VIRTUAL MEMORY IN A 64-BIT MICROKERNEL

A DISSERTATION SUBMITTED TO THE SCHOOL OF COMPUTER SCIENCE AND ENGINEERING OF THE UNIVERSITY OF NEW SOUTH WALES IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Kevin John Elphinstone

August 31, 1999

Abstract

Virtual memory is a feature of most operating systems. It presents a level of indirection between the addresses that an application views, and the physical memory addresses used by the hardware. The benefits of virtual memory include: security, reliability, application transparent relocation of physical memory, and cache partitioning.

The page table is a critical component of a paged virtual memory system. The page table contains the set of translations that map virtual addresses to physical addresses. The implementation of a page-table structure affects the cost of virtual memory to applications. This thesis explores the cost of various page-table structures to applications in a 64-bit microkernel environment. The primary goal of the thesis is to identify the page-table structures most suited to a microkernel environment, i.e. an environment that is expected to efficiently support a diverse range of applications and operating systems, including single-address-space operating systems which are expected to feature large sparse address spaces.

This thesis examines the performance of real implementations of multilevel, hashed, clustered and guarded page tables in a real 64-bit microkernel, on an architecture (MIPS R4x00) featuring a software-loaded TLB. Simulation is not used to estimate performance. We examine page-table performance in terms of TLB-refill cost, page-table memory consumption, microkernel mapping primitives and address-space setup and tear-down costs.

The results show that each page-table structure has its strong and weak points. The results identify the *spill-over* effect in guarded page tables, and demonstrates the importance of *cache priming* in hash-based page tables. In conclusion, the results show that guarded page tables augmented with a software second-level TLB is the best choice in a microkernel environment. They do not perform significantly worse than other page-table structures in the benchmarks undertaken, and perform significantly better than the other page-table structures when sparse operations are involved.

Acknowledgements

I would like to thank the many people that I have had the priviledge to know in my years at the University of New South Wales. It is the people that make UNSW the great environment it is. There are too many names mention them all, but I would like to thank the following people in particular.

I'd especially like to thank my thesis supervisor Gernot Heiser. I am grateful for his expert guidance and encouragement. I'd also like to thank the Mungi operating-system project members, past and present, but in particular Jerry Vochteloo for his contributions both at work and at the bar.

I would like to thank my parents for sending me to the "big smoke" to attend university. Their constant support and encouragement enabled me to overcome many obstacles along the way.

Finally, I'd like to especially thank my wife Julia, for putting up with all the late nights, her constant support and encouragement, and for waiting for me at the finish.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Virtual memory [Den70] is a technique that provides a level of indirection between program memory addresses and real memory addresses. Programs address memory using virtual addresses which are translated to real physical addresses upon each memory access. Virtual memory is ubiquitous. Most processors support the use of virtual memory; most operating systems make use of virtual memory.

The use of virtual memory is advantageous for many reasons. It supports concurrent programs on a single computer with their own protected virtual address space. Relocation of programs in physical memory is trivial and transparent to the application programmer. Portions of programs can also be transparently moved to disk, which frees physical memory for other programs, and also allows a single program to use more memory than is physically available.

Modern virtual memory is typically implemented by *paging* [Mil90, JM98b]. A paged virtual-memory system divides both the virtual and physical address spaces into equal-sized blocks of memory. The blocks are usually termed *pages* in the virtual space, and *frames* in the physical space. Virtual memory accesses to pages are translated to physical memory accesses to frames. The set of translations that form a mapping between virtual pages and physical frames is stored in the *page table*. Given a virtual address, the page table is searched to locate the corresponding physical address.

The act of searching the page table each memory reference would be prohibitively expensive. Instead, an associative memory is used to cache a subset of the translations contained in the page table. The associative memory is usually termed a *translation lookaside buffer* (TLB) [CP78]. Each virtual memory refer-

ence is translated by the TLB into a physical memory reference. If the required translation is not present in the TLB, a *TLB miss* occurs, which loads the missing entry from the page table into the TLB, after which the translation process is completed and the memory reference continued.

TLB misses are handled by either software or hardware, or a hybrid of both. Hardware-based TLB-miss handlers use a state machine inside the processor to search the page table contained in memory, and subsequently load the missing translation. The state machine and page-table structure are predetermined when the processor is designed.

A more flexible approach is software-based TLB-miss handling. In this case, TLB-misses are handled by the processor taking an exception which invokes a software routine. The software routine is responsible for locating the missing entry by searching the page table, and then loading it into the TLB via special processor instructions. This scheme gives operating-system designers freedom to implement any page-table structure of their choosing. Consequently, the page table can be tuned to match the behaviour of the operating system.

This dissertation focuses on pages-table structures used in conjunction with a software-loaded TLB. In particular, this dissertation concentrates on their use in a 64-bit microkernel.

The concept of a microkernel (or as originally termed, a *nucleus*) was first demonstrated on the RC4000 computer [Han70]. The following quote describing HYDRA [WCC+74], a later microkernel-based system, also describes the basic philosophy of microkernels.

> ... at the heart of the system, one should build a collection of facilities of "universal applicability" and "absolute reliability" — a set of mechanisms from which an arbitrary set of operating system facilities and polices can be conveniently, flexibly, efficiently, and reliably constructed.

Mach [ABB+86] later popularised the microkernel concept, and formed the base of a significant body of operating-system research. However, microkernels gained the reputation of being slow, inflexible, and not at all "micro". This reputation was later shown to be ill deserved by the L4 microkernel [Lie95b, HHL+97]. L4 boasts fast IPC, flexibility, and is micro both conceptually and in size.

This thesis explores the effect different page tables have on a 64-bit version of the L4 microkernel. L4 is minimalistic in design. It features only three major abstractions: threads, address spaces, and inter-thread communication. These abstractions can be considered fundamental to operating systems in general. Hence, examining the effect of different page tables on the L4 microkernel can also be considered an examination of the effect of different page tables on operating system fundamentals.

1.1 Motivation

The desire to efficiently support the Mungi single-address-space operating system [HEV+98] is the motivation for studying page-table structures in the L4 microkernel. Like similar single-address-space systems [MWO+93, CLFL94], Mungi uses a single 64-bit address space which contains all data in the system. There is no traditional file system in Mungi, instead data simply persists in the virtual address space for as long as it is needed.

Objects are the basic storage abstraction in Mungi. Objects consist of contiguous sets of pages in the virtual address space. The system imposes no structure on objects. Objects can be contiguous or sparsely populated, or even totally empty. Objects persist at the virtual address at which they are initially allocated, thus preserving intra- and inter-object pointers.

While the Mungi address space could be contiguously populated with objects, the set of active objects at any point in time is likely to be a small subset of the total. Hence, the active address space at any one time is likely to consist of almost randomly scattered objects, with the objects themselves possibly sparsely populated.

It has been pointed out that traditional hierarchical page tables are unsuited to a single-address-space environment, and that hash-based page tables would be a good match [CLFL94]. Traditional hierarchical page tables are prone to high memory overhead in sparse address spaces. However, hierarchical page tables have the potential advantages of sharing page-table nodes among independent page tables, and performing hierarchical operations. Hash-based page tables have good memory consumption characteristics that are reasonably independent of virtual address-space layout. However, hash-based page tables treat virtual pages individually. Thus, operations on a group of pages requires probing for each potential page in

the group.

Given these trade-offs exist, the choice to optimise the page table for Mungi alone might penalise more conventional operating systems and applications running on the microkernel. Instead, I chose to take a microkernel approach and "make no assumptions about the particular strategy needed to optimise a given installation" [Han70]. The goal of this thesis is to examine, understand, and identify which page-table structures minimise the overall cost of the microkernel to applications, and also to examine the limitations of generality: can a single page-table design efficiently serve all needs? Such an optimised system might efficiently support both single address space and conventional operating systems alike.

1.2 Thesis Overview

This thesis aims to identify which page tables are most suitable for a 64-bit microkernel to concurrently support a wide variety of environments, but especially a single-address-space environment. The strategy taken to achieve this aim is to take a real implementation of a microkernel and examine its performance, rather than build a simulated environment. Typically, page-table investigations have used either trace-driven [HH93] or trap-driven [THK95] simulation. These techniques suffer from limitations including: only considering short traces, not considering cache effects completely, ignoring instruction costs in TLB-refill handlers, or ignoring kernel TLB misses. Examining a real implementation includes all these effects, albeit with the disadvantage of only examining a particular processor architecture in a particular machine configuration.

A new 64-bit implementation of the L4 microkernel was developed that allows different page-table implementations to be easily substituted with each other. The microkernel features an internal, page-table independent, virtual-memory interface. L4 provides a simple model of virtual memory to applications. As such, the internal virtual-memory interface mostly mirrors the virtual-memory primitives provided by the microkernel, and is much simpler than Mach-like virtual-memory systems [RTY+88].

The facets of microkernel performance that are affected by the underlying page-table structure are identified. They are TLB-refill performance, page-table memory consumption, microkernel (un)mapping performance, and address-space setup and

tear-down costs. A set of benchmarks are developed to test each identified facet of performance. The benchmarks consist of micro-benchmarking the microkernel primitives, monitoring performance of various applications, and monitoring synthesised scenarios that are expected to exist in a single-address-space operating system. Further details of the benchmarks are contained in Chapter 4.

Guarded page tables (GPTs) are one of the page-table types investigated. GPTs are attractive for use in the microkernel as they are hierarchical and feature satisfactory worst-case memory consumption. GPTs are extremely flexible, but at the same time complex to implement. Given the flexibility and the complexity of GPTs, it is not clear from the available theory how they will perform in a real implementation, or even what is the best implementation strategy. Hence, Chapter 4 is devoted to a practical evaluation of GPTs.

TLB-refill performance is shown to be a weaker aspect of GPTs. A software second-level TLB (STLB) is investigated to remedy this problem. A detailed discussion of implementation issues of STLBs in general is presented, after which STLBs are evaluated in terms of their effect on TLB-refill performance and in terms of their effect on other aspects of page-table performance. The results of the evaluation reveal the GPT+STLB hybrid page table to be very attractive for microkernel use.

The GPT+STLB combination is then compared to other page-table types, including multilevel-, hashed-, and clustered page tables. An implementation of each page table that is suitable for 64-bit address spaces is described. Each page-table is tested using the developed benchmarks. The benchmark results indicate that the GPT+STLB combination performs significantly better than the other page tables in some situations, and not significantly worse in the other situations. Unlike the other page tables tested, the GPT+STLB combination did not perform poorly in any of the tests undertaken.

The following is an outline which maps out the rest of the thesis.

Chapter 2 introduces the test bed used for the studies undertaken in this thesis. The test bed used is the L4 microkernel. It was re-implemented and, to some extent, internally re-designed for the MIPS R4x00 microprocessor. The chapter briefly describes L4 itself and then focuses on the microkernel's internal design pertaining to its use as a test bed.

Chapter 3 surveys the major page-table types. The chapter begins with the origins

of page tables themselves, and then goes on to review multilevel-, linear-, inverted-, hashed-, clustered- and guarded page tables.

Chapter 4 is a practical evaluation of guarded page tables. The chapter describes the guarded page tables used in testing. The benchmarks used for testing are also described and justified. The benchmarks test TLB-refill performance, page-table memory consumption, microkernel virtual-memory mapping performance, and task creation-and-deletion overhead. The results of the benchmarks are then presented and analysed.

Chapter 5 examines the addition of a software second-level TLB to guarded page tables to improve TLB-refill performance. The issues involved in designing a software second-level TLB are described in detail. The two designs chosen are then evaluated in terms of the benchmarks introduced in Chapter 4.

Chapter 6 is a detailed performance comparison of all major page-table types. The chapter describes the implementation of the page-table types used in testing. The page tables are benchmarked using mostly the same tests developed in Chapter 4. The results of testing are then presented and analysed.

Chapter 7 concludes and describes future work.

The main contributions of the thesis are:

1. An in-depth performance study of guarded page tables.

2. A study of a page-table's effect on microkernel performance outside the usual focus of TLB refill.

3. A comparative study of most major page-table types in a software-loaded TLB environment.

Guarded page tables are theoretically well understood. However, there is little experience and practical understanding of them. To my knowledge, this thesis is the first practical evaluation of guarded page tables. This thesis studies a real implementation of guarded page tables on a software-loaded TLB architecture, and provides a practical understanding of them. It identifies that they perform significantly better than worst-case predictions, even in really sparse environments.

In addition to TLB-refill, the studies in this thesis also examine the effect page-table choice has on microkernel primitives. More specifically, the page table's effect on microkernel virtual-memory mapping performance, and task creation-and-deletion overhead. Page-table selection is shown to have a significant effect on these over areas of system performance.

This thesis also provides a comparative study of real page-table implementations on a software-loaded TLB architecture. To my knowledge, all other comparative studies are based on either trace-driven or trap-driven simulation. Hence, this study includes all influences on page-table performance and uses metrics based on real time, not average cache-lines accessed or similar metrics which approximate actual performance. The strength of the cache priming effect is demonstrated. Cache priming is not visible in metrics such as cache-lines accessed.

Chapter 2

The L4 Microkernel

This chapter describes the L4 microkernel. L4 is the test bed used for experiments in this thesis. Firstly, this chapter briefly introduces the basic abstractions of the microkernel. It then moves on to describe the MIPS R4x00 implementation with a focus on the virtual memory subsystem and other test bed related issues.

2.1 L4 Abstractions

There is a central philosophy in the design of the L4 microkernel. The philosophy is that *a feature or service should only be included as part of the microkernel, if and only if its exclusion would prevent the implementation of a systems required functionality*. The most common and overwhelming reason dictating inclusion of a feature is security. A feature can never be moved out of the microkernel if it compromises security. Following this design philosophy ensures minimality of features, without restricting flexibility available to system designers.

L4 features three major abstractions: address spaces, threads, and inter-process communication (IPC). The following sections briefly describe each of the abstractions. For further details of the microkernel, a concise description can be found in the reference manual [EHL97].

Address Spaces

An address space at the hardware level is a mapping between virtual pages and either physical frames or null. This mapping is implemented and enforced by the hardware TLB, which caches the mapping represented by a page table.

The abstraction of an address space presented by the microkernel is a recursively defined mapping of virtual pages from the current address space, to virtual pages in other address spaces. The recursion eventually terminates with all address spaces being composed from one special address space (σ_0), which is a virtual representation of physical memory.

Figure 2.1 shows an example consisting of five address spaces derived from σ_0. The address spaces P1 and P2 are composed directly from σ_0, with P1 and P2 having physical memory divided equally between them. P1 and P2 act as pagers to clients C1, C2, and C3 via the same mechanisms that σ_0 acts as a pager to P1 and P2. P1 and P2 are free to present any abstraction of virtual-memory objects they choose using the virtual pages they have. Example virtual-memory objects are shared memory, anonymous paged memory, pinned memory, etc.

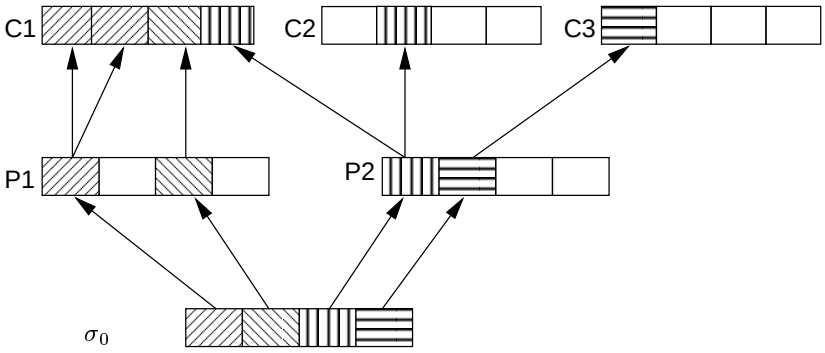


Figure 2.1: An example address space composition.

The microkernel provides three primitives for construction and management of address spaces: map, grant, and flush.

Map The map operation allows a thread in one address space to give access to its accessible virtual pages to all threads in another address space. This operation is achieved via IPC, and only achieved with the explicit consent of both the sender and receiver threads. Subsequent to the map operation, the virtual page appears in both the sender's and receiver's address space (at a potentially different address).

Flush The flush[1] operation allows the sender (mapper) of a virtual page to revoke access to the virtual page. Flushing a virtual page removes access to the page

[1]The term *unmap* is used interchangeably with *flush* to describe the same operation.

from the receiver's address space, and from any address spaces to which the receiver has passed on access to the page.

Grant The grant operation is similar to the map operation in that it is performed via IPC and gives access to the granted virtual page in the receiver's address space. However, the virtual page being granted is removed from the sender's address space. The sender thus foregoes its ability to revoke access to the page from the receiver. A grant operation effectively passes on the sender's control of a page to the receiver.

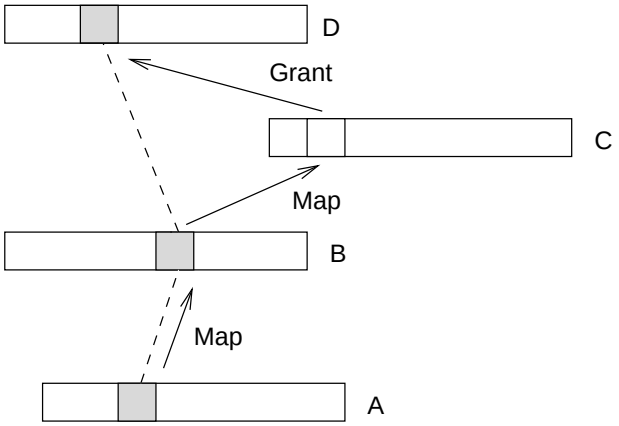


Figure 2.2: Example of L4 virtual memory primitives.

Figure 2.2 shows a simplistic example use of the L4 primitives. Address space *A* contains an initial page which it has access to. *A* maps the page to address space *B*, after which B gains access to the page with the same or reduced permissions. In turn, *B* maps the same page to *C*. Subsequently, *C* grants the page to *D*. Granting removes the page from *C*, consequently *C* can no longer flush the page from the address space receiving the grant (*D*). *A* can flush the page thus simultaneously removing access to it from both *B* and *D*, and even from *A* if *A* chooses. *B* can flush the page from *D* and *B* itself, *D* can only flush it from itself.

Threads

Threads are the basic abstraction of execution. Threads have associated with them a register set (consisting of at least a stack pointer and instruction pointer), an address

space in which it executes, and a *pager* thread to which page faults are forwarded via IPC. Each thread has its own unique system-wide identifier.

A pager thread is simply a thread that agrees to function as a mapper of pages to satisfy page faults of other threads. Pager threads do not have any extra abilities or properties when compared to other threads in the system. Indeed, any thread can become a pager, or cease to function as a pager.

IPC

Inter-thread IPC provides a primitive for: information exchange between threads, synchronisation, page-fault handling, and interrupt delivery. IPC is synchronous and allows both by-value (copying), and by-reference (mapping and granting) information transfer.

Device drivers run in user mode. Device interrupts are transformed into, and delivered via, IPC. Device registers are accessed via mapped memory.

2.2 MIPS/L4 Implementation

The original L4 microkernel was designed and implemented for the Intel x86 architecture. The major abstractions presented by the microkernel are intended to be general enough to be architecture independent. However, many of the internal algorithms are specific to the x86 architecture, and some are even specific to particular processors in the x86 family. Moving the microkernel to the 64-bit MIPS architecture required a complete re-write and some internal redesign.

The MIPS R4x00 architecture is significantly different to the x86 family. The MIPS R4x00 family of processors feature 64-bit integer and floating point operations. They have a larger register set consisting of thirty-two general-purpose 64-bit integer registers, and thirty-two 64-bit floating-point registers. CPU exceptions (including system calls, TLB refills, etc.) are handled mostly in software in contrast to the x86's mostly hardware-based approach. Software is responsible for saving (and restoring) state on exceptions, managing the TLB, and even managing the cache in some situations. The software control provides operating-system designers with flexibility in managing the hardware, but at the same time it provides a lot of complexity to manage.

The particular member of the R4x00 family used throughout this thesis is the R4700 [R4795]. It features a primary 16KB instruction cache and a 16KB data cache on chip. Both caches are two-way set associative, use a 32 byte line size, and FIFO replacement within a set. Both caches are virtually indexed and physically tagged. Secondary cache is external and optional. The actual machine configuration used for experimentation is described later in Section 4.2.1.

The R4000 architecture has a 64-bit virtual address space, however the R4700 only implements a 1TB (40-bit) user mode virtual address space together with a 64 GB physical address space. It uses a joint translation lookaside buffer (JTLB) to translate instruction and data virtual memory references to physical memory references.

