark	Speed-up
gzip	1.123
vpr	1.167
gcc	1.092
mcf	1.094
crafty	1.152
parser	1.163
eon	1.120
perl	N/A
gap	1.059
vortex	1.222
bzip2	1.143
twolf	1.124

Table 6.12: SPEC CPU2000 results from Winwood et al. [WSF02] (Table 2) presented for comparison.

commensurate with Navarro's work. The speedup seen in our work is almost exactly as we expect from our results; for 635 million misses at 19 cycles on a CPU running at 1.5GHz works out to ≈6 seconds, which is 1% of the 10 minute run time.

Unlike the more widely deployed Linux, the Itanium port of FreeBSD has not received close scrutiny from an army of experienced developers looking to optimise performance-critical code paths. Therefore avoiding software faults has a relatively greater performance impact on Navarro's work. Removing the out-lier of `mcf` we calculate Navarro averaged a speed-up of 1.08 whilst our work remains essentially constant at 0.999.

Shimizu and Takatori Unfortunately exact overall results of Shimizu and Takatori [ST03] are difficult to ascertain due to the resolution of the graphs in the paper. Visual inspection shows considerable variation and illustrates enabling larger page sizes is not always a performance improvement, corresponding to our results.

Winwood et al. Winwood et al. [WSF02] (Section 2.4.1) cite extremely impressive results with an average 15% speed-up; full results are reproduced in Table 6.12. We note, however, that their benchmarking modified `sbrk` to return much larger areas of memory and consequently the `malloc` implementation does not fall back to the presumably slower `mmap`. Their work was also implemented on IA-32 which only supports a large 4 MiB large page, thus presumably there was considerable resource under-utilisation when mappings did not fully use their allocation.

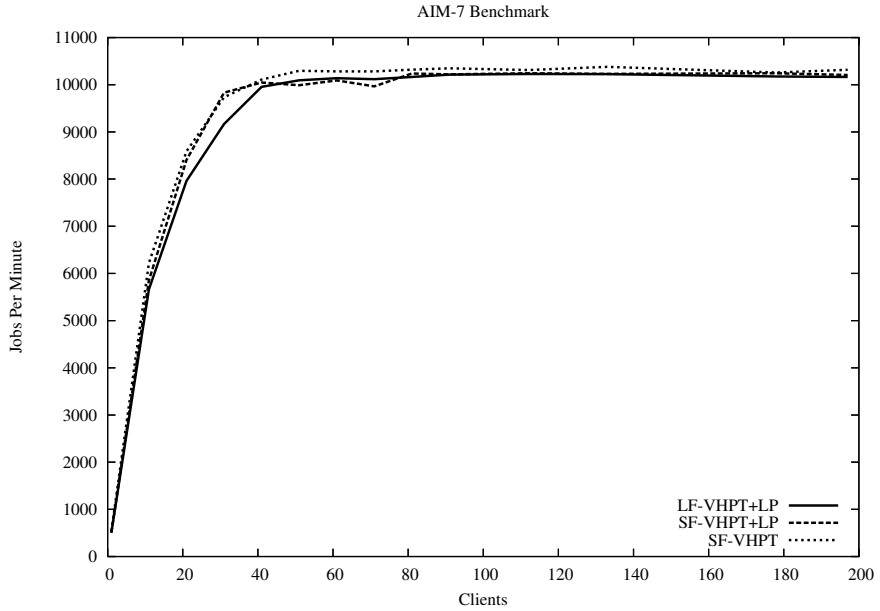


Figure 6.15: Results of the OSDL AIM-7 benchmark from 1 to 200 clients with large-page support.

HPW	Extract	Build
None	18.41	708.89
SF-VHPT	15.55	693.33
SF-VHPT+LP	15.04	694.09
LF-VHPT	15.60	696.43
LF-VHPT+LP	15.05	694.10

Table 6.13: Time (in seconds) for a kernel build benchmark with large-page support. See also Table 4.7.

6.4 System Performance

6.4.1 AIM

The AIM benchmark does not appear to rely heavily on TLB performance; Section 4.6.1 showed little difference between cases with the HPW enabled and disabled. Results from benchmark runs with large-page support are shown in Figure 6.15.

The test only generates modest amounts of large pages. The largest page allocated order-6 (1 MiB) and ≈10% of the total pages allocated for a benchmark run are order-2 (64 KiB) pages.

6.4.2 Kernel Compile

Results of a kernel compile benchmark (previously discussed in Section 4.6.2) are presented in Table 6.13. Results show large-page support makes an improvement in the CPU and memory intensive decompression phase but and for the LF-VHPT HPW recovers the costs incurred by extra management of the hash table.

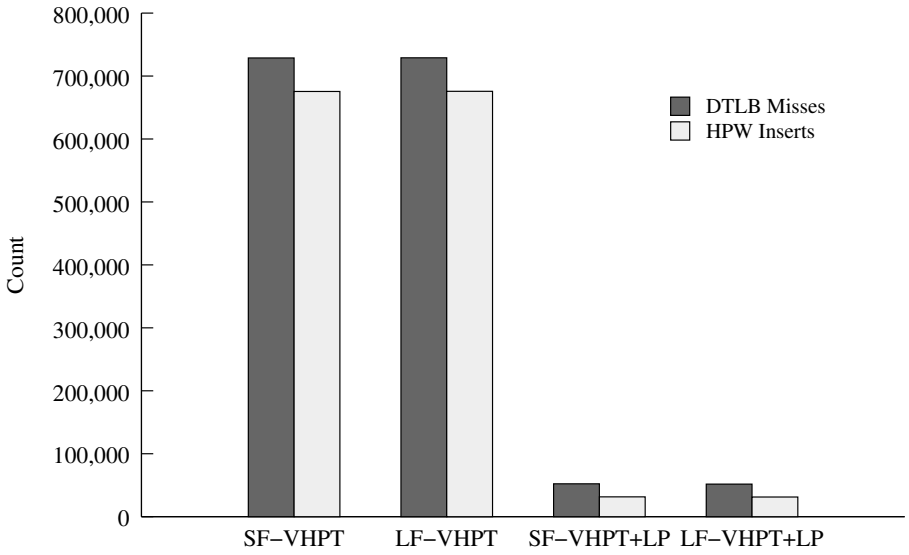


Figure 6.16: TLB statistics for `gcc` building a large kernel file.