The JTLB is a 48-entry fully-associative memory. Each entry maps an even-odd pair of virtual pages to their corresponding physical frames, giving the potential of 96 concurrently mapped virtual pages. Page size is per entry configurable from 4KB to 16MB in powers of 4.

An 8-bit address-space identifier (ASID) is associated with each entry in the JTLB. The ASID is used together with the virtual address when checking for a match, thus allowing multiple address spaces to exist in the JTLB simultaneously, which reduces the need for JTLB flushing during context switching.

The handling of JTLB misses is via a *TLB-refill* exception and a software routine to load a new entry into the JTLB. Other TLB related exceptions are handled by the processor general-exception mechanism, alleviating the TLB-refill routine from determining the exception involved, and allowing it to be optimised solely for refill. Refill software can overwrite selected TLB entries or use a hardware provided mechanism to overwrite a randomly selected entry.

To use the kernel as a test bed for a page-table investigation, the kernel needs to be representative, i.e. it should be as efficient as possible (or at least as efficient as other kernels), and not sway the results by being a poor implementation. L4/MIPS features IPC times of 99 CPU cycles, which is comparable to other L4 implementations [LES+97]. This is the approximate IPC overhead experienced by applications mapping memory using IPC, i.e. if mapping a page of memory costs 5 microseconds, one microsecond of that time is due to IPC overhead. Hence, the cost of mapping operations is clearly visible and readily comparable between different page-table implementations.

The cost of a null system call (`id_myself`) is 56 cycles. This is the overhead of entering and exiting the kernel experienced by applications calling kernel primitives, including the *unmap* system call. Spawning a thread costs 105 cycles, thus creating a task (which consists of a thread, an address space, and extra bookkeeping) should clearly show address-space creation overhead.

2.2.1 The Mapping Trees

The L4 *map* primitive allows cooperating threads to securely construct a virtual address space in terms of other virtual address spaces. The virtual address-space composition eventually recurses such that all virtual address spaces are defined in terms of the σ_0 address space, i.e. physical memory.

The flush operation allows a thread to remove any mapping derived from an accessible virtual page in its address space. In order for the kernel to provide this functionality, it must record the relationship between all accessible virtual pages in the system. The kernel achieves this by constructing and maintaining *mapping trees* during virtual-memory-primitive invocation.

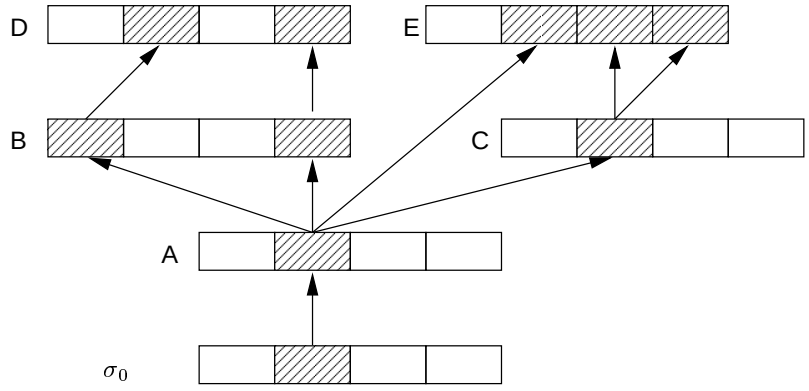


Figure 2.3: Example address-space hierarchy.

For every accessible virtual page in the system, mapping trees record which page a particular page was derived from, and which pages are subsequently derived from the page. Referring to Figure 2.3, we see that the dependency relationship between derived mappings forms a tree-like structure rooted at the a virtual page in the initial address space (σ_0). σ_0 maps each page idempotently and only once[2],

[2]The restriction of mapping pages only once provides simple security to applications. Control of memory is based on a first-come first-served basis.

thus forming an initial trunk of the tree. After the initial mapping, a pager is free to map the page to as many clients as it chooses; in turn, clients can do the same.

In designing a data structure to represent the mapping trees, the following must be considered.

- A virtual page can have an arbitrary number of mappings derived from it.
- A node in the tree is allocated for each active mapping in the system, hence the structure must be space efficient.
- A node in the tree is allocated, traversed, or deleted for each map or flush operation. Tree manipulation must be time efficient.
- The kernel is to be used as a test bed for page table implementations, therefore the data structure cannot rely on the structure of, or properties of the targeted page table.

The data structure chosen to implement the mapping trees was a 24 byte structure as detailed below as a C type definition, with the fields described afterwards.

```
typedef struct {
  uint64_t vaddr;
  uint32_t task;
  int32_t parent;
  int32_t child;
  int32_t sibling;
} mt_node_t;
```

vaddr is the 64-bit virtual address associated with the node.

task is the address-space number in which the above virtual address is contained.

parent is a pointer to the mapping-tree node, i.e. the virtual address and address space, from which this node is derived.

child is a pointer to a mapping-tree node derived from this node.

sibling is a linked-list pointer for a linked list starting at a child node that contains the list of all other nodes derived from the parent node.

Note the use of 32-bit pointers that rely on sign extension to form valid 64-bit pointers. The MIPS R4x00 architecture allows this simple optimisation as physical memory is accessed through a window in the upper 2 gigabytes of the 64-bit kernel virtual address space.

The most important feature of the above structure is the use of the pair (*virtual address*, *task*). The pair is used to locate page-table entries requiring manipulation, and in operations to ensure TLB consistency. Another potential implementation might have been to simply store a pointer to the page-table entry associated with the mapping-tree node. However, doing this prevents the goal of having a mapping structure that is independent of the page-table structure. A pointer prevents the page-table independence goal for two reasons:

1. It forces page-table entries to be at a fixed location once allocated. This unfairly disadvantages page-table structures which rely on dynamic movement of page-table entries to achieve better performance.

2. Given a pointer to a page-table entry in some page-table structures, it is not always possible to quickly deduce the virtual address and address space associated with the entry. The virtual address and address space is required for selective TLB consistency operations. A page table requiring global TLB flushes would be at a significant disadvantage when compared to a page table that supports selective flushing of only a single inconsistent TLB entry.

Figure 2.4 shows an example mapping tree for the shaded page in the previous address-space example (Figure 2.3).

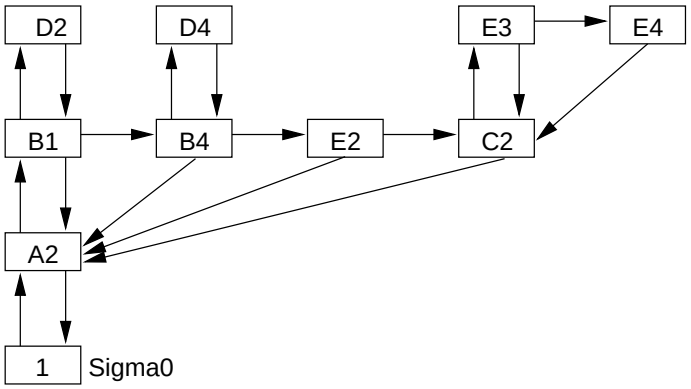


Figure 2.4: Example Mapping Tree.

2.3 The Internal Page-Table Interface

The internal page-table interface is a page-table independent interface to the page-table data structure. The usual motivation for such an interface is portability, i.e. the desire to encapsulate the machine-dependent page tables into a simple, well defined module [RTY+88]. In this thesis, the motivation for the interface is to provide a clean boundary between the page tables and the rest of the kernel, so that a page-table implementation can be easily substituted with another.

The page-table interface is not machine independent because it exposes the page-table entries to the rest of the kernel. However, the kernel itself is mostly machine dependent, and as such, the machine dependence of the interface is irrelevant.

> We are, however, unable to measure execution time on a real system ... it requires implementing all different page-table organisations in a commercial operating system. Implementing even a single page-table organisation is a multi-man-year project ... [THK95]

Contrary to this quote, this thesis (and Mach [RTY+88]) show that with carefully interface design, it is possible to prototype and test competing page-table implementations in a reasonable amount of time.

The page-table interface is briefly described below.

vm_new_as Creates an address space for a new task. It initialises a page table if necessary.

vm_delete_as Deletes an address space. It garbages collects the memory associated with the used page table.

vm_lookup_pte Given a virtual address, this function returns the corresponding page-table entry, if any.

vm_map Given two address spaces and a power-of-2 region, this primitive performs the L4 *map* operation.

vm_fpage_unmap Given an address spaces and a power-of-2 region, this primitive performs the L4 *flush* operation.

vm_sigz_insert_pte This function is used to insert a page-table entry in the $\sigma 0$ address space. It initialises the root of the mapping tree for that particular frame.

vm_tcb_insert Inserts a page-table entry for a thread control block in kernel space. It is expected to implement semantics of being globally shared among all address spaces.

vm_tcb_unmap Unmaps the thread control block from kernel space, returning all frames to the kernel frame allocator.

Chapter 3

Page-Table Structures

This chapter provides a review of page tables. It begins with a brief introduction to page tables, followed by reviews of the basic page-table types.

3.1 Introduction

Demand-paged virtual memory was first demonstrated on the Atlas computer [Fot61]. The Atlas featured a 20-bit address space of 48-bit words. The Atlas address space, core memory, and drum backing store was divided into 512-word blocks. The blocks in the fast core memory acted as a cache of the blocks on the slower drum memory. Applications were presented with a 20-bit virtual address space, independent of whether a block of memory was in core or on the drum. Address translation was used to preserve block (page) location within the virtual address space while blocks moved into, and out of, differing locations in the core memory.

Each 512-word block in core memory had an associated *page address register* which contained the current virtual address associated with the block. Address translation was performed by comparing the virtual page number of the current virtual memory access with all page address registers in parallel. When a match occurred, the associated core block was used as the virtual page. When no match occurred, a page fault was signalled resulting in the *supervisor* fetching the appropriate block from the drum and subsequently replacing a block in core with it.

This address translation via a register per potential translation was typical of early machines which featured either small core sizes, or small virtual address

spaces, or both. For example the PDP-11 featured 8K pages and two 64K virtual-address spaces for code and data. A translation register was used per virtual page giving 16 translation registers on the machine.

The register-per-translation technique did not scale well to newer machines such as the GE645 [BCD69, Org72], and the IBM 370 [CP78]. These machines had a larger number of potential translations due to larger physical and/or virtual memories. To overcome the scaling problem, both these machines possessed two features that now dominate modern virtual memory implementation: the page table, and the translation lookaside buffer (TLB). The page table was a per-segment array of translations stored in main memory rather than in registers. To avoid accessing the page table to translate every memory reference, an associative memory (the TLB) cached a subset of page-table entries. Hits in the TLB resulted in fast translation; misses resulted in a page-table lookup and loading of the appropriate page-table entry into the TLB.

Modern microprocessor virtual-memory implementations evolved from these beginnings. The TLB and page-table structures have been intensively studied resulting in a variety of implementations. Smith provided a bibliography on early paging-related research [Smi78a]. Milenkovic provides a good survey of microprocessor memory management in the late 1980s [Mil90]; Jacob provides a more recent survey [JM98b]. TLBs are not the focus of the thesis so I do not describe them further.

The following sections survey the major page-table designs. For each page table, the basic structure and translation mechanism is described. The translation section is described assuming the absence of a TLB for illustrative purposes. The advantages and disadvantages of each page table is described. The likely effect of using the page table to support 64-bit address spaces is also examined.

3.2 Multilevel Page Tables

Early page tables consisted of simple arrays of entries representing address translations. However, simple arrays do not efficiently represent large, discontiguously populated, virtual address spaces. For example, a processes' virtual memory layout typically looks similar to the representation in Figure 3.1. The *text* and *data* sections are located at the bottom of the address space and expand upwards. The

stack section is located at the top of the address space and expands downwards. A simple array-based page table representing such an address space layout is space inefficient. The centre of the array is unused.

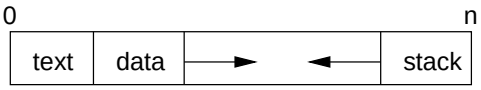


Figure 3.1: Typical user address space layout.

If the architecture supports segmentation with a per-segment page table, segmentation can avoid the problem while segments remain contiguously populated. A less complex solution, both from a hardware and software perspective, is to use a multilevel page table.

A multilevel page table consists of two or more levels of array-like nodes forming a tree-like structure. Multilevel page tables are also termed *forward-mapped* or *top-down* page tables. Several examples are reviewed in [Mil90], including the SPARC reference MMU and the Motorola 68000 series microprocessors. Figure 3.2 illustrates a two-level MPT. In this example, a selection of upper bits of the virtual address (I1) is used to index the root of the tree. The root of the tree contains pointers to nodes in the next level of the tree. The pointer-selected node is indexed by a selection of middle bits of the virtual address (I2), which selects the page-table entry to be later combined with the offset bits to form the physical address.

The MPT efficiently supports the typical split address-space layout with the text and data at one end, and the stack at the other. The second-level nodes associated with the unused centre region of the address space are not allocated until needed. Traditionally, an MPT also allows for part of the page table itself to be paged. Paging the page table conserves physical memory by storing inactive page-table nodes on disk. The inactive nodes contain the disk-block addresses of the paged out memory. However, most systems separate address translation from backing-store management. Only translation information is stored in the page table, enabling the encapsulation of page management in a "hardware address translation layer" [GMS87]. Backing-store management is handled independently. Thus, it no longer makes sense to page the page table to disk as it is faster to reconstruct a page-table node from information held elsewhere, rather than paging it in from disk.

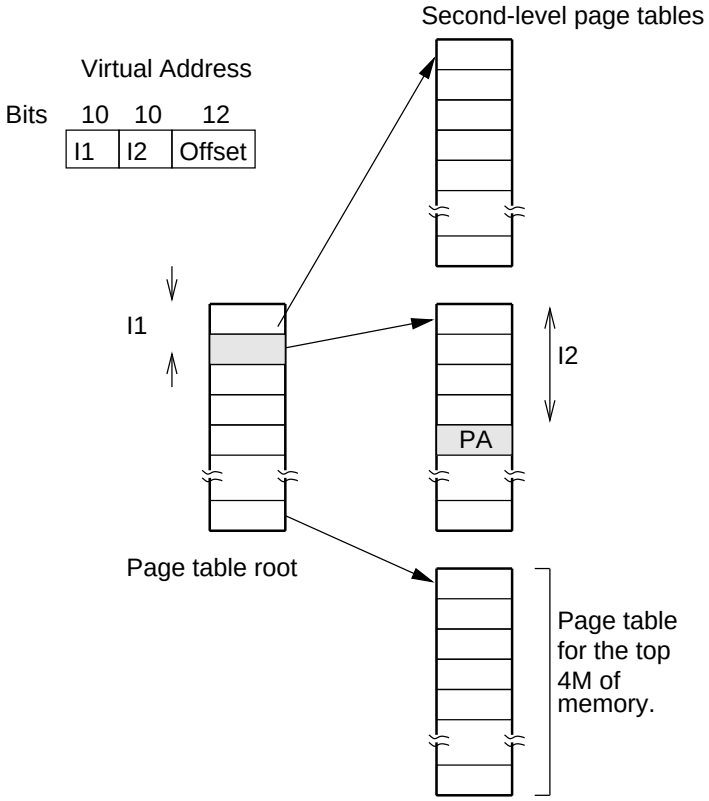


Figure 3.2: Example multilevel page table.

Being a tree-like structure, the MPT supports hierarchical operations. Operations on regions can be applied at higher levels in the tree, hence encompassing all nodes contained in subbranches. This provides for fast locking and unlocking of regions which are aligned appropriately with the page-table structure. A tree-like structure also allows sharing of branches in the tree. Sharing allows for simplified and more efficient management of a global memory region [YR93].

An MPT allows basic support of different page sizes by terminating translation at higher-level nodes. MPTs efficiently support in-order traversal which provides for operations on groups of pages.

However, the granularity of the node size in an MPT is a disadvantage. As a consequence of the typical 4K node size of the MPT, hierarchical operations, tree sharing, and multiple page-size support is only available at a very coarse granularity. The use of these properties is not arbitrary. Operations relying on them, must take page-table structure into account. Arbitrary sparsity is also not efficiently supported. For example, assume 32-bit page-table entries, a 4K MPT node size, and a 4K page size. In this situation, each leaf MPT node stores page-table entries for 4M of virtual address space. If the virtual address space is constructed such that a single 4K page is place at each 4M boundary, then at least a 4K page-table node is needed to represent each 4K page.

Supporting a 64-bit virtual address space with an MPT involves a trade-off. Designing an MPT to accommodate 64-bit translation requires either increasing the node size to translate more bits at each level, and/or increasing the number of levels. Increasing the node size has the disadvantage of increasing memory consumption in sparse environments. Increasing the number of levels has the disadvantage of increasing the number of memory references required for each page-table lookup. Therefore, MPTs inefficiently support translation of large sparse address spaces in either the space or time domain.

3.3 Linear Virtual Arrays

A linear virtual array is an array of page-table entries stored in the virtual address space. Unlike a simple page-table array stored in physical memory, a linear virtual array efficiently supports the conventional address-space split as the unneeded centre of the array is not backed by physical memory until it is required.

TLB misses can be serviced by indexing the array based on the missed virtual address and loading the appropriate entry. A lookup requires a trivial calculation based on the virtual address, and a single memory reference. This is, in theory, both simpler and faster than searching through several levels of an MPT. However, accessing the array virtually requires a TLB entry to be present for the LVA itself. If a TLB entry is not present, resulting in a nested TLB miss while servicing the original TLB miss, then either the nested miss is resolved via a page-table entry elsewhere in the/another LVA, or the nesting misses eventually resolve as the root of the LVA is stored in physical memory.

The classic example of an LVA is the VAX [LL82, CE85]. A simplified diagram is shown in Figure 3.3. The page table for user space is an array in kernel virtual space. The page table for kernel virtual space is an array in physical memory.

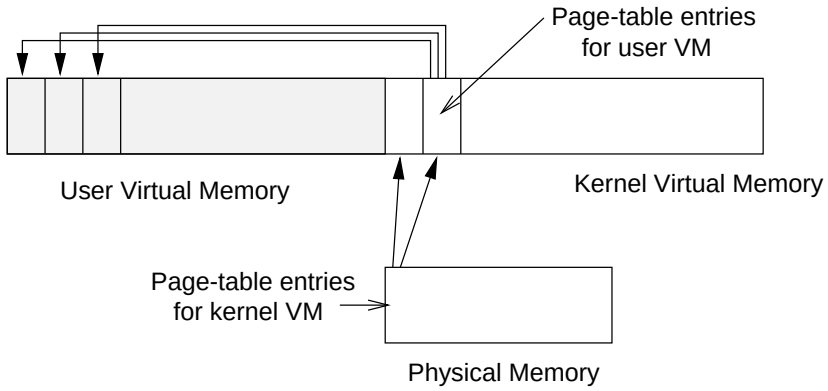


Figure 3.3: Simplified VAX page table diagram.

The VAX page table can be viewed as a two-level structure with the root node in the physical address space and the leaf nodes placed consecutively in the virtual address space. Upon a TLB miss, the leaf nodes are indexed first to lookup a page-table entry. If the TLB misses again during this lookup, the root node is used to resolve the second miss, after which the first miss is resolved.

The LVA can be considered an MPT that is searched from the leaf nodes to the root, rather from the root node to the leaves. To support a 64-bit address space requires many levels in the tree. For example, assume 32-bit page-table entries together with a 4K hardware page size. A 64-bit virtual address space needs a 2^{54}-byte LVA to store all the page-table entries required to use the entire address space. A second 2^{44}-byte array is needed to store the page-table entries for the first level

of the page table. The second array needs a third 2^{34}-byte array, the third needs a fourth 2^{24}-byte array, the fourth needs a 2^{14}-byte fifth, and the fifth needs a 2^4-byte sixth level. The sixth level is placed in physical memory to avoid needing more page-table entries.

Note that if all the virtual page-table levels are in the one address space, then the 2^{54}-byte level-one LVA contains the 2^{44}-byte level-two array at the appropriate place to provide entries for the level-one array. Similary, the level-three, level-four, and level-five arrays are all appropriately located within the level-one array. Figure 3.4 shows diagramatically how the smaller arrays are structured within the larger one.

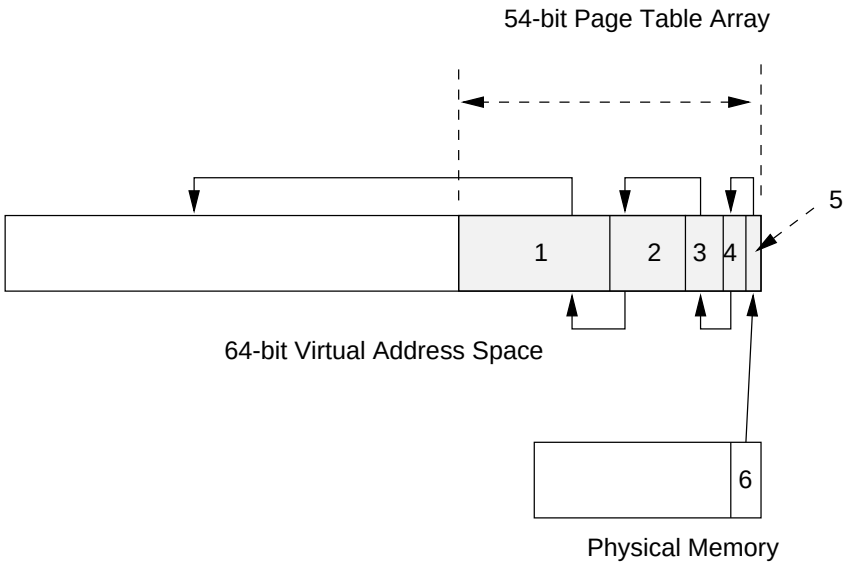


Figure 3.4: Simplified 64-bit 6-level LVA (not to scale).

A TLB miss generated by accessing user space may subsequently generate a TLB miss while accessing the first-level LVA. This nested TLB miss may trigger additional nested TLB misses in the second, third, fourth and fifth levels. Nested TLB misses are more time consuming to resolve than simple root-to-leaf tree traversal, especially on a software-loaded TLB architecture [NUS+93]. One strategy to avoid nesting is to partition the TLB into a region for user TLB entries and system TLB entries. The idea being that only a few system entries are needed to map the user's page-table entries, so partitioning results in a small reduction in TLB capacity, but expensive-to-replace system TLB entries are not displaced by user TLB

entries.

Partitioning is not likely to be sufficient for a 64-bit address space for two reasons. Firstly, instead of one level of virtual page tables needed in the VAX, 64-bit addresses in the scenario described above need 5 levels of virtual page tables, which would require a significant fraction of the TLB to cache even a small virtual-page-table working set. Secondly, 64-bit address spaces will not be contiguously populated with the typical text, data, and stack layout. Address spaces can contain regions of shared libraries, memory-mapped files, persistent stores and objects backed by external pagers (these are described in more detail later in Section 4.3). The more active regions in user address spaces, the larger a TLB partition required [NUS+93]. In the case of very sparse accesses (not necessarily as a result of a sparsely allocated address space), 5 TLB entries per data page would be required.

Hybrid LVA implementations are a more promising approach. Hybrid approaches use an LVA for one level of page-table entries, and store the page-table entries for the LVA itself in physical memory, thus limiting nesting of TLB misses to one level. Examples of this approach include caching higher-level page-table entries in a software TLB to reduce nesting [BKW94], or storing higher-level page-table entries completely physically in an MPT or an IPT (as described in following section). However, if the address space is sparsely allocated, the need to allocate page tables page-wise will lead to high memory overhead independent of nesting.

3.4 Inverted Page Tables

Inverted page tables (IPTs) [Coc81] get their name from inverting the virtual-to-physical translation map. Instead of a page table, which is indexed by virtual page number and stores a physical frame number per virtual page; an inverted page table is indexed by physical frame number and stores a virtual page number per physical frame. A major advantage of this scheme is that *page-table size is approximately proportional to physical memory size, not virtual address-space size*.

Given a virtual page number, virtual-to-physical address translation is achieved by searching the page table to find a matching virtual page number. The physical frame number is deduced from the table index to the matching entry. The searching is usually performed with the aid of a hash table [IBM78, CM88, Lee89]. Such an IPT is illustrated in Figure 3.5. The IPT consists of two parts, the *hash table* (or

hash anchor table) and the *page directory*[1] (or *frame table*). The page directory has an entry for each physical frame, and is indexed by physical-frame number. Each page-directory entry contains the virtual-page number of the page associated with the entry, together with other attributes such as permissions, overflow link, etc. Each hash-table entry contains a physical-frame number, which is an index to the page directory.

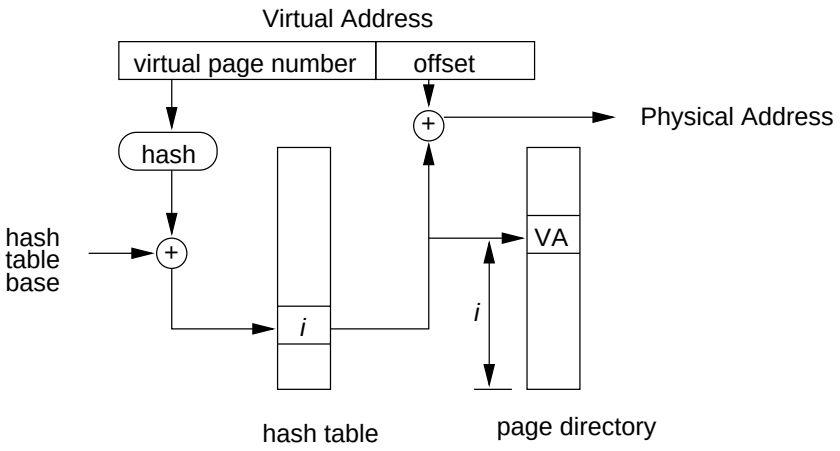


Figure 3.5: A diagram of an inverted page table.

To translate a virtual address, the virtual-page number is hashed to form an index into the hash table. A page-directory index (i) is obtained by loading the selected entry. If the selected entry in the page directory matches the page number, the physical frame index (i) is combined with the offset bits to form the physical address.

Collisions in the hash table are usually resolved by linear chaining. Selecting the hash-table size is a trade off between table size (load factor) and average collision-chain length [Knu73].

The IPT has the desirable property of being able to compactly store sparse address spaces. Given a reasonable hash function, and a non-pathological address-space layout, the IPT performance is relatively independent of the virtual-address-space layout. However the IPT has several drawbacks:

- It lacks aliasing support. Each physical address can have only one virtual to physical translation associated with it. This prevents sharing memory at

[1]Note that I use the term *page directory*, rather than *page table*, to avoid confusion by overloading the later.

different addresses unless the page table is continually modified at appropriate times. IPT systems generally partially overcome this by having a global address space.

- An IPT that is searched via a hash table always requires at least two memory references each lookup, one to the hash table, and one to the page directory.
- Discontiguities in the physical address space result in unused regions in the page directory. Discontiguities occur with memory mapped I/O devices, as they are not necessarily placed contiguously with respect to RAM and each other.
- IPTs treat virtual memory as a composition of single virtual pages with no relationship between them. Operations on objects consisting of groups of pages require either an IPT probe per page in the object, or a scan of the entire page directory.

Using an IPT for 64-bit address translation is no more (or less) complicated than using an IPT for 32-bit address translation, except that the page-directory entries may need to be enlarged to accommodate larger virtual-page numbers. The IPT advantages and disadvantages remain the same.

3.5 Hashed Page Tables

The hashed page table (HPT) places both the virtual-page number and the physical frame number in the hash table [HH93]. It is similar to a hardware based translation scheme used in MONADS [RA85]. The addition of the physical frame number to the hash table gives the HPT several advantages over the IPT.

- The HPT removes the memory reference to the hash table typically used to search the IPT. Potentially, TLB-misses can be serviced with a reference to a single cache line, which improves TLB-refill performance.
- The HPT supports aliasing. Multiple virtual addresses can be translated to the same physical address via multiple entries in the HPT.
- Since the HPT is not indexed physically, it accommodates holes in the physical address space efficiently.

Figure 3.6 illustrates a HPT. To translate a virtual address, the virtual-page number is hashed to form an index into the HPT. If the selected entry matches the virtual-page number, the co-located physical-frame number is combined with the offset bits to form the physical address.

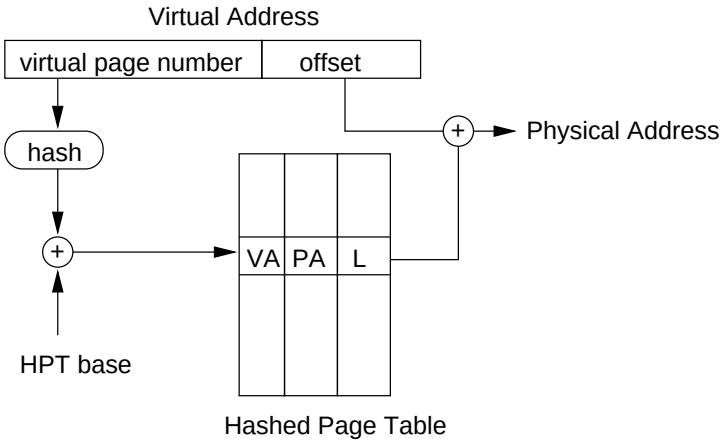


Figure 3.6: A diagram of a Hashed Page Table.

Like the IPT, using the HPT to support a 64-bit virtual address space simply involves enlarging HPT entries to accommodate larger virtual-page numbers.

3.6 Clustered Page Tables

A clustered page table (CPT) is basically a hashed page table with the addition of subblocking [THK95]. Figure 3.7 shows an example configuration of a CPT. There are other structural configurations of CPTs which are described in [Tal95], but they are not detailed here.

Translating a virtual address with a CPT is similar to translating an address with a HPT. The virtual block number is hashed to form an index into the CPT. If the index-selected entry matches, the block offset is used to index and select a page-table entry. The page-table entry is combined with the remaining offset to form the physical address.

Clustering the page table aims to reduce the space overhead by storing several consecutive page-table entries with a single tag and link pointer. Clustering is effective only if the address-space allocation is “bursty”, i.e. clustering relies on

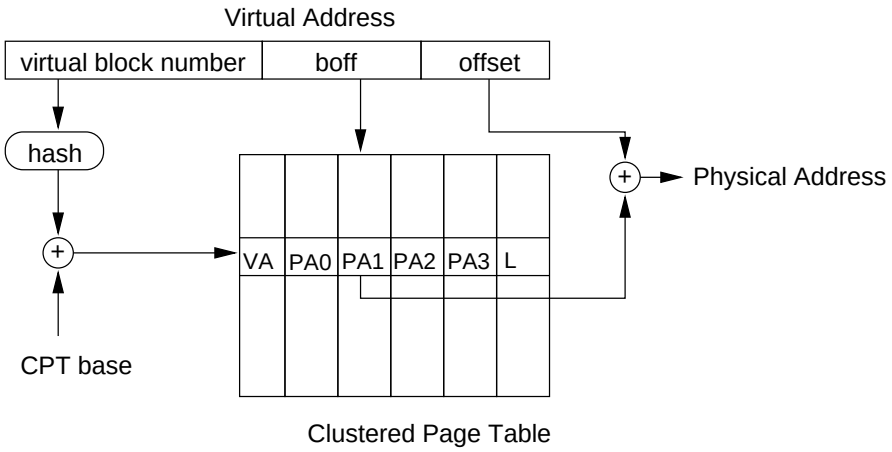


Figure 3.7: A diagram of a clustered page table with subblock factor 4.

programs mapping objects into their address space in groups of contiguous virtual pages. Address spaces consisting of isolated virtual pages will result in higher space overheads when compared to a HPT.

Clustering also aims to efficiently support two different page sizes, i.e. a smaller base page size, and a larger page size consisting of a subblock-factor number of base pages. Supporting two different page sizes in a HPT can be achieved by replicating the larger page's page-table entry in all potential base-page locations in the page table, or by using two page tables (one for each page size). In contrast, the CPT hash function is based on the block size, not the base page size. This makes it simple to transform a block of base pages to (or from) the larger page size, thus permitting the CPT to support both page sizes without replication.

3.7 Guarded Page Tables

3.7.1 The Basic Idea

This section contains excerpts from previous work [LE95]. Guarded Page Tables [Lie95a, Lie96] combine the advantages of tree-structured multilevel page tables and hashed page tables: unlimited sparsity (2 page table entries per mapped page are always sufficient), tree structure (subtree sharing, hierarchical operations), and multiple page sizes. These properties are described in more detail later and in [Lie96]. What follows is only a short sketch of the basic mechanism.

The main problem with multilevel page tables is sparsity: they need huge amounts of page table entries for non-mapped pages. Referring to Figure 3.8, where the mapping of page $11\,10\,11\,00$ in a sparsely occupied address space is shown. (For demonstration purposes we use very small addresses and small page tables. Nil pointers are marked by "•".) The second- and third-level page tables are extremely sparse page tables: each contains one single non-nil entry. Consequently, there is only one valid path through these two tables: when the leftmost two bits are "11", the subsequent address bits must be "10 11"; all other addresses lead to page faults. As shown in Figure 3.8, we can omit the two page tables and skip the associated translation steps. Whenever entry 3 of the top-level page table is reached, we have to check whether "10 11" is a prefix of the remaining address. If so, this prefix can be stripped off, and the translation process can directly continue at the level-4 page table.

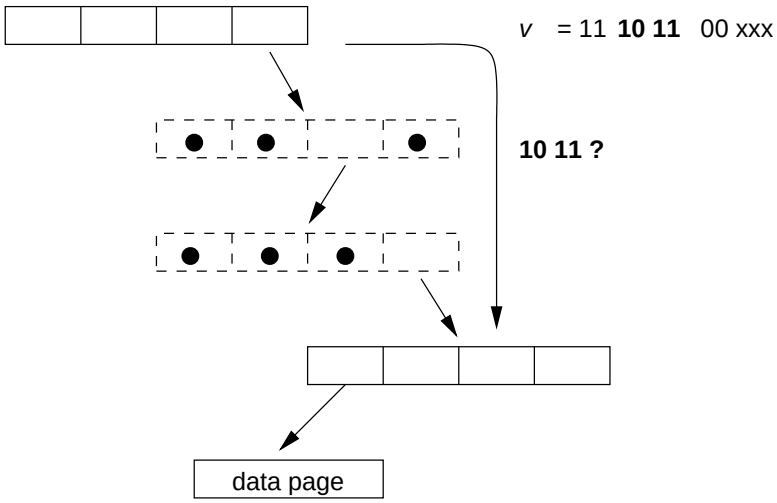


Figure 3.8: Guarded page table.

Therefore, each entry is augmented with a bit string g of variable length, which is referred to as a *guard*. This is the key idea of *guarded page tables*.

The translation process works as follows: upon each transformation step, a page table entry is selected by the highest part of the virtual address in the same way as in the conventional multilevel page-table method. The selected entry contains not only a pointer (and perhaps an access attribute), but also the guard g. If g is a prefix of the remaining virtual address, the translation process either continues with the remaining postfix, or terminates with the postfix as the page offset.

As an example, Figure 3.9 presents the transformation of 20 address bits by 3 page tables.

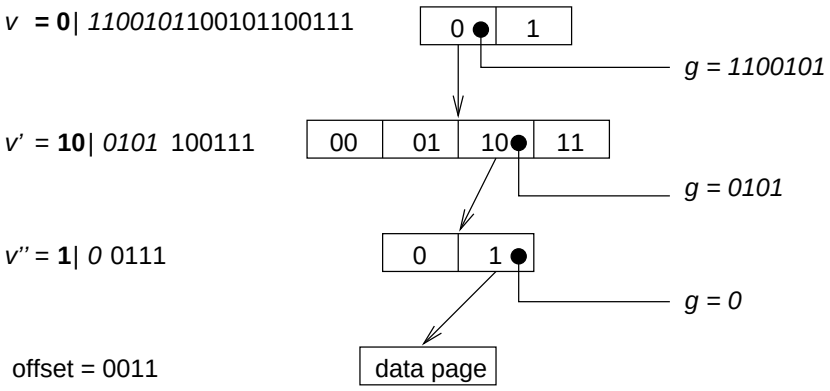


Figure 3.9: Guarded page table tree.

Note that the length of the guards may vary from entry to entry. Furthermore, page-table node sizes can be mixed; all powers of 2 are admissible. The same holds for data pages, i.e., a mixture of 2-, 4-, ... 1024-, ... entry page-table nodes and pages can be used.

Guarded page tables contain conventional page tables as a special case: if a guard has length zero, a translation step works exactly like in the conventional mechanism. However in all cases conventionally requiring a table with only one valid entry, a guard can be used instead. It can even replace a sequence of such single-entry page tables. This saves both memory capacity and transformation steps, i.e., guards act as a shortcut.

3.7.2 The Important Properties

Guarded page tables are tree based and feature advantages similar to MPTs, including: subtree sharing, hierarchical operations, and multiple page-size support. However, GPTs are much more general (and flexible) than MPTs. Any power-of-2 region can be represented by a subtree within the GPT. The subtree can be shared between independent GPTs, which provides efficient and simplified shared-memory support. Hierarchical operations, such as marking a region *read-only*, can be achieved by simply operating on the root of the subtree. A subtree can also be replaced entirely by a single region-sized page-table entry, or vice versa.

GPTs feature satisfactory worst-case memory consumption. Any address space with k mapped pages can be represented by a GPT containing $2(k-1)$ GPT entries [Lie96]. Such a GPT is termed *small*. For an MPT to perform similarly, it would have to be a binary tree with n levels (to support a 2^n-byte address space). With n levels, an MPT requires n translation steps to lookup a page-table entry, both in the best case and the worst case. In the best case, a GPT can perform a lookup in a single translation step. The worst-case number of translation steps depends on many factors. A detailed theoretical analysis of many scenarios can be found in [Lie96], but major points are outlined below.

- In the worst case, a lookup in a *small* GPT can be performed in $n/2$ translation steps.
- If 4 guards can be checked in parallel, a 4-associative GPT lookup can be achieved in $n/4$ steps, with a slight (14%) increase in worst-case page-table size.
- In the absence of parallel checking, and if worst-case page-table size is allowed to increase significantly, then a lookup can be performed in at most $\lceil n/m \rceil$ steps[2], where m is the minimum number of bits decoded in a translation step. The worst-case page-table size increases to $2^{m-1}k$, where k is the number of mapped pages.

Using GPTs to support a 64-bit address space appears to have positive and negative aspects. Worst-case memory consumption is good. A *small* GPT needs only $2(k-1)$ GPT entries, for k mapped pages. However, worst-case tree depth (translation steps) for the same *small* GPT is 32, a value likely to be prohibitively expensive if realised. Reducing tree depth via associativity in a software-loaded-TLB environment does not look feasible as GPT parsing is complex. Sequentially comparing 4 guards is likely to be expensive, even with some degree of parallelism afforded by a multi-issue CPU.

Allowing memory consumption to increase creates a trade-off between tree depth and memory consumption. Exploring this trade-off between space and time seems to be the most promising approach to supporting 64-bit address spaces with GPTs. Chapter 4 is devoted to a practical evaluation of GPTs, including exploring the trade-off between space and time.

[2] $\lceil x \rceil$ denotes x rounded up to the nearest integer.

Chapter 4

Guarded Page Table Evaluation

This chapter analyses the performance of several GPT variants. A comparison of GPTs with other page tables is postponed until Chapter 6.

Liedtke derived many theoretical results predicting the worst-case behaviour of GPTs; the most relevant results are summarised in Section 3.7. Liedtke's derivations focus on predictable and satisfactory worst-case behaviour. There has been little evaluation, or testing, of actual implementations under *real-world* conditions. It is unclear whether GPTs will perform near their predicted worst case, or near their potential best case. One of the goals of this chapter is to quantify typical GPT behaviour. The other goal is to gain an understanding of how various GPT design options affect GPT performance.

To achieve these goals, various implementations from the GPT design space were constructed (as described later in Section 4.1). To compare the different GPT implementations a set of benchmarks were developed. The benchmarks are mostly based on micro-benchmarking the L4 microkernel, rather than instrumenting the page-table manipulations themselves. The performance of the underlying page-table manipulations is readily visible at the micro-benchmark level, as L4 has an extremely low system call and IPC overhead [LES+97]. Micro-benchmarking also provides an indication of the performance an application might expect when using the microkernel primitives under test.

The micro-benchmarks test four facets of page-table performance:

1. **TLB-refill costs** represent the system overhead indirectly experienced by applications. These costs are independent of whether an application relies heavily on directly invoked system services or not. Minimising TLB-miss penalty

is becoming increasingly important as will be highlighted in Section 4.2.

2. Increasingly innovative address-space usage has brought **page-table size** into focus. Page-table size is an indirect memory overhead experienced by applications. Ideally this overhead should be low and independent of the address-space layout of the application. Section 4.3 examines this area in greater detail.