	SF-VHPT	SF-VHPT+LP
TLB Misses	5,711	2,127
Covered by HPW	5,554	378
Software (@ 61 cycles)	9,577	106,689
Hardware (@ 18 cycles)	99,972	31,482
Total	100,929	138,171

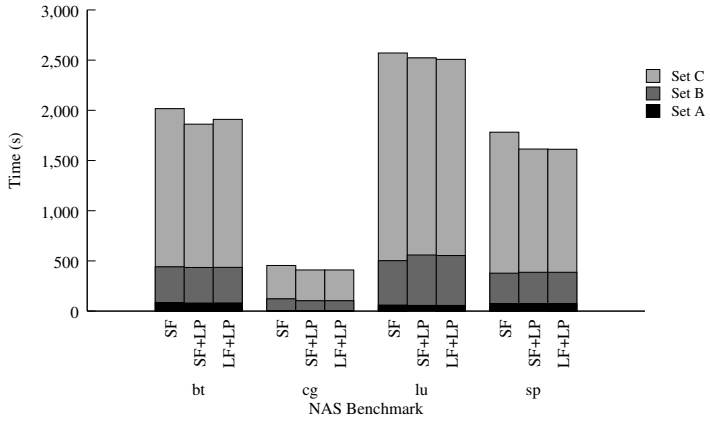
Table 6.14: Cycles spent handling TLB refill for a trivial compilation.

Breaking the results down, large-page support drastically reduces the number of TLB misses incurred by each compiler process. Figure 6.16 shows the TLB performance for each HPW combination when compiling a single large and complex file. However, the nature of a compiler is such that it will have to process quite different amounts of work depending on the size and complexity of the input. To illustrate this point details for the compilation of a trivial "hello, world" program are presented in Table 6.14.

We use the miss timings derived in Section 6.3.2, generously estimating the SF-VHPT software cost as the same as that for SF-VHPT with large pages. Despite the large reduction in TLB misses the overall estimated cycles spent handling TLB reload increases. At 1.5GHz this small difference is imperceivable, however as seen in the slight increase in runtime for the SF-VHPT+LP kernel, over the more than 1000 varied `gcc` iterations of the kernel compile benchmark small extra overheads can accumulate.

6.4.3 NAS

The NAS benchmark suite represents the workload of a typical scientific application (full details in Section 4.6.4). Results with large-page support are presented in Figure 6.17. The results show some variation but in general show a considerable performance improvement. The TLB efficiency results shown in Table 6.15 (derived as per Figure 6.13) show that large-page support is very effective in increasing the cycles between TLB misses significantly enough to achieve performance improvements. Table 6.16 shows the large-page allocations for the benchmark run. For the tests with the largest dataset we see a significant number of very large (64 MiB and 256 MiB) pages being allocated; each order-14 page can save up to 16,384 faults.



Benchmark	Speed-up SF+LP	LF+LP
bt.A	1.069	1.053
bt.B	1.004	1.004
bt.C	1.103	1.068
cg.A	1.193	1.193
cg.B	1.176	1.177
cg.C	1.083	1.083
lu.A	1.047	1.048
lu.B	0.883	0.892
lu.C	1.053	1.058
sp.A	1.007	1.008
sp.B	0.969	0.970
sp.C	1.144	1.146

Figure 6.17: NAS Benchmark results for kernels with large-page support. Speed-up is over a standard SF-VHPT kernel.

Benchmark	TLB Efficiency SF	SF+LP	LF+LP
bt.A	0.981	1.000	0.981
bt.B	0.977	0.943	0.927
bt.C	0.975	1.000	0.957
cg.A	0.998	1.000	1.000
cg.B	0.998	1.000	1.000
cg.C	0.996	1.000	1.000
lu.A	0.981	1.000	1.000
lu.B	0.980	1.000	1.000
lu.C	0.965	1.000	1.000
sp.A	0.983	1.000	1.000
sp.B	0.962	1.000	1.000
sp.C	0.934	1.000	1.000

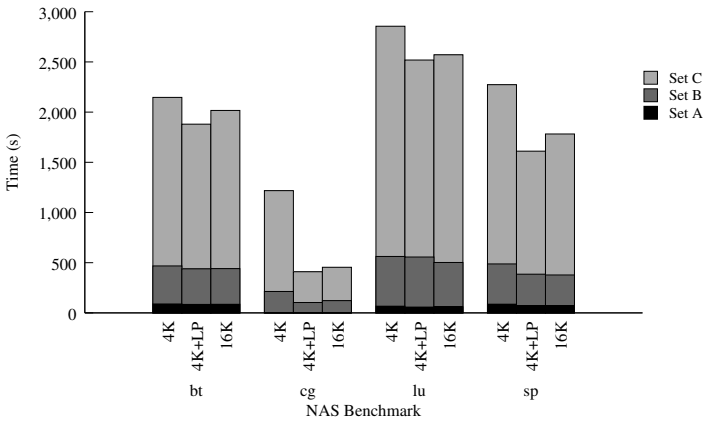
Table 6.15: TLB Efficiency for the NAS benchmark suite, using results from Table 6.3

Benchmark	Page Order (2^n) 2 (64 KiB)	4 (256 KiB)	6 (1 MiB)	8 (4 MiB)	10 (16 MiB)	12 (64 MiB)	14 (256 MiB)
bt.A	40	4	6	5	1		
bt.B	67	25	26	29	12	1	
bt.C	294	125	116	120	98	56	10
cg.A	47	9	10	10	1		
cg.B	71	28	29	34	18	2	
cg.C	300	129	121	125	102	62	10
lu.A	56	16	18	17	6		
lu.B	84	36	35	41	25	7	
lu.C	314	138	129	131	108	74	12
sp.A	63	20	22	23	7		
sp.B	91	41	39	45	31	8	
sp.C	328	143	134	134	112	80	13

Table 6.16: Large page allocations for NAS runs on LF-VHPT. Base page-size is 16 KiB.