3. **Mapping and unmapping**[1] are the virtual-memory primitives of the L4 microkernel. These primitives form the basis of both operating-system personality and application-controlled virtual-memory management. The importance of the performance of these primitives is described further in Section 4.4.

4. The cost of **task creation and deletion** is influenced by setup and tear-down costs of kernel data. In a highly optimised microkernel, page tables represent a large proportion of kernel data used to manage a task. Section 4.5 explores the effect of the various page-table implementations on task creation and deletion performance.

4.1 GPT Implementation

GPT implementation is quite complex. The data structure used to represent a GPT aims to reduce the size of the page table, and the time it takes to parse or manipulate the tree. The exact layout of the structure is dependent on architecture-dependent optimisations. Previous work [LE95] describes how the representation used was derived. Relevant excerpts are included in Appendix B.

Figure 4.1 illustrates the representation used for a binary GPT. Nodes contain a pair of 64-bit guards (G_n) and 64-bit pointers (P_n). Leaf GPT entries point to non-GPT leaf nodes which contain two 32-bit page-table entries (PTE_n) and two 32-bit mapping-tree pointers (MTP_n). These shortened 32-bit entries rely on sign extension to form valid 64-bit quantities.

The leaf nodes store pairs of PTEs instead of storing a single PTE and mapping-tree pointer in the pointer field of leaf GPT entries. The justification for this is

[1] Note that the L4 **grant** virtual memory primitive is not included in these tests. It has no analogue in more conventional systems making it less relevant as a benchmark for comparison.

to accommodate the R4x00 TLB which contains and consequently loads pairs of page-table entries. It was determined experimentally that it is was more efficient to load pairs from specialised leaf nodes than it was to either load half a TLB entry at a time using a single PTE stored in the pointer field of the GPT, or load the TLB in pairs by finding the potentially non-existent adjacent PTEs on the fly. Loading a single page-table entry at a time effectively performs 2 TLB refills for each TLB miss. Loading pairs stored singly penalised applications by up to 4% of application runtime, and 1% on average (see Table A.5, in Appendix A). In no case did a GPT loading pairs stored singly perform faster than a GPT loading pairs stored in specialised leaf nodes.

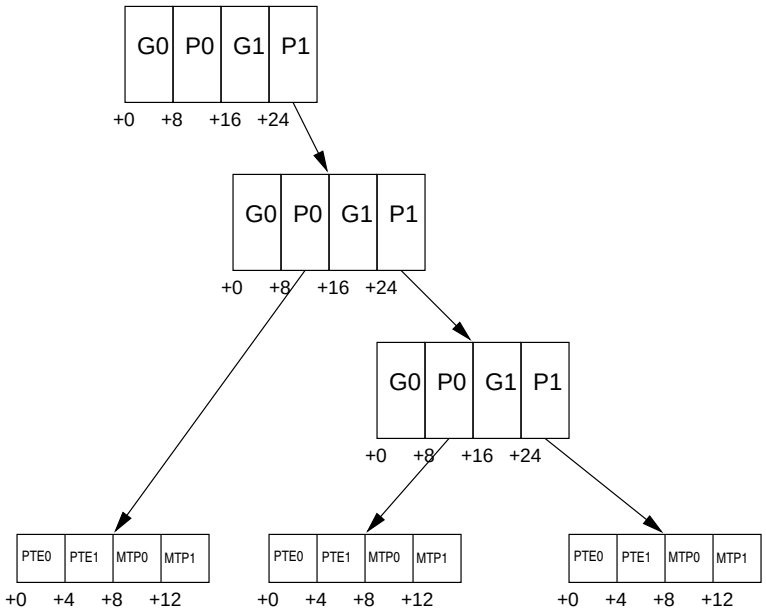


Figure 4.1: Diagrammatic representation of binary guarded page table.

GPTs are also flexible with regard to the size of nodes that form the GPT tree. Node sizes can be freely intermixed at any level of the tree. Larger node sizes reduce the tree depth at the possible expense of increased tree size if nodes are sparsely populated.

There are three basic policies for choosing node sizes in a GPT tree: using a single node size, using multiple node sizes, or using dynamically-variable node sizes.

The single node-size policy uses a single predetermined node size for all nodes

within the GPT tree. This has the advantage of simplicity. It removes the complexity of having to manage multiple-sized memory objects which is a low, but not negligible overhead [PT77, LB89]. It also avoids any complexity associated with choosing a node size. However, a single-node-size policy has the disadvantage of not adapting to the memory layout of the application, i.e. it could use more memory for a given tree depth than a scheme using a multiple size policy. Section 4.3.1 examines this issue more deeply, deriving the worst-case page-table size for a single-node-size policy.

A multiple-node-size policy is one that uses different node sizes with the restriction that once a node is created, it remains the same size for the life of the address space. Having multiple node sizes introduces the complexity of both managing memory in multiple sizes, and choosing an appropriate node size. Memory management is best handled by a buddy allocation scheme because nodes are always a power of two in size.

The choice of node size can be made in many ways. Node size can be based on the known structure of the application. This takes advantage of contiguous regions such as text, data, and stack segments, and other regions such as shared libraries. The choice of node size could also be based on hints from the application, or heuristics based on typical application behaviour.

A disadvantage of fixed node-size assignment is that the policy prevents the GPT from adapting to an application's changes in virtual-memory usage. There is the possibility for the operating system to monitor an application's behaviour and evolve the GPT structure with the application to ensure the best performance. This is the basic idea of the dynamically-variable node-size policy.

A dynamic node sizing policy could base sizing decisions on the same factors as the static policy. However, a dynamic policy is free to modify a node's size based on other policies. For example, one could apply the constraint of *all nodes in the tree must decode at least 8 bits in each translation step.* This constraint would be satisfied in a binary node, as long as both entries contain a guard which decodes at least 7 bits. Should a guard decode less than 7 bits (e.g. 6), the node could be expanded to a larger (4 way) node so that the constraint is still satisfied.

There disadvantages of dynamic node sizing are the extra complexity in choosing when to change node sizes, choosing which node size to change to, and the copying of entries from the original node to the new node of differing size.

While these multiple node size schemes have the potential to reduce tree depth and memory usage, they also complicate GPT management by adding buddy allocation and as-yet-unknown methods for node size selection and adaption. Rather than focusing on developing methodologies for multiple node sizing, I chose to evaluate single-node-size policies. The aim of this strategy is to evaluate the performance of such a policy and use the knowledge gained to identify areas where multiple-sized nodes would be most likely to be advantageous.

To evaluate how GPT node size affects performance, eight GPTs were implemented, each featuring a node sizes of S entries, where $S = \{2, 4, 8, \ldots, 256\}$. These implementations are referred to as G2, G4, G8, ..., G256 from here on in the text. The largest node size tested (G256) corresponds to using page-table nodes of 4K in size which is also the page size on the MIPS R4x00 architecture used as the test bed. G256 allows direct comparison to standard multilevel page tables at a later stage, as they would also use this node size. Given the likelihood of granularity being a problem under sparse conditions, it does not make sense to use larger sizes.

4.2 TLB Refill Performance

It has been estimated that memory sizes are quadrupling every three years [PH90], which is creating problems for both CPU architects and operating-system designers [Mog93]. One area receiving much attention is the TLB's coverage of physical memory. The TLB aims to hold as many relevant virtual-to-physical translations as possible to *cover* as much of physical memory as possible.

TLB coverage is not keeping pace with increasing physical memory sizes. In the past, TLB-miss handling typically contributed less than 5% [CE85] of overall runtime. However in recent studies, miss handling is not unknown to contribute 40% of application runtime [HH93].

Various methods have been proposed to combat increasing TLB miss ratios. Associativity trade-offs and changes [NUS+93, CLK97], micro-TLBs [CBJ92], variable page sizes [CBJ92, TKHP92, KTNW93, ROKB95], and subblocking [Tal95] have been examined and incremental improvements made in effective TLB coverage. TLBs have been removed altogether in some experimental systems [WEG+86, CSD86, JM97] which perform address translation in the cache; however, the tech-

niques have yet to appeared in a commercial microprocessor. There is little to suggest that minimising the TLB-refill penalty will be of lower importance in the future than it is now.

4.2.1 Benchmarks

Before introducing the benchmarks, it makes sense to describe the hardware platform used for testing throughout the thesis. The machine used was designed and built by the author for use as a teaching platform at the University of New South Wales. It features a 100 MHz R4700 [R4795] and a GT-64010A memory and PCI controller from Galileo Technology [Gal96]. The details of the machine relevant to benchmarking follow:

- The CPU features a 16K data cache and 16K instruction cache, both being two-way set associative. There is no second level cache in the machine.
- The machine has 64M of RAM on a 50MHz memory bus.
- The CPU has a 48-entry TLB with each entry holding two page-table entries. The TLB is reloaded by a specialised software exception handler.

Conventional Application Description

To examine performance of GPTs under traditional UNIX applications, a subset of the SPEC95 benchmarks [SPE95] were chosen. An initial selection was based on a previous examination of SPEC95's TLB behaviour [CLK97]. Applications exhibiting high TLB miss rates were chosen together with their ability to run with the skeletal UNIX emulation library. Also used were a number of applications exhibiting high TLB miss rates from the collection maintain by Al Aburto [Abu]. Applications exhibiting low TLB miss rates were not considered as their performance is independent of the underlying page-table structure. The applications are collectively termed the conventional applications throughout the rest of the thesis. The conventional applications are used in several benchmarks which are described later.

Table 4.1 lists each of the applications used, together with their size, type, and a brief description. The size refers to the amount of memory consumed (mapped)

by the application, not the span of the text, data, etc. The type indicates whether the application is an integer (I), or floating point (F) benchmark.

The applications used were all run unmodified from their original benchmark specifications, except for `gcc` and `mm`. Only a single file (`1ampt jp.i`) was compiled in the modified `gcc` benchmark instead of the usual several files compiled consecutively, as the skeletal UNIX emulation environment only allowed for a single invocation per benchmark run. The `mm` benchmark was run using only the "normal" algorithm as smarter algorithms feature lower TLB-miss ratios. The normal algorithm is a naive (inner product) matrix multiply.

	name	*size (Mb)*	*type*	*remarks*
SPEC	go	0.8	I	game of go
	swim	14.2	F	PDE solver
	gcc	9.3	I	GNU C compiler
	compress	34.9	I	file (un)compression
	apsi	2.2	F	PDE solver
	wave5	40.4	F	PDE solver
Alburto	c4	5.1	I	game of connect four
	nsieve	4.9	I	prime number generator
	heapsort	4.0	I	sorting large arrays
	mm	7.7	F	matrix multiply
	tfftdp	4.0	F	fast fourier transform

Table 4.1: Conventional applications from SPEC95 and Alburto suites. Size indicates amount of memory used at conclusion of application. Type "I" or "F" designates either integer or floating point, respectively.

Metrics

To examine the TLB-refill performance of different GPT implementations, the conventional applications were run while monitoring TLB-refill performance. Two metrics were chosen to evaluate the performance of GPTs with respect to each other. Elapsed application runtime is one metric, and average TLB-refill time is the other. Elapsed application runtime is used to quantify the effect TLB-refill has on each of the applications under test. Average TLB-refill time is used to obtain a comparative indication of TLB refill performance that is independent of the applications under test.

Application runtime is obtained by running the individual UNIX applications together with a skeletal support library directly on the L4 kernel. The L4 kernel was a "clean" kernel, i.e. the kernel contained no instrumentation. Applications are contained completely in physical memory, hence there is no disk I/O component to the elapsed time.

Average TLB-refill time is obtained by instrumenting the TLB-refill handler to count cycles spent inside the handler, and the number of TLB refills. The TLB-refill instrumentation code was carefully constructed as follows:

- The code counts cycles spent in the TLB-refill routine using the CPU count register which ticks constantly at half the internal clock rate.
- The code avoids, as much as possible, affecting cache misses (or hits) of the refill routine. For example, memory used for instrumentation is aligned to appear in different cache-lines to the memory used by the refill routine as temporary storage.
- The code does not count refill-exception overhead, i.e. the instrumentation does not record the time taken between the initiation of a TLB-miss and the execution of the TLB-refill routine, and the time between the exit of the TLB-refill and the resumption of the faulting instruction. However, the uncounted component is constant for different page-table implementations, and thus irrelevant for comparison purposes.

Thus the overhead of the refill-routine instrumentation should not significantly affect results and is equivalent for each page-table configuration tested. This can be confirmed by considering the following equality,

$$t_{elapsed} = t_{tlb} + t_{app} + t_{var}$$

where the elapsed runtime of an application ($t_{elapsed}$) is equal to the sum of contributions from TLB-refill handling (t_{tlb}), the application itself (t_{app}), and some varying time (t_{var}) which includes TLB-refill instrumentation and cache artifacts. For each combination of application and page table, $t_{elapsed}$ and t_{tlb} is known accurately (see Appendix A for the raw results). Table 4.2 shows the average and normalised standard deviation of $t_{var} + t_{app}$ for each application, averaged across all runs of the eight page tables.

BENCH	Mean	STD (%)
go	947.1	0.7
swim	2121.1	0.0
gcc	43.3	0.3
compress	875.7	0.4
apsi	1794.6	0.5
wave5	3024.5	0.1
c4	34.3	2.5
nsieve	150.0	0.2
heapsort	29.1	0.1
mm	54.2	3.1
tfftdp	11.2	0.8

Table 4.2: Mean (in seconds) and normalised standard deviation of $t_{var} + t_{app}$ for each application, averaged across all GPTs.

Given that t_{app} is constant for each application, the standard deviation of each of the times (in Table 4.2) is due to variation in t_{var}. Making the pessimistic assumption that t_{var} is solely due to TLB instrumentation, we can conclude that the effect of TLB instrumentation is either negligible, or at least an equal contribution to most benchmarks. Both `mm` and `c4` feature higher standard deviations than the other benchmarks as, for some scenarios, TLB-refill overhead is over 60% of application runtime. However, this variation is not enough to significantly affect later results.

4.2.2 Results

The results of the TLB-refill performance experiment are summarised in two ways. Table 4.3 compares elapsed times of each combination of application and page table. Results are normalised to the elapsed time of G2 to allow comparison between applications as well as between page tables. Figure 4.2 shows TLB-refill time averaged across all benchmarks for each page table. The raw results that these summaries are based on are contained in Appendix A.1.

With reference to Figure 4.2, the average refill time reduces as the GPT node-size increases. This corresponds to the reduction in tree depth as node size increases, which is illustrated in Figure 4.3. Initially large improvements in average

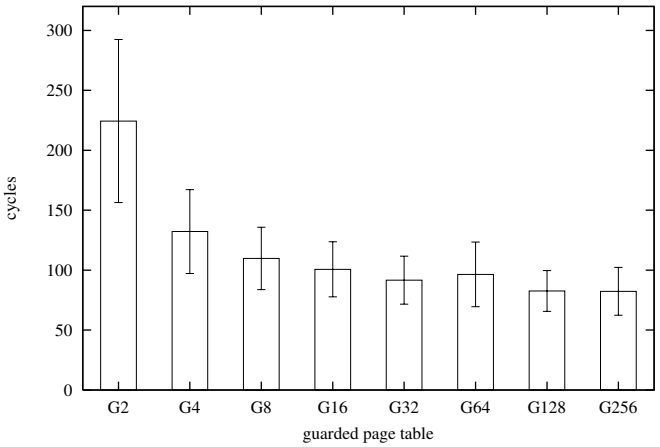


Figure 4.2: Average refill time across conventional benchmarks for each page table. Error bars represent standard deviations.

refill time are made in moving from G2 to G4 and then to G8. The move from G8 to G16 and beyond results in smaller incremental improvements as a result of the smaller reductions in tree depth.

The deviation from the trend at G64 is partially explained by the page table having the same average depth as G32 which results in similar performance of the applications with the exception of `go`. With reference to Table A.1, the application `go` contributes disproportionately high average-refill costs which I believe, though it is difficult to confirm, is due to a pathological page-table layout resulting in a larger number of cache misses during TLB refill.

Table 4.3 illustrates the effect the TLB-refill cost has on application runtime. The runtime of the majority of applications behaves in a manor consistent with the average TLB-refill costs discussed above. More precisely, the runtime improves significantly in moving from G2 to G4 (11% on average), and then to G8 (a further 3%). There are small incremental improvements (1% or less) in moving to G16 and onwards. Some applications with lower miss ratios (`apsi`, `swim`, `heapsort`, `go`) illustrate this to a lesser extent, while `mm` with a high miss ratio continues to improve its performance with any reduction in TLB-refill cost.

Figure 4.4 illustrates elapsed-time results graphically. The figure is intended to show the trends exhibited by the applications for each page table. It is not intended to show trends for any application in particular, and as such, it is unlabelled and

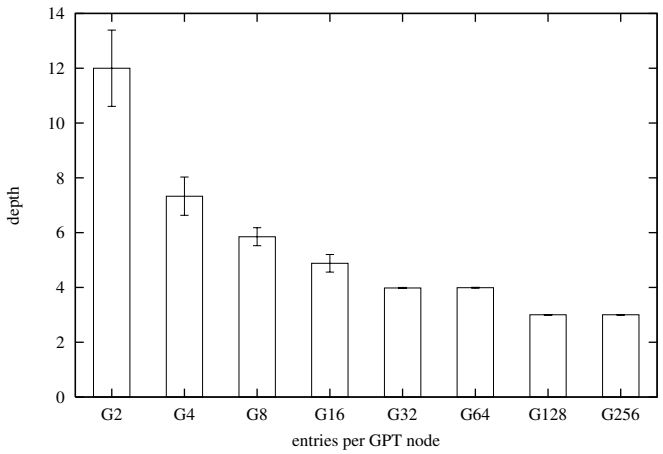


Figure 4.3: Average GPT tree depth averaged across all the conventional applications. Error bars are standard deviations.

somewhat cluttered.

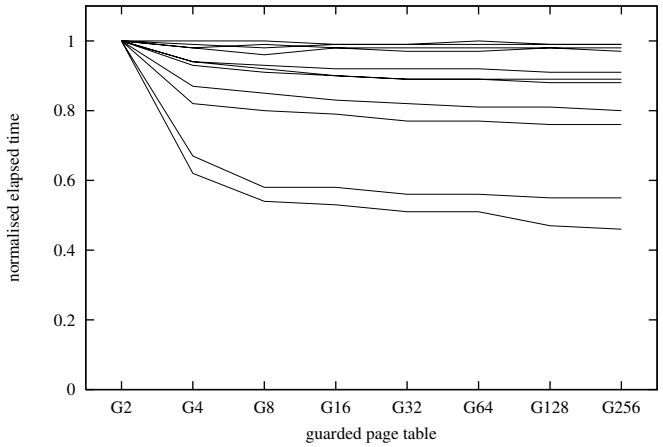


Figure 4.4: Plot of elapsed times normalised to G2 for each combination of application and guarded page table.

4.3 Page Table Size

Page-table size is an issue for several reasons. A small compact page table is more likely to to retain a greater portion of itself in the cache, which improves lookup

BENCH	G2	G4	G8	G16	G32	G64	G128	G256
go	1.00	0.99	0.98	0.99	0.99	1.00	0.99	0.99
swim	1.00	1.00	1.00	0.99	0.99	0.99	0.99	0.99
gcc	1.00	0.82	0.80	0.79	0.77	0.77	0.76	0.76
compress	1.00	0.94	0.92	0.90	0.89	0.89	0.88	0.88
apsi	1.00	0.98	0.96	0.98	0.97	0.97	0.98	0.97
wave5	1.00	0.93	0.91	0.90	0.89	0.89	0.89	0.89
c4	1.00	0.67	0.58	0.58	0.56	0.56	0.55	0.55
nsieve	1.00	0.94	0.93	0.92	0.92	0.92	0.91	0.91
heapsort	1.00	0.98	0.99	0.98	0.98	0.98	0.98	0.98
mm	1.00	0.62	0.54	0.53	0.51	0.51	0.47	0.46
tfftdp	1.00	0.87	0.85	0.83	0.82	0.81	0.81	0.80
Average	1.00	0.89	0.86	0.85	0.84	0.85	0.84	0.83

Table 4.3: Elapsed times normalised to G2 for each combination of application and page table.

speed. A compact page table also displaces less application data from the cache, which reduces the indirect overhead to applications. Displaced application data is a significant cost of VM to applications [JM98a]. A small overall page-table size is also important to conserve physical memory consumption. A study of online transaction processing reported a 17% improvement in transaction throughput directly attributable to an increase in available physical memory which was achieved via a reduction in page-table size [YR93]. Task creation and deletion costs are directly influenced by page-table size. Minimising page-table size reduces the cost of managing the page table itself. This area is covered in greater detail in Section 4.5.

To compare sizes of the different page tables under various conditions, a metric independent of allocated address-space size is chosen. An independent metric allows the comparison of page-table memory consumption among different applications with the same page-table structure. A metric dependent on application size would only allow comparison between different page tables under the same benchmark conditions. The metric chosen was average number of page-table bytes used per mapped page.

The page-table size examination looks at each page table's size behaviour from three perspectives.

1. The theoretical **worst case**. I derive a formula to predict the upper bound of page-table memory consumption given the single node size.

2. The page-table memory consumption as seen under **conventional applications**. Conventional applications are run and monitored to examine the memory overhead that one might expect in a normal system.

3. The page-table memory overhead that might be observed in **sparse address spaces**. Two types of sparse address space are synthesised, one consisting of single pages, and another consisting of objects spanning several consecutive pages. Each synthesised address space is analysed to compare page-table memory consumption with that predicted by worst-case analysis, and that observed under conventional applications

4.3.1 Worst-Case Analysis

To facilitate comparison with later results, I first derive worst-case page-table memory overhead for GPTs with a single node size.

Lemma 1 *All nodes in a guarded page table contain at least two non-nil entries.*

> *Proof*: Given an arbitrary guarded page table representing a particular translation, we colour all nodes in the tree that contain all nil entries red, and colour all nodes containing one non-nil entry blue. All other nodes containing two or more non-nil entries are left uncoloured.
>
> We first eliminate all red nodes as they have no effect on the set of valid translations. We then eliminate all blue nodes by concatenating guards with the node above which also has no effect on the set of valid translations.
>
> This results in an uncoloured tree, i.e. a tree with nodes containing two or more non-nil entries.
>
> *q.e.d*

Theorem 1 *A guarded page table containing k leaf entries with nodes of size S entries, contains at most $S(k-1)$ entries.*

Proof: A guarded page table will contain the most entries when each node contains the minimal number of valid entries. From Lemma 1, each node in a GPT will minimally contain two non-nil entries. Thus one can think of a GPT being a binary tree containing two entries and unused entries in each node. A binary tree with k leaf nodes contains $k-1$ internal nodes [Kin90, page 83].

If internal nodes are of size S entries, k leaf entries are represented by $S(k-1)$ entries in the GPT.

q.e.d

Thus we need $\frac{S(k-1)+k}{k}$ PTEs per mapped page, i.e. approximately $S+1$ PTEs for large k. The implemented GPTs have 16-byte entries, which results a memory overhead (normalised to per mapped page) of $16(S+1)$. Table 4.4 shows the worst-case memory consumption per mapped page, for each of the GPT implementations.

GPT	worst-case
G2	48
G4	80
G8	144
G16	272
G32	528
G64	1040
G128	2064
G256	4112

Table 4.4: Worst-case memory consumption per mapped page (in bytes), for each of the GPT implementations.

4.3.2 Benchmarks

Conventional Applications

While predictable and satisfactory worst-case page-table size is an important property, it becomes less so if normal system behaviour never approaches the bounds of worst-case behaviour. The examination of page-table size under conventional applications should provide insights as to what is typical behaviour, and how close typical behaviour is to worst-case behaviour.

The experiment is conducted by running each application in turn until completion. On completion of each application, the page table is analysed to gather statistics on memory consumed, average depth, and number of leaf, null, and internal entries. The memory consumed includes both the memory consumed by the GPT and the memory consumed by the external leaf nodes. Leaf entries are entries in the page table that point to the external page-table pair. Internal entries are GPT entries that point to another GPT node. Null entries are entries in the GPT that are neither leaf nor internal. Average depth of the GPT is calculated by averaging the depth of each leaf entry. For a more detailed description of the statistics gathered, and the raw results, see Appendix A. The applications chosen for the experiment are the same applications described earlier in Section 4.2.1.

Sparse Address Spaces

The conventional applications previously described can be considered a gentle test of page-table size behaviour. The conventional applications consist of a contiguous region containing the *text-, data-,* and *heap* sections, and another contiguous region of *stack*; both regions grow towards each other.

This typical address-space layout is not necessarily the normal case in newer systems. Modern systems have a more fragmented address space, consisting of shared libraries, shared memory, memory-mapped files, and memory objects backed by external pagers.

Shared libraries are used to reduce both disk and memory consumption, and allow for application library replacement without the need for relinking all user programs. Shared libraries could be statically linked to fixed addresses in all processes with each library having its own region in the address space [Arn86]. However, modern shared libraries are dynamically linked into an applications address space reducing fragmentation of the address space. See [HCL95] for a list of references for shared libraries.

Modern operating systems allow user processes share regions of virtual memory with other processes. These regions can provide for fine grain multiprocessing, fast IPC, and database support. Typical UNIX systems support shared memory via the BSD `mmap` system call [MKL+94]. `mmap` also allows applications to map a file into their virtual address space. Memory-mapped files reduce the cost of file I/O by avoiding copying between the buffer cache and the application, and also allows

applications sharing files to share the same physical image.

External pagers, such as in Mach [RTY+88], allow user processes to control the content of memory objects mapped into the address space of other user processes. Pagers are free to implement the functionality they choose for the memory objects they control. They can supply zero-filled memory, persistent memory [ABC+83], memory-mapped files, etc. A memory object could be empty, completely mapped, or sparsely populated.

Shared memory, memory-mapped files, and memory objects backed by external pagers all increase the number of distinct regions used in an address space. This creates a fragmented address space consisting of many regions of varying population density. A data structure used to store page-table entries must be able to efficiently support such an address-space layout.

Moreover, single-address-space systems [CFL93, HEV+98, MWO+93] expect to have extremely sparse address spaces. Single-address-space systems store all data in a single large virtual address space. An application's currently active mappings consist of a selection of memory objects scattered throughout the address space. Ideally, a page table should efficiently support such a scenario with minimal memory overhead.

To gain an understanding of GPT behaviour in sparse address spaces, we constructed two synthetic benchmarks. The benchmarks are synthetic in the sense that they do no real work other than generate an address-space layout. The two benchmarks are described below.

sparse page This benchmark randomly allocates single 4KB pages in a 40-bit (1 terabyte) address space. This is a tough benchmark, as such an address-space layout could be considered pathological in most real systems.

The number of pages allocated is varied from 64 (256KB) to 8192 (32MB) to expose any dependency on the amount of memory consumed. Each individual size is run separately 10 times, each run using a different seed to start the random number generator. The random number generator is based on a lagged Fibonacci series generator [Pet].

sparse file To model a more realistic situation, this benchmark allocates randomly located, within the 1TB address space, randomly sized objects from a file-size distribution. This model is aimed at approximating what might happen in

a single-address-space operating system where "files" are objects in memory.

The file distribution used is shown in Figure 4.5. It is from a local study [Elp93] and is similar to distributions from other file system studies [Sat81, OCH+85, BHK+91].

Like the SPARSE-PAGE benchmark, the number of objects allocated is varied from 64 to 2048 in powers of 2, each number of objects is run 100 times with a different random number generator seed each time. For each set of benchmark runs with a constant number of objects, the actual number of pages allocated in each distinct run varies because randomly sized files are used. Table 4.5 tabulates the average number of pages allocated (and normalised standard deviation) for all runs allocating the number of objects indicated.

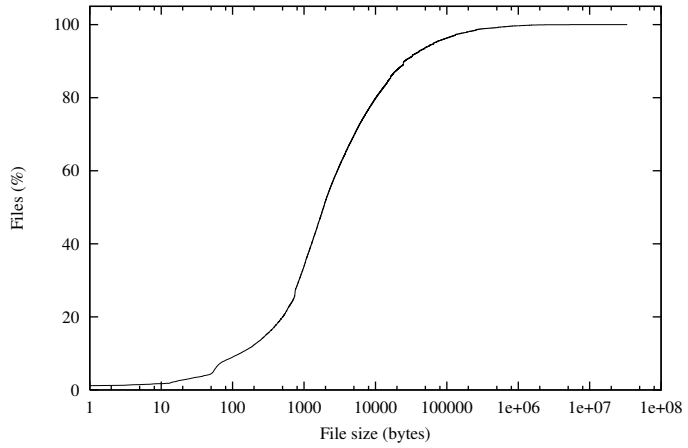


Figure 4.5: Cumulative distribution of object size used in SPARSE-FILE benchmark.

4.3.3 Results

Results for the conventional applications are presented in Figure 4.6 and Table 4.6. The results have been normalised to represent the memory overhead per mapped page, i.e. number of bytes of page table divided by the number of mapped pages in the address space.

Figure 4.6 displays the memory overhead for each page-table averaged across all applications, together with the lowest and highest observed overhead. The fig-

Objects	Pages	NSTD—
64	505	2
128	989	1
256	1867	0.9
512	3836	0.6
1024	7969	0.5
2048	15826	0.3

Table 4.5: The average number of pages allocated (Pages) and normalised standard deviation (NSTD), for all runs allocating the number of objects indicated (Objects).

ure reveals that initially for G2 approximately 24 bytes per page is required, with similar low and high values. In moving from G2 to G4 the overhead is reduced to approximately 19 bytes per page. Moving to G8 and then G16 results in small incremental improvements. This trend is due to the reduction in tree depth and hence a reduction in the number of internal tree nodes which do not contribute directly to the number of mapped pages.

In moving from G16 to G32 and onwards, the average memory overhead increases. This trend is explained by the increase in the number of empty GPT entries in the larger GPT nodes. The higher node sizes are also more susceptible to exhibiting high overheads due to idiosyncratic address-space layouts, as illustrated by the highest overhead diverging significantly from the average for G256.

Referring to Table 4.6 and Figure 4.7, one can see the majority of the applications behave similarly and follow the trend of the mean described previously. However two applications (`apsi` and `go`) only follow the trend from G2 to G16 and then deviate significantly at higher node sizes due to their sparse address-space usage.

Figures 4.8 and 4.9 summarise the results for the SPARSE-PAGE benchmark. The average normalised space overhead (over 10 runs) for each combination of the number of mapped pages and page table is plotted for each page table. Error bars represent standard deviation over the 10 runs.

As each of the individual runs of the same benchmark uses a different randomly generated address space, the standard deviation can be viewed as an indicator of page-table sensitivity to address-space layout. The small node size page tables vary very little in overhead, as do all page tables when the number of pages increases.

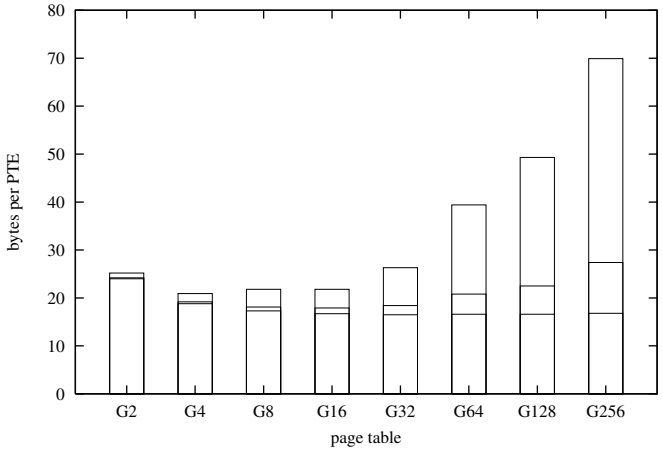


Figure 4.6: Page table bytes per mapped page for conventional applications. The three values represent the highest, average, and lowest page table memory overhead observed in the conventional applications.

BENCH	G2	G4	G8	G16	G32	G64	G128	G256
go	25.2	20.9	21.8	21.8	26.3	39.4	49.3	69.9
swim	24.0	18.8	17.3	16.8	16.6	17.1	17.1	18.2
gcc	24.1	18.9	17.6	17.1	17.0	17.9	18.3	20.1
compress	24.0	18.8	17.3	16.8	16.7	16.9	16.7	17.2
apsi	24.5	20.1	19.2	20.3	22.0	26.8	30.3	45.2
wave5	24.1	18.8	17.3	16.7	16.5	16.6	16.6	16.8
c4	24.1	19.0	17.8	17.4	17.5	19.2	20.7	23.9
nsieve	24.1	18.9	17.7	17.2	17.1	18.8	19.6	21.3
heapsort	24.1	19.0	17.8	17.5	17.7	19.3	20.3	24.4
mm	24.1	18.9	17.6	17.2	17.2	18.0	18.5	20.6
tfftdp	24.1	19.0	17.7	17.6	17.5	19.0	20.0	24.0
Average	24.2	19.2	18.1	17.9	18.4	20.8	22.5	27.4

Table 4.6: Normalised page-table memory overhead for each combination of application and page table.

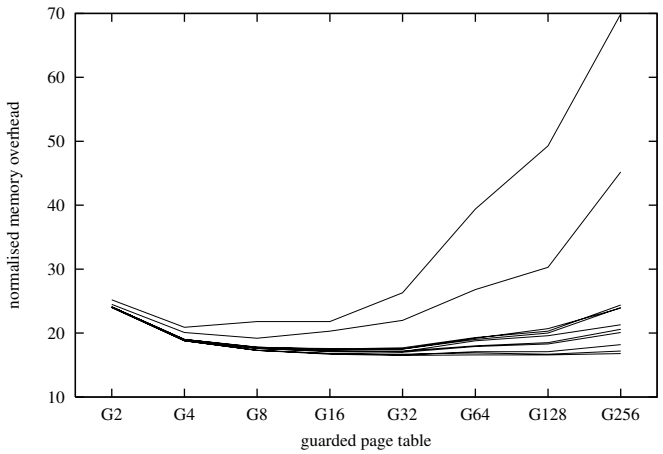


Figure 4.7: Normalised page-table memory overhead for each combination of application and page table.

The situation displaying the highest variability is when page tables with high node sizes combine with small numbers of pages. In this situation, standard deviations are approximately 10% of the mean. Thus GPTs are reasonably insensitive to address space layout, especially GPTs with small nodes sizes.

The two figures also illustrate that for small node sizes, address-space overhead is independent of the number of pages mapped. However, for medium and large node sizes there is an interesting dependence on the number of pages mapped that is not entirely obvious. The dependence can be explained in terms of the *spill-over* effect which is described in detail in Section 4.3.3. *Spill-over* affects many of the presented results. Consequently, a brief diversion to describe *spill-over* is warranted.

The *Spill-Over* Effect

The peaks in page-table overhead for the SPARSE-PAGE benchmark (Figure 4.8 and 4.9) can be explained as follows.

Firstly, assume we have a GPT that contains the page-table entries for a fully populated region of equal size to the region under test in the SPARSE-PAGE benchmark (1 terabyte). This GPT would have a single long guard from the root of the page table to the node of the page table which forms the root of page-table tree spanning the 1-terabyte address-space region.

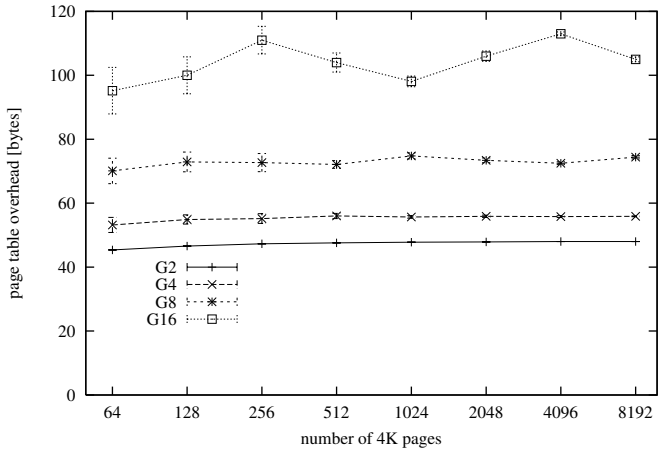


Figure 4.8: Normalised page table space overhead for SPARSE-PAGE benchmark for page table G2 – G16.

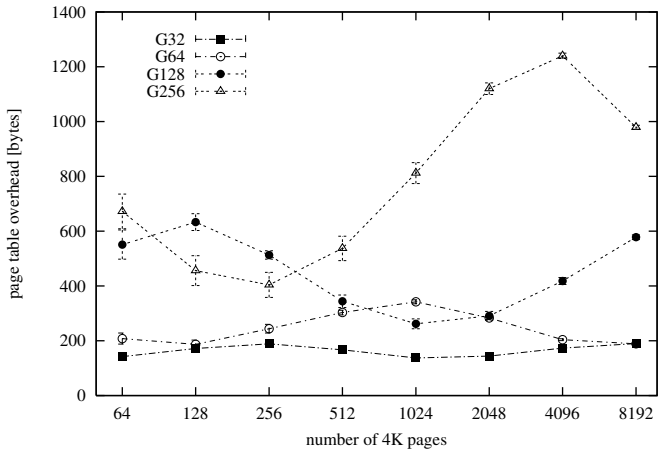


Figure 4.9: Normalised page table space overhead for SPARSE-PAGE benchmark for page table G32 – G256. Note the much larger y-axis scale in comparison to Figure 4.8.

The node that forms the root of the address-space region can either be partially or fully utilised. More precisely, if the node is partially utilised, then only some of the entries in the node are required to point to subtrees that map the region. Full utilisation indicates that all the entries in the node are needed.

The number of entries used in the region root node can be statically determined if one knows that node sizes are fixed, the size of the nodes, and the page size. This is achieved by examining the number of bits required to decode the address space. Given a one terabyte (40-bit) region with 8K (13-bit) page pairs, one needs to decode 27 bits from the top of the tree to index a pair of page-table entries. The number of bits needing decoding in the upper node is $R = 27 \bmod S^{'}$ where $S^{'}$ is the number of bits decoded in a node, i.e. $S^{'} = \log_2 S$ where S is the number of entries in a node. Assume for simplicity sake that we are dealing with 8K pages, rather than pairs of 4K pages.

For example, $R = 0$ for 8-entry nodes and thus the region root node is fully utilised. $R = 3$ for 16-entry nodes, hence the region root node is partially utilised, using 8 entries to cover the 1 terabyte address-space region.

Now assume that a 1 terabyte region is populated such that the region root node is allocated first, successively followed by nodes closest to the region root. A distribution of equally spaced nodes within the region will achieve this. For example, if the region root node has 8 entries, 8 equally spaced pages will ensure each entry is used. 16 equally spaced pages will ensure the region root node is full, and that the next level of nodes below the root is allocated and contains 2 entries in each node.

For each node size (G2 – G256), we can calculate how many pages can be mapped for a given number of levels in the tree. To elaborate, if nodes contain 4 entries (G4), a 1-level tree can map 4 pages, two levels can map 16 pages, etc. Now examining the case of populating the tree from the region root node down, a G16 GPT can map 8 ($2^{27 \bmod 4}$) pages within the 1 terabyte address space using the region root node alone. Adding a complete level to the tree below the region root enables the tree to map $16 \times 2^{27 \bmod 4} = 128$ pages, another level gives $16 \times 16 \times 2^{27 \bmod 4} = 2048$ pages, etc.

Generalising this to a GS GPT, we can calculate the maximum number of pages that can be mapped (N) in a GPT as folows:

$$N = S^n 2^{(A^{'} - P^{'}) \bmod S^{'}} \tag{4.1}$$

where A' is $\log_2$ of the address space size (40 in our case), P' is $\log_2$ of the effective page size (13), n is the number of levels below the root node. By applying this equation to G16 – G256 we can calculate N for each of the page tables for a 1-, 2-, and 3-level tree. The results are tabulated in Table 4.7. One level corresponds to having only the region root node.

Level $(n+1)$	G16	G32	G64	G128	G256
1	8	4	8	64	8
2	128	128	512	8192	2048
3	2048	4096	32768	1048576	524288

Table 4.7: Maximum number mapped pages for a given number of levels in the tree.

Each entry in the table corresponds to a completely filled tree, and thus is indicative of when each GPT has minimal normalised memory overhead. The situation that produces maximum memory overhead occurs when one doubles the number of mapped pages indicated in the table. Doing this requires adding a complete extra level of nodes in the tree, each node in the new level populated by only 2 entries.

In a GS GPT which is used to map equally spaced pages, the points of highest memory overhead occur when N page pairs are mapped, N being predicted as follows:

$$N = 2^{nS' + 1 + ((A' - P') \bmod S')} \tag{4.2}$$

for $n \in \{0, 1, \ldots\}$. Noting that n, the number of levels below the root node in the tree, is limited by the number of possible levels in tree.

Figure 4.10 is the result of repeating the SPARSE-PAGE benchmark (Section 4.3.2) with equally-spaced pages instead of using a uniform random distribution. Note that in a sparse environment, allocating a single page effectively corresponds to allocating a page pair. The figure illustrates the *spill-over* effect predicted for the larger node-size GPTs. Equation 4.2 correctly predicts the number of pages at which each peak occurs.