Metric	SF-VHPT	SF-VHPT+LP
L1D_DCURECIR	225,388,161,352	242,224,154,417
L1D_L2BPRESS	332,504,678,771	439,962,001,684
L1D_TLB	3,119,659	18,863
L1D_HPW	46,051,582,991	16,563

Table 6.17: Performance counter results for the `lu.B` test (see Figure 4.4)



Benchmark	Speed-up SF-4K	Speed-up SF+LP-4K
bt.A	0.954	1.007
bt.B	0.941	1.004
bt.C	0.938	1.093
cg.A	0.750	1.193
cg.B	0.569	1.177
cg.C	0.330	1.083
lu.A	0.898	1.040
lu.B	0.893	0.887
lu.C	0.902	1.054
sp.A	0.843	1.008
sp.B	0.759	0.971
sp.C	0.787	1.146

Figure 6.18: NAS Benchmark results for kernels with a 4 KiB base page-size. All kernels are using the SF-VHPT HPW. Speed-up is over a standard 16 KiB base page size SF-VHPT kernel.

The `lu.B` benchmark shows a particular anomaly which suggests further investigation. Section 4.3.2 previously discussed the Itanium PMU and the hierarchy of events used to account cycles to a range of processor resources (Figure 4.4).

Investigation found the extra cycles for the `lu.B` were attributed to L1D pipeline events as shown in Table 6.17. As expected, these results show higher cycles accounted to TLB and HPW events for the fixed page-size kernel but the large-page kernel has a very large number of extra cycles attributed to the `L2BPRESS` event, indicating the L2 cache read queue is full and hence the processor is stalled waiting for data. The code in question is very loop based and it is possible that aggressive pre-fetching requests inserted by the compiler could end up flooding the cache when there is no interruption for TLB misses. This anomaly would require human intervention (most likely with a range of profiling and monitoring tools) to solve and serves to illustrate that the elimination of TLB misses may have unexpected flow-on effects to other areas of the program.

Figure 6.18 shows the results of running the tests with a smaller 4 KiB base page-size kernel. As with the results presented in Section 6.2.2 the smaller page-size kernel responds well to large-page support and (modulo the inconsistencies examined above) generally equals or beats the standard 16 KiB base page size kernels.

6.4.4 Other Results

The SPECWeb benchmark from Section 4.6.3 was repeated with large-page kernels but did not show any significant variations.

A selection of other benchmarks used in Navarro [Nav04] were also considered. In general, any benchmarks done in that work not presented here were not found to have a significant dependence on TLB behaviour

in our tests. For example the rendering process with the standard Povray [Pov] benchmark is largely CPU bound with a working set of less than 10 MiB. large-page support does reduce the number of TLB misses but since fault overheads are a small percentage of total time it is not enough to make a significant difference. Similar results were seen with a JPEG transformation test.

Investigation of database loads was also undertaken. The Open Source Database Benchmark [OSD] was run against a MySQL and again despite a considerable reduction in TLB misses a difference able to be considered significant was not observed. As mentioned in Section 3.5.3 this work does not support mapping the Oracle SysV shared global area with large pages at this stage so no direct comparison was undertaken.

6.5 Summary

There is no doubt that a range of applications can benefit from the advantages of transparent large pages. The extreme example of the matrix transformation benchmark presented in Section 6.2.1 achieves 100% TLB coverage with large pages and becomes limited only by memory bandwidth. The commercial workload simulator Tlbcover (Section 6.2.2) and scientific applications in the NAS suite also respond well (Section 6.4.3). However, the benchmarks have highlighted a number of important factors for the effectiveness of large-page support.

6.5.1 SF-VHPT HPW Performance

On Itanium large-page support "competes" against the fast and efficient HPW. Section 6.2.2 showed that on a commercial type workload the SF-VHPT HPW is very effective and in general was shown to cover a very large percentage of TLB misses. Although the SF-VHPT is not immune to extreme adversarial cases such as the sparse access across 32 MiB boundaries presented in Section 4.3.1, it would be a very rare application which would exhibit this behaviour. Good programming practice strives to exploit locality, but if profiling were to reveal such behaviour it could be combatted by increasing the base page-size to 64 KiB (expanding the area covered by a single virtual linear page to $\frac{65536}{8} \times 65536 = 512\mathrm{MiB}$).

6.5.2 Effect of Base Page-Size

The HPW is particularly effective when combined with the relatively large 16 KiB static page-size provided by Itanium Linux. Section 6.2.2 showed that for a TLB intensive benchmark, a great proportion of the benefits of large pages were gained by increasing from a 4 KiB to a 16 KiB page size.

However, results presented in Section 6.2.2 and Section 6.4.3 show that large-page support effectively removed the penalty of using a smaller page size. Smaller pages allow for a lower protection granularity and, combined with a system such as protection keys (Section 3.1.1) may lead to higher performance via greater potential for sharing translation entries.

6.5.3 Changed Behaviour

Several benchmarks showed quite different profiles when their micro-architectural behaviour was studied with an alternative HPW or large-page support (Section 4.3.2, Section 6.4.3). This is to be expected,

as tool-chains, operating systems and programmers tune their work to the status quo and fundamental changes such as multiple page sizes can therefore have many unintended implications. These anomalies can usually be removed or avoided via optimisation after analysis of the problematic code, so we can expect that over-time performance would only further improve.

6.5.4 Cost of Faults

As explained in Chaper 5, hardware limitations of dynamic large-page support on Itanium are expected to result in an increased TLB miss costs for large pages. This expectation was confirmed with a measurement of the costs presented in Section 6.3.2.