This expression can be used to explain the results of the SPARSE-PAGE benchmark. A property of the uniform random number generator used in the benchmark is that (obviously) it spreads out allocated pages uniformly in the address space. This approximates, given enough samples, the situation described above where the

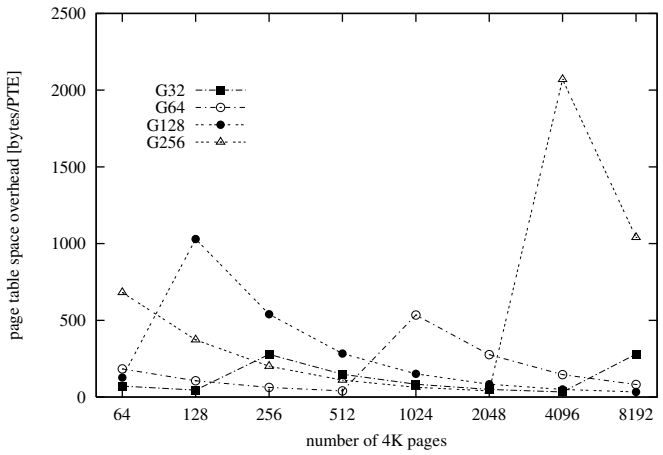


Figure 4.10: Page table space overhead for various numbers of equally spaced 4K pages in a 1 Terabyte address space for each page table.

tree is populated from the root node to the leaves. Thus the *spill-over* effect is visible when using the uniform random distribution, and points of highest overhead are predicted using the above expression.

Results Continued

Figures 4.11 and 4.12 summarise the results for the SPARSE-FILE benchmark. The average normalised memory overhead (over 100 runs) is presented for each page table and object number combination. Error bars represent standard deviations.

Like the SPARSE-PAGE case, the SPARSE-FILE results show the low and stable memory overhead of the small node-size GPTs. The larger node sizes exhibit much higher and less consistent overhead.

The trends in the curves can be explained in terms of two effects. The general trend, of average overhead reducing slightly as the number of objects increases, is due to large objects swaying the average overhead. The greater the number of objects allocated, the higher the probability of a sample object set containing large efficiently stored objects. The *spill-over* effect is superimposed on the trend of gradual reduction in memory overhead. The previous SPARSE-PAGE results show the *spill-over* effect in action for single page-size objects. An object in SPARSE-FILE has a root node topping the GPT branch containing the object's page-table entries. From the perspective of *spill-over,* a pointer to a root node is the same

as a pointer to a page-table entry. Hence for the SPARSE-FILE benchmark, the illustrated dependence of memory overhead on number of objects corresponds to the SPARSE-PAGE benchmark's dependence on number of pages. For example, there is a deviation in G256 upwards between 256 and 2048 allocated objects, and a downward deviation for G128 between 256 and 1024 allocated objects.

As in the previous benchmarks, the standard deviation shows the general stability of the smaller node sizes, and the high variability of the larger node sizes.

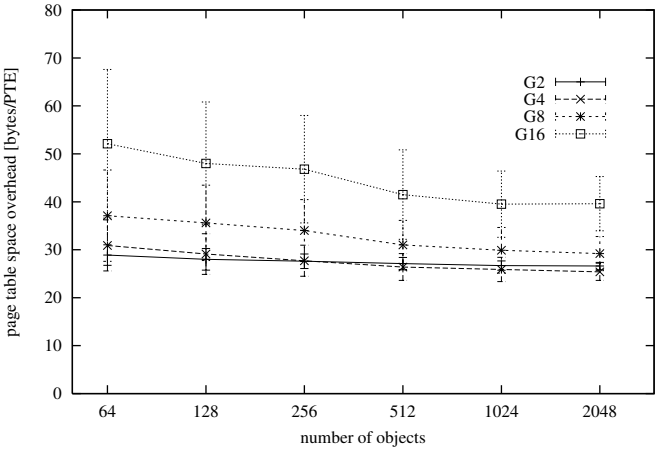


Figure 4.11: Normalised page table space overhead for SPARSE-FILE benchmark for page table G2 – G16.

To compare and give perspective to the results thus far, Figure 4.13 illustrates the results from all three benchmarks and the theoretical worst case. More specifically, it plots the highest observed overhead for conventional application benchmarks, the average overhead for the SPARSE-PAGE benchmark, the average overhead for the SPARSE-FILE benchmark, and the theoretical worst case.

The most noticeable, and somewhat unexpected result, is the SPARSE-PAGE benchmark's result being well below that predicted as being the theoretical worst case. SPARSE-PAGE is a tough benchmark unlikely to occur in practice, hence the theoretical worst case is also unlikely to occur unless a systematic attack on the page table is made.

Comparing the SPARSE-PAGE result with the SPARSE-FILE result, the clustering of pages into objects in the SPARSE-FILE benchmark approximately halves the overhead per PTE. This is partially explained by the page table storing PTEs in

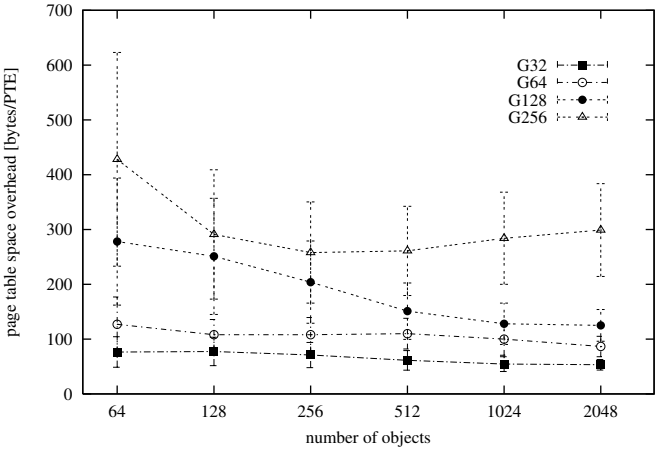


Figure 4.12: Normalised page table space overhead for SPARSE-FILE benchmark for page table G32 – G256.

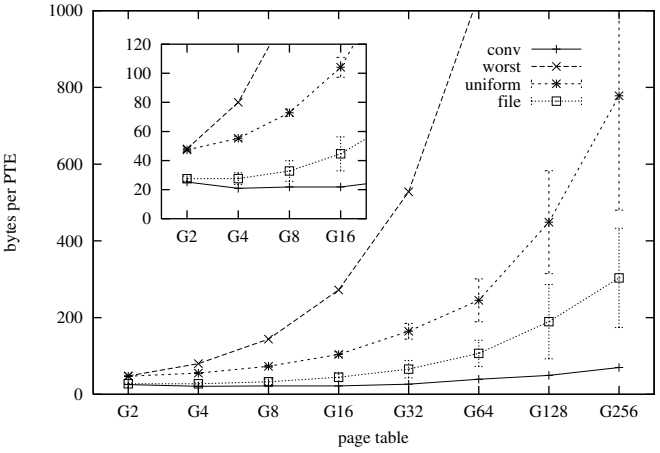


Figure 4.13: Page table bytes per PTE for conventional applications (highest observed), worst case, SPARSE-PAGE, and SPARSE-FILE benchmarks.

the leaves as pairs for efficient TLB refill. Each PTE pair, when both valid, halves the leaf's contribution to per-PTE page-table size and halves the number of leaves when compared to leaf entries containing only a single valid PTE. Noting that in the size distribution used, 66% of objects are single pages; hence, the rest of the reduction can be explained by the occasional larger files (3% $>$ 128KB) being stored with efficiency comparable to the contiguous conventional applications. A large object stored efficiently counteracts, to some extent, the effect of many single page objects stored inefficiently.

The SPARSE-FILE benchmark, while being "easier" on the page table than SPARSE-PAGE, is still a significantly more difficult benchmark than the conventional applications tested. This is shown by further size improvements when compared to the highest observed overheads for the conventional applications. I believe it is not unwise to consider the results for SPARSE-FILE to be the practical worst-case overhead for medium to large applications, with the reservation that a larger sample of applications is needed for it to taken as a serious estimate. Also, smaller applications are more likely to have higher overheads (e.g., having only a few separated pages). However page-table size is not an issue for small applications.

4.4 Kernel (Un)Mapping Performance

Mapping and unmapping are powerful and flexible primitives in the L4 microkernel. Not only do they allow designers of operating-system personalities to implement performance optimisations such as copy-on-write [FR86], they also allow applications the same degree of control to manipulate their own virtual memory. Appel and Li [AL91] point out that efficient implementation of virtual-memory primitives is crucial to performance of some applications using virtual memory tricks.

4.4.1 Benchmarks

To examine how different GPT implementations affect mapping performance, two benchmarks based on those used by Appel and Li were constructed. The first benchmark tests the performance of the combination of taking a protection fault, mapping, and unmapping virtual memory pages. All the operations are performed

on single pages at a time. The test consists of a client task and a server task acting as the client's pager.

The first benchmark, from here on referred to as MAP1, is described below:

- In the client, access a random page in a 16 megabyte region generating a protection-fault IPC to the pager.
- In the pager we select some other mapped page and protect it (mark it read-only via `l4_fpage_unmap`), and unprotect the faulting page (i.e. map it read-write via mapping IPC).

This is repeated 4096 (16M/4K) times to form the benchmark. The benchmark itself is repeated 10 times to get a statistical sample. The random number generator is the same as previously used, a different seed is used for each benchmark run.

The second benchmark is similar to the first except that it protects (makes read-only) all pages as a region, instead of protecting a single page at time. The benchmark is used to compare the performance of protecting (unmapping) memory in a batch rather than as individual pages. The benchmark, from here on referred to as MAPN, works as follows:

- Protect all pages in a 16 megabyte page region using a single unmap.
- In the client, access each page in random sequence.
- In the pager, unprotect the faulting page via mapping IPC.

Like before, the benchmark is repeated 10 times to get a statistical sample.

In addition to the two benchmarks based on the Appel and Li benchmarks, a third benchmark was designed to evaluate a GPT's ability to perform operations on sparsely populated regions. A trait of several algorithms using virtual memory is that they protect regions in large batches, and unprotect them a page at time. Such algorithms include concurrent garbage collection, concurrent checkpointing (both described in [AL91]), and exclusivity based virtual-memory mechanisms [Lie94]. These algorithms protect a logical entity which is a region of memory or an entire address space. However, the physical entity corresponding to the logical entity may be a subset of the it. The logical entity may be a partially utilised address space, a fragmented memory region, or only partially present if the region is paged to external storage. Performing operations on the logical entity is desirable as it

presents a "clean" abstraction to applications, and may also be necessary in the case where applications are unaware of the underlying physical state of the logical entity (for example, if the region is demand paged).

The third benchmark, from here on referred to as MAPS, is described below:

- 16 pages in a region of an aligned XKB in size are randomly accessed, after which
- the region is protected with a single unmap operation.

The time taken to perform the unmap is the metric used, the size X is varied between 64KB and 16MB in powers of 4. Thirty runs are made at each size to get a statistical sample, each run with a different random number generator seed.

4.4.2 Results

Figure 4.14 shows the results for the MAP1 and MAPN benchmarks. The results are normalised to the per-page cost, more precisely each of the benchmarks protects and unprotects 4096 pages, the illustrated cost is the elapsed time divided by 4096. Note that each benchmark was run 10 times, the standard deviations from the mean where negligible.

The results reveal a general trend of reduced mapping cost as GPT node size increases. This can be attributed to the reduced tree depth of larger node sizes, and consequently reduced search time needed to manipulate GPT entries. However, there are seemingly arbitrary deviations from the trend. These deviations are due to cache artifacts. The benchmark results are influenced by the cache friendliness of the page table layout.

To confirm this, a second experiment using custom built kernels with caching disabled was conducted. The same benchmarks ran previously were run with the new kernels. The results are displayed in Figure 4.15. The experiment reveals the expected trend of decreasing (un)mapping costs with reduction in tree depth.

Figure 4.16 shows the results for the MAPS benchmark. The results for G4 and G8 are not shown for clarity and lie approximately between G2 and G16. Given the unmapped regions are powers of 2 in size and aligned, it could be expected that unmapping cost would be constant, as one possible implementation of unmap is simply marking the region as unmapped at the internal node in the GPT that is the

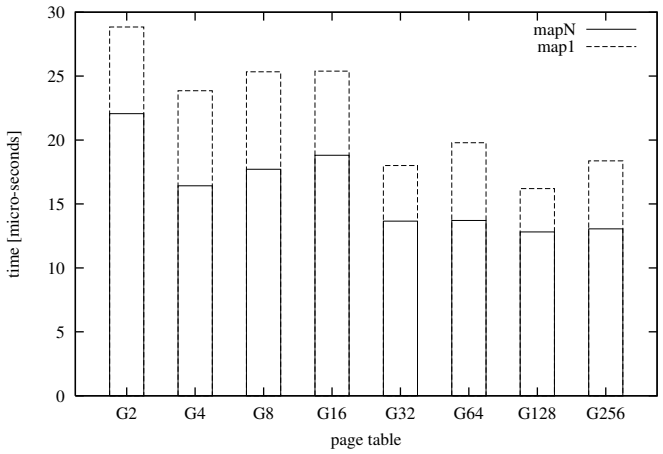


Figure 4.14: Normalised mapping speed for MAP1 and MAPN with caching on.

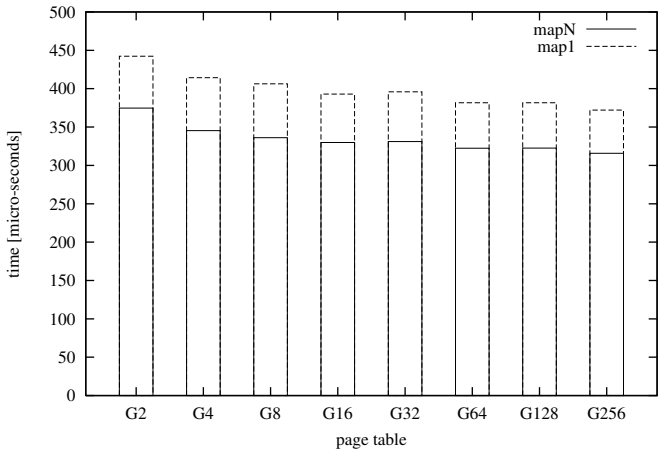


Figure 4.15: Normalised mapping speed for MAP1 and MAPN with caching off.

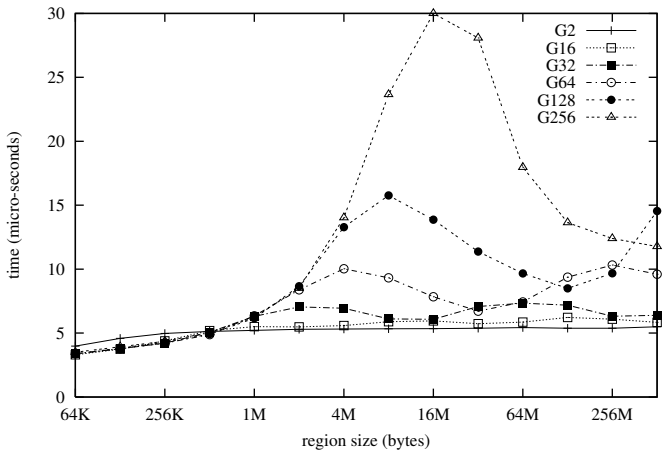


Figure 4.16: Average unmap time for regions of 64K to 512M in size, containing 16 randomly placed pages, for page tables G2 and G16–G256.

root of the region. However, the unmap cost is not constant due to an L4 specific feature. Since L4 allows revocation of pages mapped from one address space to another, the unmap code must ensure any pages contained in the region do not have pages derived from them. If they do, the kernel must also unmap the derived mappings. The kernel achieves this by scanning all the PTEs in the region.

The results reveal that for the smaller nodes sizes (G16 and below) unmap cost is approximately constant, i.e. dependent on the number of pages and not the size of the region being unmapped.

For larger node sizes, the unmap cost displays a dependency on region size not obvious. This dependency can be explained in terms of the *spill-over* effect described in Section 4.3.3. Peaks in the unmap cost correspond to region sizes with the highest probability of the maximal number of nodes needed to store the 16 page-table entries. Given the maximal number of nodes, the higher costs are due to scanning the higher number of nil entries when searching for the 16 valid entries.

The explanation of the unmap cost's dependency on region size can be confirmed by predicting the high-cost points using *spill-over* theory. Re-arranging the *spill-over* equation (4.2) gives the following:

$$A^{'} = \log_2 N + P^{'} - 1 \pm kS^{'}, k \in \mathbb{N} \tag{4.3}$$

Where $A^{'}$ is $\log_2$ of the region size, N is the number pages being unmapped, $P^{'}$ is the effective page size in bits, and $S^{'}$ is the number of bits needed to index a GPT

node. For the MAPS experiment: $N = 16$; $P^{'} = 13$, as PTEs are stored in pairs.

Taking G64 (i.e. $S^{'} = 6$) as an example, high unmap cost should occur at region sizes of 2^{22} (4M) and 2^{28} (256M). Doing likewise for G256 ($S^{'} = 6$), high unmap cost should occur at a region size of 2^{24} (16M). Refering to Figure 4.16, we see that *spill-over* theory correctly predicts the obtained results.

Being able to predict points of high unmap cost seems to be of little practical value other than to confirm exactly what effect is causing the variability illustrated. The important result is that higher GPT node sizes are more likely to suffer the variability illustrated, while smaller node sizes are relatively immune.

4.5 Task Creation and Deletion

The cost of task creation and deletion is influenced by setup and tear-down costs of kernel data structures. In a highly optimised microkernel, the page tables form the dominant part of these costs.

The ability to create tasks quickly is highly desirable in some situations. Besides the vague "efficiency is good" argument, many applications would benefit from low task overhead. Some examples are:

- Scripting languages that execute helper applications, the UNIX shell being an example.
- The Common Gateway Interface (CGI) to web servers. Each transaction results in the executing of a new task which produces a result which is returned by the web server. The overhead associated with executing a new task and its subsequent initialisation has proved a bottleneck resulting in proposals to use persistent tasks [Bro96].
- Protection-Domain eXtensions (PDX) [VERH96] in the Mungi operating system uses tasks to cache address-space mappings for cross-domain procedure calls. The initial PDX invocation involves creating a new task to contain the extended protection domain. Task creation costs represent a significant proportion of the initial invocation costs [Voc99].

4.5.1 Benchmark

A task benchmark was created to evaluate how much of an influence page tables have on task creation and deletion, and evaluate the effect of different GPT implementations. The benchmark aims to measure the cost of creating and destroying a task. To this end, the benchmark measures the elapsed time of 100 interations of:

- create a child task,
- wait for null IPC from child, and
- delete the child.

The child task is a two-page task[2] that simply sends a null IPC to the parent and waits. The benchmark was run 10 times to gain a statistical sample.

4.5.2 Results

The results for the task creation and destruction benchmark reveal that the page-table setup and tear-down costs do influence task creation and destruction significantly. Figure 4.17 shows that page tables G2–G16 have similar low initialisation and destruction costs. In moving to G32 and then on to larger node sizes, the page table management costs increase incrementally, eventually resulting in a more than twofold (113%) increase in the combined task creation-and-destruction cost for G256 when compared to G2. The high penalty for initialising and garbage collecting the larger node sizes is due to the complexity of guard manipulation. For example, the loop that initialises a new node with null guards uses 15 cycles (instructions) per entry, assuming no cache misses. The overhead might be reduced by clever hand-coded assembler, however even halving the overhead still produces the same conclusion: task setup and tear-down costs are significantly affected by page-table management.

[2]Two pages are required to ensure that a GPT node other than the intial root node is required. As mentioned before, root nodes differ in size compared to the other nodes in the tree and can be markedly smaller.

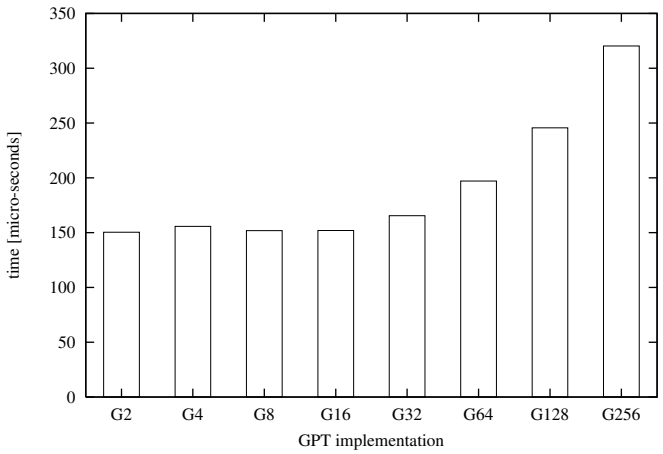


Figure 4.17: Task creation and destruction cost for each GPT implementation.

4.6 Summary and Conclusions

As expected, TLB-refill performance is dependent on GPT tree depth. The tree depth for the conventional applications under test reduces as GPT node size increases and thus refill performance improves as node size increases. The results indicate decoding 4-bits (G16) or 5-bits (G32) per translation step is the point where increasing the node size stops providing a significant pay-back. It is also the point where the depth of the GPT tree is less than depth required for a normal multilevel page-table tree, though a direct comparison is left until Chapter 6.

The current GPT depth is rather ominous for future applications as they grow in size. Larger applications will result in deeper trees requiring decoding more bits per step to keep the number of memory references at the current level. Larger applications are more likely to suffer higher TLB miss ratios, increasing the impact of TLB refill on overall performance.

Page-table size experiments provide several results:

- For the conventional applications, minimal page-table memory overhead is achieved with medium-sized nodes (G16). Small node sizes result in deeper trees with higher overhead due to increased numbers of internal nodes. Larger node sizes result in increased overhead due to higher numbers of empty entries in the page table.

- Page-table overhead for sparse address spaces does not approach the pre-

dicted worst case. Any degree of clustering in the address space reduces the memory overhead significantly.

- The memory overhead of small and medium node sizes is quite immune to variations in address-space layout; they exhibit relatively stable overhead. The higher node sizes are more susceptible to variations in address-space layout and exhibit high variability in memory overhead.

The page-table size experiments also identified the *spill-over* effect. *Spill-over*-effect theory is able to explain (and predict) the situations of high memory overhead.

The tests on (un)mapping performance reveal that for contiguous regions, there is trend of reduced (un)mapping cost as node size increases, though performance is significantly affected by the cache friendliness of a particular GPT's layout. The tests on manipulating sparsely populated regions show that for node sizes of (G16) and below, the performance is determined by the number of valid mappings in the region, and the region size itself has little influence. However, larger node sizes are susceptible to reduced performance due to excessive processing of null guards.

The page-table setup and tear-down costs influence task creation-and-destruction costs significantly. Large node sizes feature significantly higher costs. Small and medium node sizes have little affect on task costs, i.e. they scale with address-space size, with small tasks having low overheads, and large tasks having proportionately high overheads.

4.6.1 Conclusions

Medium single-size GPT nodes perform well in all examined facets of page-table performance. They have manipulation costs and memory overhead directly related to the number of page-table entries involved, not the address-space layout or size. Small single-size GPT nodes exhibit good manipulation and memory overhead properties, but suffer from excessive tree depth and thus poor TLB-refill performance. Large single-size nodes have better TLB-refill performance, but leave manipulation costs and memory overhead susceptible to idiosynchratic address-space layouts. Medium single-size GPT nodes provide the best compromise in overall performance.

4.6.2 Discussion

Two issues that remain unresolved are the effect of future application growth, and the use of multiple node sizes. The applications used for page-table testing are modest in size, larger applications will have an effect on the results. Likewise, multiple node sizes would allow the GPT's structure to be tuned to the application, which will also affect the results.

The performance of TLB refill will be adversely affected by application growth. This is due to the increase in the number of entries in the GPT and the consequent increase in tree depth. The memory overhead, mapping performance, and task creation-and-deletion overhead remains largely unaffected as they are respectively either: independent of application size, not strongly related to tree depth, or expected to be proportional to application size.

The use of multiple node sizes is a way to reduce tree depth and improve TLB-refill performance. Node size could be tuned to application layout to reduce the tree depth by having large populated nodes that decode many address bits in one translation step, and simultaneously provide low memory overhead.

For example, Figure 4.18 shows how the typical text, data, and stack memory layout could be supported with at most two levels in the GPT. The root of the page table divides the address space into two regions. The two child nodes of the root are sized appropriately to contain all the page-table entries for each region. Each node can be enlarged as both regions expand.

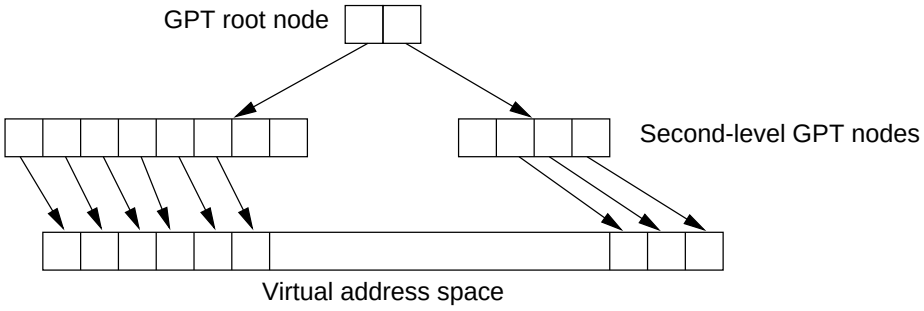


Figure 4.18: Example two-level GPT tree for typical split address-space layout.

As detailed in Section 4.1, the use of multiple node sizes requires a solution to several issues, including how to choose, and how to adapt node sizes. It is not clear what the best methodologies are to choose and adapt node sizes. However, assume

for the sake of argument that the method for choosing and adapting node sizes is at least good enough to achieve the theoretical worst-case page-table size predicted by Liedtke of $2^{m-1}k$ entries, for k mapped pages, where m is the minimum number of bits decoded per translation step. Given such a method, the question arises of how will it affect the results for TLB refill, mapping performance, memory overhead, and task creation-and-deletion overhead. The following examines each aspect in turn and speculates what the effect might be.

The cost of task creation and deletion is proportional to page-table size. A reduction in page-table sized achieved via multiple node sizes would result in a proportionate reduction in task costs.

A reduction in tree depth should result in a reduction in mapping costs due to reduced tree traversal time. However, this expectation makes the unreasonable assumption that the node sizing and adaption scheme adds zero overhead to mapping-primitive costs. The node adaption schemes would be applied during mapping primitive invocation and would have some small or large negative impact on mapping performance.

The use of multiple node sizes has the potential to reduce the memory overhead of the page table. Figure 4.19 shows the theoretical worst-case page-table memory overhead when using multiple node sizes. Note that for the worst-case example, G_n on the x-axis represents a page table that decodes at least $\log_2 n$ bits per translation step. The figure also displays the theoretical best-case memory overhead, the highest observed overhead for the conventional applications, and the results for the SPARSE-FILE and SPARSE-PAGE benchmarks.

In the figure, it is immediately evident that the worst case for multiple nodes sizes is better than the result for the SPARSE-PAGE benchmark with a single node size. The switch to a multiple node size arrangment would result in a significant improvement in memory overhead for the SPARSE-PAGE benchmark. It is also likely that using multiple node sizes would result in a modest improvement in the SPARSE-FILE result. However, the results for the conventional applications are near the best-case memory overhead; only a small improvement is possible, independent of how nodes are sized.

Using multiple node sizes also has the potential to improve TLB-refill performance. As pointed out previously, a typical memory layout can be supported in two levels. However, this simplified memory layout becomes more complex with the

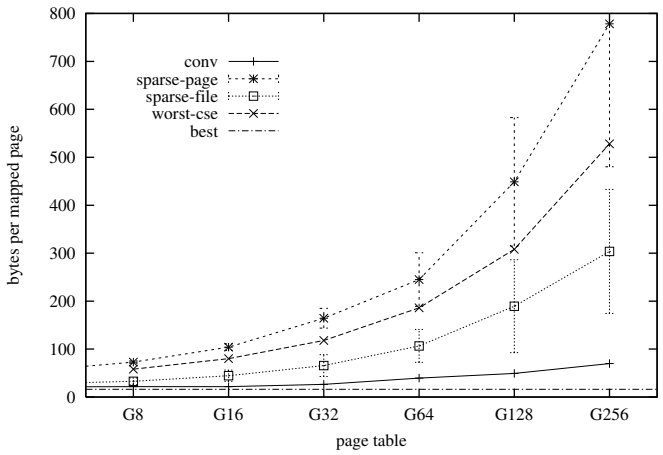


Figure 4.19: Comparison of observed page-table size performance with worst-case size if multiple node sizes are used.

addition of kernel virtual memory. Kernel virtual memory is used in L4 to manage thread control blocks [Lie93], and in other operating systems to page kernel data. On the MIPS R4x00 architecture a minimum of 3 levels is required to support the 3 regions of text-and-data, stack, and kernel virtual memory. This is a result of the virtual address-space limitations of the architecture. Figure 4.20 shows the minimal tree required; kernel virtual memory must be located in the top half of the address space, and text, data, and stack is located in the lower 40-bit user address space.

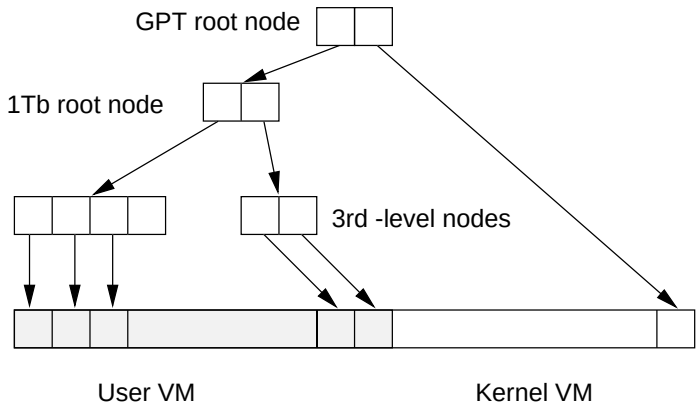


Figure 4.20: Minimum depth GPT for MIPS architecture.

The restriction on user virtual address-space size prevents using a large-sized root node which could point directly to appropriately aligned text-and-data and

stack regions. Instead, the root node points to a root of the user address space, which then points to the two user regions. For the address-space root node to be able to point to both text-and-data and stack simultaneously, the address-space root node needs to contain entries for every 0.5 Tb. Such an address space node would need 2^{25} entries, and thus is impractically large.

The TLB-refill performance of a 3-level tree is the same as for the G128 and G256 GPTs. For the applications used, both these trees were 3 levels deep, and featured an average TLB-refill time of 82 cycles. This can be viewed as the best-case TLB-refill time for GPTs on the MIPS architecture; using multiple node sizes can only improve memory overhead.

Rather than focus on tackling the complexities of using multiple node sizes for improved memory overhead, I chose to leave this for future work. Instead, I focussed on adding a software second-level TLB which should have a more dramatic effect on TLB-refill performance, a fixed memory overhead, and a negligible effect on mapping performance. A software TLB is the topic of the next chapter.

Chapter 5

GPTs with a Software TLB

As pointed out in the previous chapter, guarded page tables have properties that make them desirable as a page table. These include low space overhead, low setup and tear-down costs, and predictable manipulation costs in sparse environments. However, GPTs feature inherently higher TLB-refill costs than hash-based refill schemes which often need only a single memory reference. This chapter explores adding a hashed structure (a software TLB) to GPTs to provide competitive TLB-refill performance while retaining the desirable properties of GPTs.

5.1 Background

The software TLB, second-level TLB, or TLB cache has developed from work on inverted and hashed page tables. Software TLBs (STLBs) are fundamentally hashed page tables with no direct provision for overflow. Collisions in the hash table are usually resolved via replacing the old entry with the new entry, which results in the software TLB acting as a cache of recently used entries. A separate data structure is used to store and retrieve the complete set of TLB entries.

Several variations of the idea exist. One approach taken in the PA-RISC [HH93] is to support the initial lookup, upon TLB miss, in hardware. If the hardware refill handler fails to find a matching entry, then a trap to a software routine is taken which can look up the entry in another page table, or follow an overflow chain if a hashed page table is used. A similar approach is taken on the PowerPC [Pow97].

A completely software-based approach [BKW94] is also possible. In this particular approach the general exception handler was modified to detect kernel TLB

misses early in the exception handler, which then used a direct-mapped[1] software TLB to cache entries and reduce the kernel PTE refill costs. To elaborate a little more, the MIPS R2000 [KH91] uses a virtual array to store page-table entries. User TLB misses are handled by a specialised software exception handler which indexes the virtual array and loads the appropriate entry. Kernel-mode misses (which include TLB misses on kernel-mapped memory and misses on the page-table array) are handled via the more expensive general exception vector. TLB misses on kernel memory (both mapped memory and user page tables) are handled via a second virtual page-table array termed a *level-2* array, with the user page tables termed *level-1*. The *level-2* page-table entries are themselves in a virtual array, which upon access can suffer yet another TLB miss. Page-table entries for this array (*level-3* entries) are stored in physical memory to halt this cascading process. The addition of the STLB improves performance by shortcuting the cascading by caching *level-2* and *level-3* entries, thus reducing expensive TLB misses via the general exception vector.

The SPARC Spitfire[You94] takes a hybrid approach, handling refill in software, but providing some hardware assistance. Like the MIPS R4x00 [Hei93], a register is provided by the hardware which, upon a TLB miss, is set to the address of the appropriate place in the page table for the miss handler to load the required page-table entry. While the register on the MIPS was designed for linear arrays, the register on the Spitfire is designed for an STLB, termed a Translation Store Buffer (TSB). The MMU allows the operating-system designer to vary the size of the translation store buffer via a second register which modifies the pointer generation appropriately. This arrangement is quite flexible with regard to size of the TSB, however no direct support is provided for varying the associativity. Associativity can be achieved at a cost, by appropriate modification of the hardware-provided pointer in software during each invocation of the refill handler.

Most of the attention given to software TLBs has been rightly focused on reducing TLB-refill time, with significant success. Uhlig [UNS+94] discusses the effect of operating-system design on TLB performance, however I found little attention has been given in the literature to what parts of the software-TLB design space affects various areas of performance of the operating system. A brief discussion of a Hashed Resident Page Table (HRPT) and "protection keys" in a single-address-

[1] "Direct mapped" in software TLB technology refers to using a selection of appropriate low order bits of the virtual address for the hash function to directly index the STLB.

space system can be found in Chase's thesis [Cha95]. Chase proposes a hash-based structure for single-address-space operating systems. The structure has multiple tags per virtual-to-physical translation. The tags form protection keys to select when a translation is active in one of the potentially many protection domains.

Channon [CLK97] discusses general issues of size, tag management, and associativity in an examination of the applicability of skewed and column-based associativity in TLBs. Briefly, skewed and column-based associative STLB lookup uses two or more hash functions for selecting entries in the STLB to provide wider dispersion of entries and hence reduced conflict misses.

Talluri [Tal95] describes the applicability of clustering to software TLBs in a more general examination of subblocking in the address-translation hierarchy.

What follows is a discussion of major components of the design space and the issues that arise in designing a software TLB for the L4 microkernel in particular, however, the discussion also has general applicability.

5.2 Software-TLB Design Issues

The software TLB is simple in concept. In implementation however, there are several parameters to the design space that need to be considered. These parameters are *size, associativity, hash function,* and whether to cluster entries. In addition, software TLBs can be per address space or can be shared between all address spaces.

5.2.1 Hash Function

The hash function transforms a virtual address into a location in the software TLB. The aim of a good hash function is to be fast, and to minimise collisions in the hash table. Ideally, the hash function provides a uniform distribution within the hash table, which is reasonably independent of the distribution of keys.

Knuth suggests two simple algorithms based on multiplication or division that exhibit good properties [Knu73], but both use multiplication or division, which are expensive operations when considered in the light of a cycle-by-cycle optimised TLB-refill handler.

TLB-refill routines use simpler hash functions that are fast to calculate and rely somewhat upon the distribution of addresses to provide dispersion, instead of the properties of the hash function itself.

The PowerPC [Pow97] and PA-RISC [HH93] use a simple XOR of a selection of upper and lower address bits of the faulting address. The PA-RISC uses the upper half of the 64-bit addresses as address-space identifiers. The address-space identifiers are pseudo-randomly allocated to processes, which helps the performance of the hash function.

Bala reports good performance with a simple bit selection (direct mapped) arrangement, which is achieved simply by masking the appropriate number of least significant bits [BKW94]. The SPARC Spitfire TSB [You94] also uses a similar direct-mapped arrangement.

This thesis is not intended to be an investigation into STLB hash functions. The STLB implementations in this thesis use bit-selection for the hash function. Bit-selection is the least complex hash function and results in the shortest TLB-refill routine.

5.2.2 Associativity

Set associativity in cache memories has been studied for a long time, starting with Smith [Smi78b]. Smith later compiled two large bibliographies on cache memories which can be consulted for further references, particularly on cache associativity [Smi86, Smi91].

Recent studies [UNS+94] on associativity in TLBs argues for increasing the number of entries via reduced associativity to combat increasing TLB miss ratios in the face of increasing working-set size and physical memory. Another study [CLK97] proposes applying column and skewed associativity techniques [Sez93]. A column or skewed associative lookup uses more than one hash function to locate entries in the TLB. Simplifying to some extent, a column associative lookup first uses one hash function for an intial probe, which if unsuccessful, a second probe based on a different hash function is performed. Argarwal [AP93] provides further details.

Skewed associativity [Sez93] divides the TLB into two distinct banks similar to two-way set associativity. However, instead of using the same hash function to index each bank, skewed associativity uses a different hash function to index each bank.

These techniques all have corresponding implementations in a software TLB, though several factors may preclude their use.

- Bit-string manipulation in software is time consuming. Calculation of indices in column and skew techniques can be done within the access cycle of a hardware TLB. A software implementation is likely to consume several cycles forming an index, which in turn, is unlikely to return a sufficiently lower miss ratio to compensate for the extra expense.

- Hardware lookups are in parallel, software lookups are sequential. A two-way set associative lookup needs to load two tags to check for a match. Hardware can do this in parallel, software requires two sequential memory references and sequential comparison to check for a potential match, at least on a single issue processor.

- An implementation of n-way associativity would ideally place the n entries of a set consecutively to minimise the number of cache lines used during a lookup. Skew and column associativity places entries of a set independently, and hence accesses more cache lines than the simple n-way scheme.

To investigate the potential for using associativity in a software TLB on the R4x00 architecture, refill routines for both a one-way (direct mapped) and two-way STLB were developed. There are several reasons for using this as a starting point:

- The one-way case, with a high hit rate, provides the best-case refill time. The one-way has both the shortest refill routine (18 cycles with no cache misses) and references the least memory. It accesses only half a cache line to refill the TLB. The routine only requires (saves and restores) a single temporary register [2].

- Two-way associativity is the next incremental step. It has the least increase in miss penalty of all associativity schemes. The bit-selection scheme used to index the STLB is similar to the one-way case. The refill routine is 20 and 22 cycles long, depending on which entry in a set matches (assuming all cache hits). It references twice as much memory as the one-way case, however the memory is still contained in the same cache line. It requires two temporary registers.

[2]The MIPS R4x00 architecture only provides two registers for use by the TLB refill routine. If extra registers are required, they must be saved and restored by the refill routine.

- Higher associativity schemes require references to more than one cache line, which increases the average number of lines referenced per lookup.
- Column and skewed associativity requires more complex indexing and also references more than one cache line.

Based on the previous reasoning, I am sceptical as to whether associativity has a place in a software-based TLB-refill routine. This assertion is re-enforced by the fact that most implementations of software TLBs are direct mapped [BKW94, HH93, You94], with the exception of the PowerPC [Pow97] with its hardware-based initial lookup.

5.2.3 Multiprocessor STLBs

The two main problems that arise for virtual memory management in multiprocessors are TLB consistency and safe multi-threading of page-table operations. TLB consistency can be managed by special hardware or, mostly, by software. Teller provides a good review of various techniques [Tel90]. The techniques are mostly independent of page-table type and thus orthogonal to STLBs.

Multiprocessor scalability of page-table operations is examined by Khalidi *et al.* [KJW94]. Their examination of the "hardware address translation" (HAT) layer in Solaris identified the single lock on the entire HAT layer as being barely sufficient for two CPUs, for the applications under test. They undertook an incremental refinement, multi-threading various parts of the HAT layer to improve scalability and performance. The techiques used were orthogonal to the page table used in their implementation.

STLBs can be treated as just another page-table type, and as such, I believe they neither significantly ease or complicate page-table management in a multiprocessor, hence I will not examine this further.

5.2.4 Single or Shared STLBs?

A software TLB can be designed as a cache for a single page table (i.e. each task has its own STLB), or a STLB can be shared between all running tasks on a machine.