The ramifications are summarised in the TLB effectiveness results presented in Figure 6.12. This graph represents how many extra cycles of useful work must gained per TLB miss to absorb the costs of the more expensive TLB miss cost with large pages. The very long and flat tail of this graph means that as the average cycles between TLB misses gets higher it becomes significantly more difficult for large-page kernels to gain enough cycles to show significant performance improvements.

Two components of the SPEC suite highlight the extremes of this effect. The `vortex` benchmark shows the extra 300 cycles per TLB miss provided by large-page support does not raise the TLB effectiveness of the `vortex` benchmark above that of the unmodified SF-VHPT level (Figure 6.13). In contrast the `mcf` benchmark shows an extremely large increase in cycles between TLB misses (Table 6.7) and consequently gains the greatest speed-up. The access patterns of the applications, presented in Figure 6.14, highlight the reason for the difference. Large-page kernels are heavily penalised when required to constantly create and destroy small order pages (Figure 6.14a) and benefit from long, linear mappings (Figure 6.14b).

A consequence of these results is that, for some applications, the penalties outweigh the gains. This is most likely in smaller, short-run applications such as the kernel compile process presented in Section 6.4.2 where the cumulation of small overheads resulted in a performance decrease. Small working set and short lived processes have a much higher probability of suffering negative effects from large-page support.

6.6 Conclusion

This chapter has undertaken a detailed analysis of the effects of transparent large pages for Itanium on a wide range of benchmarks. The results have shown the implementation universally achieves its goal of increasing TLB coverage and reducing TLB misses. Transparent operation resulted in no benchmark requiring modification to take advantage of large-page support. Many of the benchmarks responded extremely well as their overheads from TLB misses were essentially eliminated. Results also showed that large-page support can allow the system to use a smaller base page-size without penalty.

The costs associated with this support are largely related to interactions with the Itanium HPW mechanisms. The disablement of the SF-VHPT for large pages leads to an increase in fault handling time which can often equal or in some cases outweigh the benefits of fewer TLB misses. Despite the ability of the LF-VHPT to hardware load large pages, the overheads discussed in Chapter 5 manifest in a large overhead with fault processing. Overall the LF-VHPT did not live up to expectations of providing higher performance with large pages.

The results therefore suggest that, for the Itanium platform, large-page support can be very useful for gaining a significant performance improvement from a range of applications. Whilst transparency is extremely useful for reducing programming effort and supporting unmodified applications, caveats apply such that users and administrators should enable large-page support selectively.

Chapter 7

Summary and Conclusions

7.1 Summary

Chapter 2 formed a foundation for the work, based upon analysis of the literature and prior work. It provided a taxonomy of existing implementations of large-page support. The chapter concluded that page size should be transparent to the user and implemented via a minimally invasive translation-replication scheme.

Chapter 3 introduced the Itanium MMU model, and in particular the SF-VHPT and LF-VHPT hardware page-table walkers. The general tradeoffs between the two forms of HPW were examined; SF-VHPT consumes more TLB entries but has a smaller cache footprint than LF-VHPT. The interaction between both forms of HPW and the proposed implementation of large-page support was also examined. The SF-VHPT is unable to load anything but the base page-size and needs to simulate a software-loaded TLB by marking large-page translation entries as invalid. The software walker can use the virtual linear table to access the underlying translation entry, but has the penalty of an expensive transition to kernel mode. On the other hand, the LF-VHPT can directly load large-page translations kept in a separate backing hash table. However, in the current implementation, this hash table is not shared by the operating system and thus creates higher virtual memory management overheads. Finally the infrastructure and implementation of the existing Linux support for non-transparent large pages, HugeTLB, was described in detail.

Chapter 4 evaluated the operation of the LF-VHPT HPW implementation on Linux. The results showed that although the cost of handling a fault is generally higher with the LF-VHPT implementation, the benefits of reduced TLB overhead become apparent under high TLB contention. Overall the performance impact compared to SF-VHPT is negligible.

Chapter 5 described implementation details of transparent large-page support.

Chapter 6 evaluated the implementation. The results show that the goal of reducing TLB misses and increasing coverage is achieved. However, the overall goal of increased general-purpose performance is shown to rely upon interactions with the Itanium HPW. The fundamental challenge is the delicate balance between higher latencies when resolving faults and the advantages of increased TLB coverage. Using the highly-tuned and effective Linux SF-VHPT for baseline comparison lead to less impressive improvements than in some previously published work which had started with greater TLB overheads.

In general, modifying the page tables via a translation-replication approach has again shown to be practical

with a successful port to the Itanium architecture.

The work has shown that although the LF-VHPT model has greater potential to load translation entries via the HPW, it is disadvantaged by not being the primary source of translation information in the system. This results in more page-table walks and higher overheads from fault latencies, leading to reduced effectiveness. Additional concerns include inefficiencies from the hash functions reliance on page size, significant code modifications required for integration into Linux and expanding the work to achieve the required scalability. Despite the advantages shown in some situations, the LF-VHPT did not in general meet expectations of higher performance.

7.2 Future Work

7.2.1 Greater Integration

This work has only examined the case of anonymous memory allocation. However, it should be possible to integrate with the page clustering work discussed in Section 5.6.2 to allow large pages to be instantiated for disk-backed and shared memory allocations. This would allow large pages to be used with a greater range of applications.

There is also potential to integrate the work with translation sharing features such as protection keys as discussed in Section 3.1.1. Increased translation sharing may benefit from a smaller protection granularity and hence smaller page size; benchmarks showed that large-page support could ameliorate overheads of lower base-page-size kernels.

7.2.2 Optimised Implementation

The “many eyes” advantage of open source software leads to implementations becoming more optimised over time. For example, there may be cycles able to be reclaimed from the hand-written assembly miss handlers which could lead to lower fault latencies or smarter algorithms for predicting the best possible page size. Any regressions on non-TLB intensive benchmarks are small and there is a high possibility of further work making positive incremental changes.

7.2.3 Transparency Heuristics

Expanding the implementation to better implement transparency is another area of future work. The simplest solution is a static approach where the operator annotates program binaries as being suitable for large pages, as is done in other systems. It may be possible for the kernel to keep some track of memory allocations and dynamically make a “best guess” about when to enable large pages. The trade-off is that this kind of measurement may further complicate and slow the fault-processing path, resulting in less applications being suitable candidates for larger pages. There is also a policy issue about the persistence of this data; ultimately a hybrid implementation with the kernel notifying a userspace daemon which keeps a record of a programs suitability for large pages may provide the best solution.

7.2.4 Page-Table Abstraction

This work has presented a minimally-invasive approach to enabling transparent large pages. Although the translation-replication scheme has been shown to be successful, there is a possibility a more radical approach to storing translation information that might achieve even higher performance (this was discussed in Section 5.2.1).

Abstraction of the page-table layers to eliminate the assumption of a hierarchical page-table would make it much easier to implement new and innovative data structures. In particular, this may benefit large-page support by making it practical to use the LF-VHPT as the primary source of translation information. Although these abstraction layers will incur a small performance penalty due to indirection costs, the largest barriers to implementation for Linux are probably related to concerns over stability and maintenance of a re-architectured virtual memory subsystem.

7.2.5 HPW Modification

The SF-VHPT HPW is very effective at covering TLB misses in a wide range of situations. From a software-implementation point of view, the ideal situation would be the ability of hardware to load the page size from a modified translation format (as discussed in Chapter 5).

A future implementation on an alternative architecture with more support for hardware loading of large pages may provide for an interesting comparison. For example, the ARM architecture allows hardware loading of a range of page sizes with translations stored in a hierarchical page-table. For large pages the hardware can skip a translation level, presenting the possibility that loading large pages may even be faster than loading smaller pages.

7.2.6 Portability

The transparent large-page support approach presented has previously been implemented on IA-32, Alpha and SPARC. The implementation on Itanium offers more evidence of the general portability of the approach and suggests it will be appropriate for other architectures.

Compared to other processors, Itanium offers great flexibility in choice of page size. Whilst this allows for greater opportunities to create large pages this work has shown a tradeoff with the costs of instantiating those pages. This balance is very specific to the MMU architecture of the processor and any new implementation would require careful analysis of effectiveness.

7.3 Conclusion

The isolation, resource allocation and security provided by the virtual memory abstraction is fundamental to the operation of modern operating systems. Reducing address translation overheads is therefore essential to achieve the highest possible performance. This work has examined transparent large-page support on Itanium Linux as a method to achieve this.

Large pages have been shown as effective in reducing TLB misses and increasing coverage. However, on Itanium large-page support comes at the cost of more expensive reloads. While there is an overall benefit

for those applications doing large-stride data transfers or long, linear and non-repeating contiguous access, when compared with the existing highly-tuned implementation many other benchmarks were shown to actually suffer with large pages enabled. However, large-page support was shown to make a small base page-size (4 KiB) feasible, which may be desirable for other reasons.

Transparent large-page support is only likely to be generally effective when large pages are not penalised with higher reload overheads. Since large-page support offers many benefits, architects need to look at providing systems with little or no penalty for large page reloads.

Therefore, transparent large-page support for general-purpose operation is unlikely to consistently provide a benefit to Itanium Linux. However, users with identified TLB intensive working sets are very likely to achieve increased performance; in this case transparency provides important benefits by avoiding costly programmer and administrator time spent modifying or tuning applications.

Ultimately, large-page support has been shown to have the ability to provide significant performance improvements and therefore remains a worth-while goal.

Bibliography

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

[BBB+91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[Bon94] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Technical Conference*, Boston, MA, USA, Winter 1994.

[Cha] Matthew Chapman. Itanium internals. `http://www.gelato.unsw.edu.au/IA64wiki/ItaniumInternals`, cited 14 Nov 2007.

[CWH03] Matthew Chapman, Ian Wienand, and Gernot Heiser. Itanium page tables and TLB. Technical Report UNSW-CSE-TR-0307, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, May 2003.

[Den70] Peter J. Denning. Virtual memory. *ACM Computing Surveys*, 2:154–189, 1970.

[Elp99] 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 publications page at `http://www.disy.cse.unsw.edu.au/`.

[FZC+01] Zhen Fang, Lixin Zhang, John B. Carter, Wilson C. Hsieh, and Sally A. McKee. Reevaluating online superpage promotion with hardware support. In *Proceedings of the 7th IEEE Symposium on High-Performance Computer Architecture*, page 63, 2001.

[GCC+05] Charles Gray, Matthew Chapman, Peter Chubb, David Mosberger-Tang, and Gernot Heiser. Itanium — a system implementor's tale. In *Proceedings of the 2005 Annual USENIX Technical Conference*, pages 264–278, Anaheim, CA, USA, April 2005.

[GL] David Gibson and Adam Litke. `libhugetlbfs`. `http://libhugetlbfs.ozlabs.org`, Cited 19th Dec 2007.

[Gor04] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[GS98] Narayanan Ganapathy and Curt Schimmel. General purpose operating system support for multiple page sizes. In *Proceedings of the 1998 Annual USENIX Technical Conference*, New Orleans, USA, June 1998.

[GW07] Mel Gorman and Andy Whitcroft. Supporting the allocation of large contiguous regions of memory. In *Proceedings of the 2007 Ottawa Linux Symposium*, pages 141–152, 2007.

[Hen00] J. L. Henning. SPEC CPU2000: measuring CPU performance in the new millennium. *IEEE Transactions on Computers*, 33(7):28–35, July 2000.

[HS84] Mark D. Hill and Alan Jay Smith. Experimental evaluation of on-chip microprocessor cache memories. In *Proceedings of the 11th International Symposium on Computer Architecture*, pages 158–166, New York, NY, USA, 1984. ACM Press.

[Int00] Intel Corp. *Itanium Architecture Software Developer's Manual Volume 2: System Architecture*, January 2000. `http://developer.intel.com/design/itanium/family`.

[Irw03] William L. Irwin. A 2.5 page clustering implementation. In *Proceedings of the Linux Symposium*, Ottawa, Canada, 2003.

[JM97] Bruce Jacob and Trevor Mudge. Software-managed address translation. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, pages 156–167, 1997.

[Kar92] Richard M. Karp. On-line algorithms versus off-line algorithms: How much is it worth to know the future? In *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing '92, Volume 1*, pages 416–429. North-Holland, 1992.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 3rd edition, 1997.

[KP06] Dave Kleikamp and Badari Pulavarty. Efficient use of the page cache with 64 KB pages. In *Proceedings of the 2006 Ottawa Linux Symposium*, volume 2, pages 65–70, 2006.

[KS02] Gokul B. Kandiraju and Anand Sivasubramaniam. Characterizing the d-TLB behavior of SPEC CPU2000 benchmarks. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems*, 2002.

[LE95] Jochen Liedtke and Kevin Elphinstone. Guarded page tables on MIPS R4600 or an exercise in architecture-dependent micro optimization. Technical Report UNSW-CSE-TR-9503, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 1995.

[Lie96] Jochen Liedtke. *On the Realization Of Huge Sparsely-Occupied and Fine-Grained Address Spaces*. PhD thesis, TU Berlin, Munich, Germany, 1996.

[Low05] Eric Lowe. Automatic large page selection policy. OpenSolaris project Muskoka, Sun Microsystems, March 2005. `http://www.opensolaris.org/os/project/muskoka/virtual_memory`.

[Lyo05] Terry L. Lyon. Method and apparatus for updating and invalidating store data. US Patent 6920531, 2005. Assignee: Hewlett-Packard Development Company, L.P., Houston, TX(US); filed Nov 4, 2003.

[Mar] John Marvin. Port of Shimizu superpages patch to ia64/x86-64. `http://lkml.org/lkml/2006/10/30/193`, cited 30 Oct 2007.

[McD04] Richard McDougall. Supporting mulitple page sizes in the Solaris operating system. Sun Blueprints Online, Sun Microsystems, March 2004.

[ME02] David Mosberger and Stéphane Eranian. *IA-64 Linux Kernel: Design and Implementation*. Prentice Hall, 2002.

[MMS90] Mark S. Manasse, Lyle A. McGeoch, and Daniel D. Sleator. Competitive algorithms for server problems. *J. Algorithms*, 11(2):208–230, 1990.

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

[Nav04] Juan E. Navarro. *Transparent operating system support for superpages*. PhD thesis, Rice University, Houston, Texas, April 2004.

[NK98] Karen L. Noel and Nitin Y. Karkhanis. OpenVMS Alpha 64-bit very large memory design. *Digital Technical Journal*, 9(4):33–48, 1998.

[OSD] Open Source Database Benchmark. `http://osdb.sourceforge.net/`, Cited 21st Dec 2007.

[Pov] POV-Ray – The Persistence of Vision Raytracer. `http://www.povray.org`, Cited 21st Dec 2007.

[ROKB95] 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*, pages 176–87, Santa Margherita Ligure, Itay, June 1995. ACM.

[Sez93] André Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 169–178, 1993.

[Sez04] André Seznec. Concurrent support of multiple page sizes on a skewed associative TLB. *IEEE Transactions on Computers*, 53(7):924–927, 2004.

[SMPR98] Indira Subramanian, Cliff Mather, Kurt Peterson, and Balakrishna Raghunath. Implementation of multiple pagesize support in HP-UX. In *Proceedings of the 1998 Annual USENIX Technical Conference*, New Orleans, USA, June 1998.

[SPE] Standard Performance Evaluation Corporation. *SPECweb99 Benchmark*. `http://www.spec.org/osg/web99`.

[ST03] Naohiko Shimizu and Ken Takatori. A transparent Linux super page kernel for Alpha, Sparc64 and IA32: reducing TLB misses of applications. *SIGARCH Computer Architecture News*, 31(1):75–84, 2003.

[Szm00] Christan Szmajda. Virtual memory performance. (Draft Copy), July 2000.

[TH94] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 171–182, San Jose, CA, USA 1994.

[TKHP92] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in supporting two page sizes. In *Proceedings of the 19th International Symposium on Computer Architecture*. ACM, 1992.

[Tra07] Transaction Processing Council. *TPC Benchmark C Standard Specification*, 5.9 edition, June 2007.

[Tro] John Tromp. The Fhourstones benchmark. `http://homepages.cwi.nl/~tromp/c4/fhour.html`, cited 2 Nov 2007.

[WEG+86] David A. Wood, Susan J. Eggers, Garth Gibson, Mark D. Hill, Joan M. Pendleton, Scott A. Ritchie, George S. Taylor, Randy H. Katz, and David A. Patterson. An in-cache address translation mechanism. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 358–365, 1986.

[Wie06] Ian Wienand. A survey of large-page support. Technical Report 10100:2006, ERTOS, 2006. (COMP9930 Report).

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, pages 1–116, London, UK, 1995. Springer-Verlag.

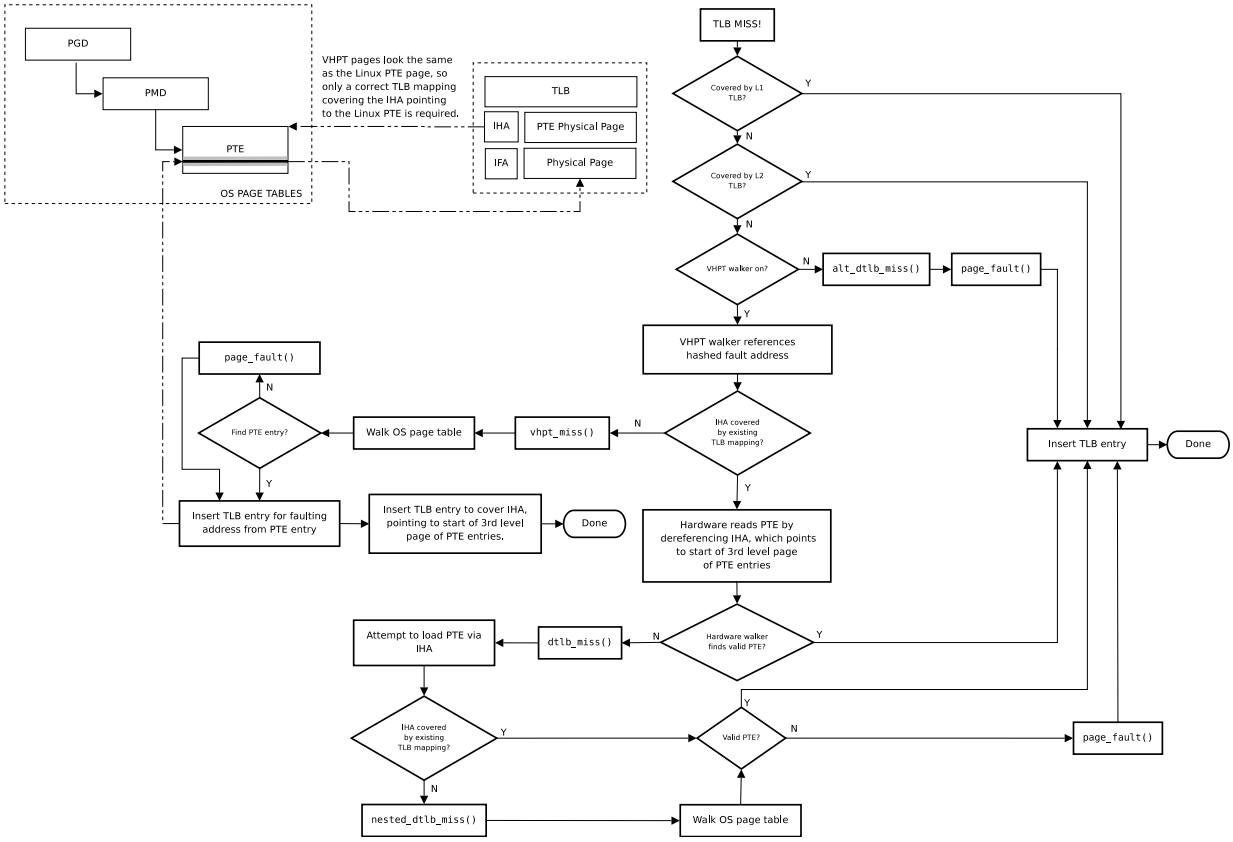
[WSF02] Simon Winwood, Yefim Shuf, and Hubertus Franke. Multiple page size support in the Linux kernel. In *Proceedings of the 2002 Ottawa Linux Symposium*, June 2002.

Appendix A

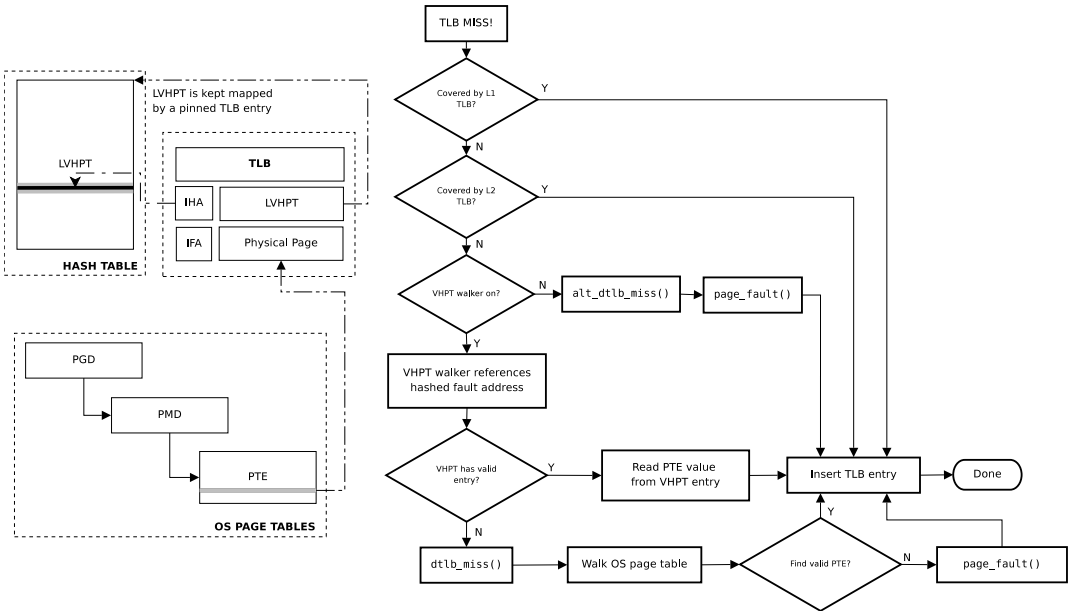
Linux Fault Paths

Below are flow-charts describing the overall fault handling path for Linux Itanium long and short format VHPT.

A.1 Short-Format VHPT



A.2 Long-Format VHPT



Appendix B

LMBench results

Below are tables of `lmbench`[MS96] results for various configurations of hardware page table walker. See Section 4.5 (page 41) for analysis.

- **None** : no HPW.
- **Short** : Virtual-linear array based short format HPW.
- **Long** : Hash-table based long format HPW.
- **Long-PTE** : As above, but with OS page table leaf entries doubled to accommodate page-size information.

Processor, Processes

Kernel	Null call	null I/O	stat	fstat	open close	signal install	signal handle	process fork	process execve
None	0.035	0.25091	1.162	0.316	2.683	0.319	1.810	72.3	299.0
Short	0.035	0.25050	1.190	0.319	2.701	0.319	1.812	68.6	290.2
Long	0.035	0.25882	1.187	0.316	2.723	0.305	1.878	71.9	297.7
Long-PTE	0.035	0.24389	1.245	0.315	2.690	0.302	1.831	73.5	301.1

Kernel	Mmap Latency	Prot Fault	Page Fault
None	193.6	0.645	1.00
Short	169.6	0.597	0.00
Long	214.6	0.610	1.00
Long-PTE	215.6	0.609	1.00

Table B.1: `lmbench` process microbenchmarks.

Local Communication bandwidths

Kernel	Pipe	AF/ Unix	TCP	File reread	Mmap reread	Bcopy (libc)	Bcopy (hand)	Memory read	Memory write
None	4069.10	4064.30	2040.42	1661.58	801.47	720.80	483.71	801.26	672.70
Short	4056.04	4022.15	2041.02	1661.21	806.84	726.45	486.20	806.89	675.6
Long	4008.63	4020.95	2028.96	1661.57	804.26	723.78	484.86	804.32	673.08
Long-PTE	4008.63	4020.95	2028.96	1661.57	804.26	723.78	484.86	804.32	673.08

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
None	1657.46	760.77	728.12	726.33	960.93	1545.17	711.93	2177.81	28.922
Short	1659.84	766.49	729.14	731.41	966.07	1550.92	716.22	2193.91	29.046
Long	1656.61	759.34	726.08	729.59	963.25	1545.83	713.88	2180.87	28.686
Long-PTE	1662.06	759.28	724.73	726.51	963.06	1543.09	714.10	2171.42	29.150

Table B.2: `lmbench` communication bandwidth results

Context switching with 0K							
Kernel	2proc	4proc	8proc	16proc	32proc	64proc	96proc
None	1.666	1.684	1.798	1.798	1.880	3.404	3.838
Short	1.628	1.682	1.924	1.800	2.112	3.144	3.524
Long	1.704	1.704	1.938	1.856	1.902	2.122	2.486
Long-PTE	1.732	1.732	1.944	1.834	1.876	2.168	2.520

Context switching with 4K							
Kernel	2proc	4proc	8proc	16proc	32proc	64proc	96proc
None	2.102	2.160	2.396	2.424	2.858	4.460	5.226
Short	2.094	2.162	2.402	2.434	2.820	4.232	4.884
Long	2.066	2.108	2.332	2.380	2.540	3.038	4.442
Long-PTE	2.170	2.202	2.436	2.438	2.588	3.012	4.020

Context switching with 8K							
Kernel	2proc	4proc	8proc	16proc	32proc	64proc	96proc
None	2.372	2.424	2.736	2.866	3.362	5.012	5.852
Short	2.386	2.426	2.748	2.876	3.312	4.642	5.402
Long	2.340	2.384	2.648	2.774	3.022	3.470	4.644
Long-PTE	2.440	2.474	2.714	2.902	3.074	3.578	4.536

Context switching with 16K							
Kernel	2proc	4proc	8proc	16prc	32prc	64prc	96prc
None	2.910	3.044	3.464	3.770	4.456	6.370	7.762
Short	2.938	3.046	3.496	3.802	4.246	5.996	6.898
Long	2.820	2.956	3.314	3.644	4.026	5.090	6.380
Long-PTE	2.942	3.082	3.506	3.784	4.090	5.502	7.004

Context switching with 32K							
Kernel	2proc	4proc	8proc	16prc	32prc	64prc	96prc
None	3.958	4.300	5.478	5.750	6.442	8.896	12.910
Short	3.984	4.314	5.502	6.006	6.170	8.100	12.310
Long	3.890	4.208	5.354	5.700	5.870	7.566	11.960
Long-PTE	4.020	4.260	5.466	5.744	6.004	7.650	12.158

Context switching with 64K							
Kernel	2proc	4proc	8proc	16prc	32prc	64prc	96prc
None	6.318	8.486	9.478	9.960	10.534	19.598	59.092
Short	6.204	8.458	9.470	9.634	9.988	23.494	57.580
Long	6.316	8.364	9.412	9.412	9.884	20.464	54.512
Long-PTE	6.368	8.416	9.508	9.454	9.924	19.414	55.230

Table B.3: Results of `lmbench` context switch overhead microbenchmarks.

Acknowledgements

I would like to thank the many people that have made it possible for me to complete this thesis.

Being involved with the Gelato project over the last several years has been a most wonderful experience, providing a range of opportunities I could never have expected. I wish to thank Hewlett-Packard for supporting research in Australia, but most importantly thank all those I have had the pleasure of collaborating with and learning from, especially Matthew Chapman, Al Stone, Lee Schermerhorn and Darren Williams.

The KEG lab may be a looser collaboration now than when I started, but no matter where people are its spirit lives on. It is truly humbling to work amongst such talent.

The technical side of this thesis is testament to the support of my supervisors Gernot Heiser and Peter Chubb.

Although I'm sure Gernot doesn't remember, he launched my career in Session 1, 2000 when I begged him to let me do his operating systems course despite not having completed one of the pre-requisite courses. It is still the best class I ever took, and I was proud to be involved in teaching it to a new generation. I am most thankful for his support both then and now.

Peter Chubb took me into the Gelato team and has been a dream boss ever since. His breadth of knowledge and experience is nothing short of astounding and I have been privileged to work with him.

The other side of a work such as this is attributable to those people who are unlikely to ever read it.

To all the desus — thanks for all the crazy adventures and here's hoping for many more.

I would never have taken on such a challenge had my parents not instilled in me a passion for learning. Their support has been constant, not the least of which involved giving me the Commodore 64 which started it all.

And of course my wife, Melinda. While this chapter now closes, I can't wait to start writing the next one together!