A per-task STLB is not the preferred option as it either requires statically reserving, or dynamically managing, the physical memory used to store them. Statically

reserving memory has the problem of *a priori* predicting the number of STLBs required. Dynamic management incurrs additional overhead. Given physical memory is a limited resource, it makes some sense to move the STLB to virtual memory, however this suffers the penalty of cascading TLB misses.

A single shared STLB can simply be placed in physical memory to avoid cascading TLB misses. It can be sized larger for a better hit ratio, and at the same time, consume less overall physical memory when compared to a per-task STLB.

To signify which entries are applicable to which task, a shared STLB requires each entry to be augmented with an address-space identifier (ASID) similar to hardware TLBs [DEC96, Hei93, You94]. Several possibilities for implementation exist depending on the architecture being used:

- On architectures with a hardware defined ASID used in the TLB, the ASIDs can be used directly to augment STLB entries. This can reduce the complexity of tag management considerably if the architecture forms a tag in the hardware suitable for comparison (e.g. SPARC Spitfire [You94]). A complication arises on architectures on which the number of address spaces can exceed the number of available ASIDs. Potential solutions to this problem are discussed in Section 5.2.5.

- On architectures without hardware ASIDs, one can synthesize STLB specific ASIDs in software. A software ASID slows the TLB-refill process as it needs to be merged in some manner with the faulting address, and compared with the tag of the appropriate candidate for a match. I do not explore this further as very few modern CPUs have no support for hardware ASIDs of some type.

Like the TLB, the STLB must be kept consistent after (un)maping operations. The (un)mapping operations involve page-table traversals that examine page-table entries, of which the associated virtual address is easily determined. Given a virtual address, the cost of probing the STLB to ensure consistency should be minimal, especially as it must be similarly done for the TLB. This is equally applicable to both shared and per-task STLBs.

Similarly, the shared STLB should have little effect on task setup and tear-down costs. The STLB need only be initialised upon system startup, after which it is always in a known state, assuming (un)mapping correctly ensures consistency. Tear-down costs should be inconsequential as either the address space is large and

a STLB scan has little effect, or if the address space is small, the STLB can be probed and invalidated for each valid entry in the page table when the page table is garbage collected.

The shared STLB has a potential advantage on some software-loaded TLB architectures as it can lie at a constant address. If placed appropriately, its base address can be loaded by a TLB-refill routine with a single immediate instruction, which avoids loading the page-table address from memory.

5.2.5 ASID Management

Managing hardware ASIDs only becomes an issue when the number of address spaces exceeds the number of available ASIDs. Prior to this point, hardware ASIDs can be used directly within the STLB. Some CPUs, notably the MIPS R4x00 [Hei93] and the SPARC Spitfire [You94], form tags including both the virtual address and ASID. These tags are suitable for use as (and subsequent comparison with) tags within the STLB.

The SPARC Spitfire features a 12-bit ASID which is likely to be sufficient for the number of concurrent address spaces in most systems. However, current MIPS and Alpha [DEC96] processors only feature 256 and 128 ASIDs respectively. Most systems are likely to exceed these numbers at least occasionally.

When the number of tasks exceeds the number of available hardware ASIDs, a method of fairly and efficiently sharing the available ASIDs amongst competing tasks is required. Efficient ASID management is an issue as it affects IPC performance. IPC from one task to a second task that is ASID-less could involve reclaiming an ASID from a third task and reassigning the ASID to the second. This introduces a level of uncertainty to IPC performance. Table 5.1 lists measured L4 IPC times for various STLB configurations with, and without, an ASID reclamation. The numbers are obtained by restricting the microkernel to a single ASID, consequently forcing the kernel to reclaim and reassign the ASID for each IPC. It can be seen that an ASID reclaim involving a large STLB adds greatly to the IPC cost. However, one would expect ASID reclaims to represent a small proportion of the overall system overhead, since one expects the set of active tasks in the system to exhibit temporal locality, with ASID reclaims happening when the active set slowly evolves with time.

While ASID reclamation and reassignment in the context of overall system

Scenario	Costs
Normal IPC	0.99 μs
IPC with ASID reclaim (no STLB)	8.5 μs
IPC with ASID reclaim (8K STLB)	46 μs
IPC with ASID reclaim (128K STLB)	1.7 ms

Table 5.1: IPC costs involving ASID reclamation.

overhead is not a huge issue, IPC latency in the context of interrupt handling is. Device drivers in L4 are user-level tasks which receive hardware interrupts via IPC. IPC latencies of a millisecond in this context is likely to cause interrupts to be missed, limit device driver performance, or even result in device driver malfunction. Device drivers need to be considered as special tasks in the system that require special ASID handling to avoid excessive interrupt latency.

The ASID management problem is analagous to page replacement in a typical virtual-memory system: ASIDs can be thought of as physical frames, tasks (or more precisely, address spaces) as virtual pages, and ASID reclamation as page faults. Thus ASID management can be divided into three policies having virtual-memory analogies.

ASID assignment is analogous to fetch policy. ASIDs could be assigned on demand, or pre-fetched in some manner.

ASID selection is analagous to page placement, but while page placement may affect performance (e.g. improving cache performance via page colouring [KH92]), all ASIDs perform equally and thus the only selection policy needed for assignment is selection of a free ASID.

ASID replacement is analagous to page replacement. Like a page-replacement algorithm that attempts to satisfy the demands for virtual memory and simultaneously minimize page-fault rate, an ASID replacement algorithm would attempt to satisfy the demands for address spaces and simultaneously minimize the number of ASID reclaims.

Initially, I planned to investigate the application of virtual-memory management policies, such as FIFO or clock [CH81], to ASID management. However, I have left this as future work as current workloads used on the system have not exceeded

the number of available hardware ASIDs. I also expect the number of available ASIDs to increase in future processors.

The current kernel is implemented as follows: ASIDs are assigned on-demand except for device-driver tasks. When a task registers to receive interrupts, its ASID is "wired" to it for the duration of its registration. All other unwired ASIDs participate in a FIFO replacement algorithm.

5.3 Experimental Evaluation

5.3.1 Implementation

To compare associativity and examine the effect of adding an STLB to GPTs, the following page-table implementations were tested.

No STLB An unmodified, 16-entry per node, guarded page table as described in the previous chapter. It forms the baseline to which the following STLB implementations are compared.

1×128K A G16 GPT with the addition of a one-way set-associative 128 kilobyte STLB. It contains 8192 entries of 16 bytes each. The number of entries is sufficient to cover all physical memory in ideal conditions.

2×64K A G16 GPT with the addition of a two-way set-associative 128 kilobyte STLB. It contains 4096 pairs of entries of 16 bytes each. The number of entries is sufficient to cover all physical memory in ideal conditions.

1×8K A G16 GPT with the addition of a one-way set-associative 8 kilobyte STLB. It contains 512 entries of 16 bytes each. In ideal conditions, it can covers 4M of memory, which is one sixteenth (6.25%) of the physical memory in the test machine.

2×4K A G16 GPT with the addition of a two-way set-associative 8 kilobyte STLB. It contains 256 pairs of entries of 16 bytes each. It can cover the same amount of memory as the 1×8K STLB.

The layout of each STLB entry is illustrated in Figure 5.1. The two-way associative STLB uses pairs of entries placed consecutively in memory and indexed appropriately.

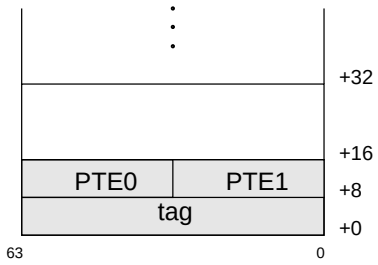


Figure 5.1: The layout of the STLB.

The motivation for the choice of the particular STLB sizes is as follows. The large STLB should achieve near 100% hit ratio and thus best-case performance for TLB refill. This allows direct comparison between the raw refill speed of the two associativity schemes, without the influence of differing miss ratios. The small STLB will exhibit a more realistic miss ratio allowing comparison of the effect of associativity on miss ratio and TLB-refill performance. The varying of STLB size will give an indication of scalability issues with respect to TLB refill and other areas of microkernel performance.

5.3.2 Benchmarks

The set of benchmarks used to evaluate STLB performance examines the same areas as the GPT evaluation. The benchmarks examine the STLB's effect on TLB-refill performance, microkernel mapping primitives, and task creation-and-deletion overhead. I do not examine page-table memory overhead as this is mostly determined by the underlying guarded page table that the STLB is caching.

To investigate TLB-refill performance, I use the metrics and the conventional applications described in Section 4.2.1. Summarising briefly, the metrics used are overall application runtime and average TLB-refill time obtained via instrumentation. The applications are traditional UNIX applications, each application is run in turn and the results recorded.

The unmap system call is examined to investigate the cost of keeping the STLB consistent during page-table operations. Unmap is a good candidate as STLB consistency cannot be managed lazily (as it can with map), and it is a simple operation that only involves system-call argument checking, generation of the boundaries of the region to be operated on, and a GPT traversal that involves both mapping-tree

modifications and simple bit manipulation in the page-table entries. Being a simple operation, unmap is likely to reveal STLB-consistency overheads better than other operations.

The metric used is elapsed time of the unmap operation normalised to a single page, i.e. elapsed time divided the number of pages unmapped. The method used is to map a region of size X, and measure its elapsed time using the hardware counter. The region X is varied from 4K (a single page) to 16M in powers of 2 increments. At each point, the benchmark is repeated 10 times to obtain a statistical sample.

The task benchmark of Section 4.5 is used to examine the effect of adding an STLB on task creation and deletion. Briefly, the benchmark measures the elapsed time of 100 interations of:

- create a child task,
- wait for null IPC from child, and
- delete the child.

5.3.3 Results

TLB Refill

The dramatic reduction in average TLB-refill cost resulting from the STLB can be seen in Figure 5.2. The large 1×128K STLB performs the best with a reduction in average TLB-refill cost from 101 cycles to 32 cycles (68%). The similarly sized two-way associative STLB did not perform as well with a reduction of 57% to 43 cycles. Given that the routines only differ by approximately 3 cycles (if all memory references hit in the data cache), the large difference between the two can only be attributed to high data-cache miss ratios during STLB refill, which results in a larger memory-reference penalty for the two-way case. To elaborate, assume for the two-way STLB an equal distribution of hits for the two entries in a set. The two-way STLB references half a cache line 50% of the time and the whole cache line the other 50%. Comparing this to the one-way STLB which accesses half a cache line 100% of the time, in the presence of cache misses the two-way has to wait on average for 1.5 times more memory to load into the cache. For the R4700, the penalty of using extra memory within a cache line is visible. The R4700 allows instructions to proceed in parallel with cache refill. Data is available

to the CPU once it is loaded in the cache. The CPU does not have to wait for the complete line to refill for the data to be available, this is not necesssarily the case with cache architectures that stall the processor during refill, or make the first word loaded into a cache line available, but stall any further access to the cache line until it is completely refilled. The memory reference penalty on these other cache architectures would be similar for both one-way and two-way associativity. Later, Section 5.3.4 discusses associativity more generally, including sensitivity to changes in the memory reference penalty.

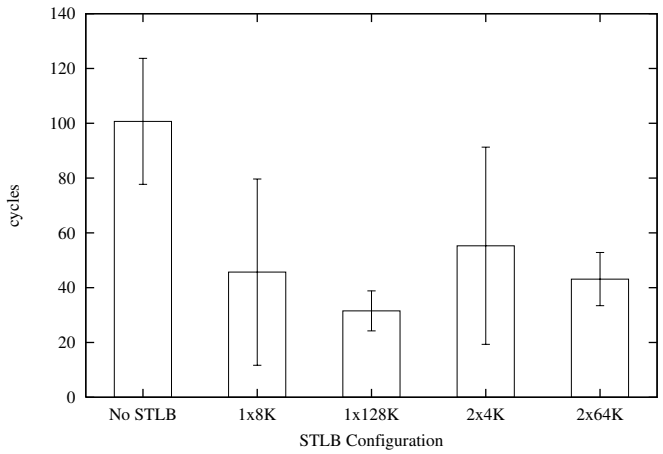


Figure 5.2: Average TLB refill time (cycles) for G16 with various STLB configurations. Error bars represent standard deviations.

The average refill cost for the smaller 8K STLBs is also significantly reduced compared to the no STLB case. The cost reduction is 55% (to 46 cycles) for the one-way case, and 45% (to 55 cycles) for the two-way case. This is still a dramatic performance improvement given the modest size of the STLB. It also corroborates that 2-way associativity is more expensive.

The standard deviation (represented by the error bars in the figure) shows the variation in TLB refill per application run compared to the overall average across all applications. The small variation for both the 128K STLBs can be attributed to the near 100% hit ratio in both the STLBs for all applications; the variation is essentially due to varying hit cost in the STLB. The large variation for the 8K STLBs is explained by the varying hit ratio (from 42% to 100%) for the differing applications. The standard deviation of individual benchmark runs normalised to

the mean are typically less than 0.1%.

Table 5.2 details the effect of adding an STLB on application runtime. The runtime is normalised to the runtime of the no-STLB case (G16) to directly compare relative performance improvement. Overall application runtime is reduced by 0 to 34%, with the average reduction being 10% for the 1×128K STLB when compared to the no STLB case. The difference in performance between larger and smaller STLBs, and also one-way and two-way associativity, is only marked for the applications having high TLB miss ratios (`mm`, `c4`, `gcc`). In the case of `mm`, which features a high hit ratio for all STLBs (see Table 5.3), we can see the penalty incurred by the extra memory reference required for the two-way STLB. Other applications feature small differences of around 1%, always in favour of one-way associativity.

BENCH	No STLB	1×8K	2×4K	1×128K	2×64K
go	1.00	0.98	0.98	0.98	0.98
swim	1.00	1.00	1.00	1.00	1.00
gcc	1.00	0.94	0.95	0.86	0.89
compress	1.00	0.91	0.92	0.91	0.93
apsi	1.00	0.97	0.97	0.97	0.97
wave5	1.00	0.94	0.94	0.93	0.94
c4	1.00	0.80	0.82	0.73	0.76
nsieve	1.00	0.96	0.97	0.96	0.97
heapsort	1.00	0.99	1.00	0.99	1.00
mm	1.00	0.67	0.75	0.66	0.75
tfftdp	1.00	0.89	0.89	0.88	0.89
Average	1.00	0.91	0.90	0.92	0.92

Table 5.2: Elapsed times normalised to G16 for each combination of application and page table.

Table 5.3 also illustrates that there is no consistent difference in hit ratio between the one-way and two-way STLBs for the applications under test. In no case tested was the hit ratio of the two-way STLB large enough above the one-way STLB to translate into better overall performance.

BENCH	1×8K	2×4K	1×128K	2×64K
go	1.00	1.00	1.00	1.00
swim	0.46	0.42	1.00	1.00
gcc	0.68	0.72	1.00	1.00
compress	0.99	1.00	1.00	1.00
apsi	0.99	1.00	1.00	1.00
wave5	0.98	1.00	1.00	1.00
c4	0.82	0.81	1.00	1.00
nsieve	1.00	1.00	1.00	1.00
heapsort	1.00	1.00	1.00	1.00
mm	1.00	1.00	1.00	1.00
tfftdp	0.99	0.99	1.00	1.00

Table 5.3: STLB hit ratio for each combination of page table and conventional benchmark.

Unmapping Performance

Figure 5.3 illustrates the cost of keeping the STLB consistent after unmap operations. It shows the cost of an umap operation for a G16 without, and with one-way and two-way STLBs. Different sized STLBs are not shown as they performed similarly and only clutter the graph. The unmap cost is normalised to a per-page-unmapped cost, i.e. the unmap cost is divided by the unmapped-region size.

In the worst case, keeping the STLB consistent costs about 9% for the one-way STLB and approximately 17% for the two-way. Cache effects cause a minor inconsistency in the results for smaller region sizes, the cost of STLB consistency "improves" performance. Figure 5.4 shows the results for the same experiment with the cache turned off, thus removing the cache effects to show the inconsistency in the previous results was not due to other influences.

This inconsistency can be attributed to cache artifacts that are more likely to manifest themselves in operations on small regions. The operations on large regions provide a better indicator of the inherent cost of STLB consistency.

Task Performance

The effect an STLB has on the task benchmark is illustrated in Figure 5.5. Consistency managment of the STLB results in a slight increase (5%) in the cost of

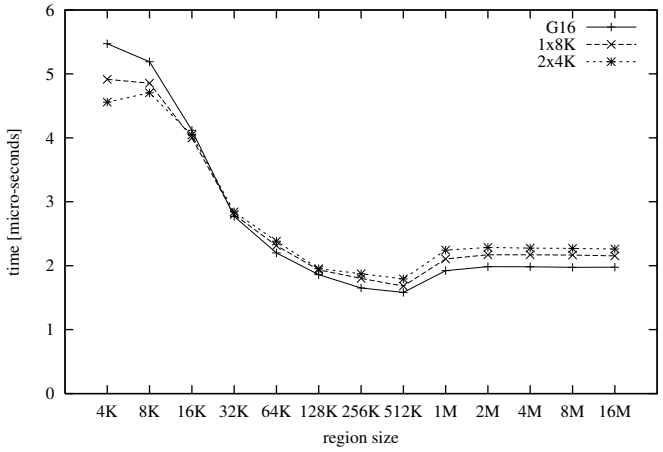


Figure 5.3: Unmap cost normalised to per page unmapped, for increasing region size and each page table implementation with cache on.

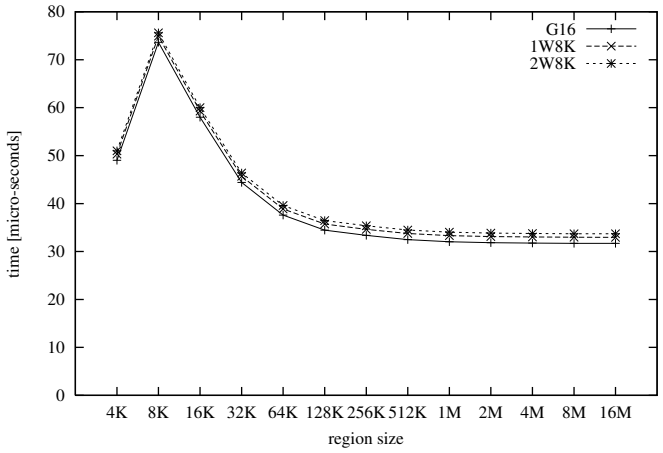


Figure 5.4: Unmap cost normalised to per page unmapped, for increasing region size and each page table implementation with cache off.

creating and deleting a task. This is a negligible increase given that the benckmark spends very little time in user-mode code.

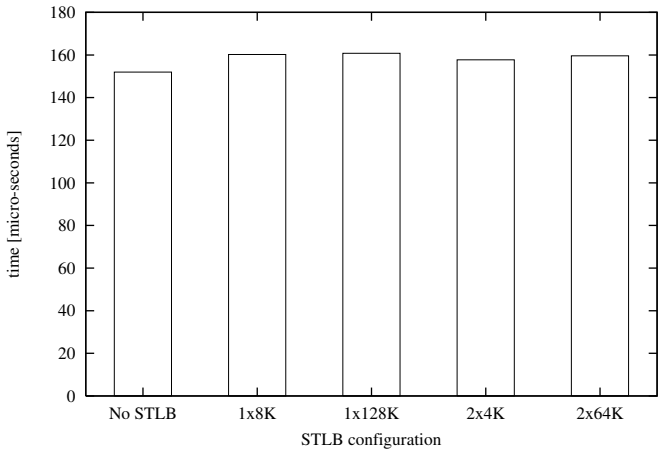


Figure 5.5: Task creation-and-destruction cost for G16 with various STLB configurations.

5.3.4 Associativity Revisited

From the previous experimental results it is clear that the extra TLB-miss penalty associated with the two-way STLB is not compensated by a sufficiently improved STLB hit ratio for the two-way STLB to perform as well, or better than, the one-way STLB. However, the experiment does not give an idea of how much an improvement in two-way STLB hit ratio would be sufficient for it to perform as well as the one-way STLB, nor does it provide any information as to how sensitive this *sufficient improvement* in hit ratio is to changing the TLB-miss penalty associated with the two-way STLB.

The STLB hit-ratio can vary greatly with the software application. The difference in hit ratio between STLBs of differing associativity can similarly vary. The average refill penalty difference between two STLBs can also vary significantly with a change in cache size or memory bus speed. To provide a more general comparison between two STLB implementations, I derive an approximate formula for required improvement in hit ratio and analyse it.

The following notation is used: R is the STLB average hit ratio of the first implementation, $R + \Delta R$ the hit ratio of the second implementation; H is the

average cost of a TLB refill that hits in the STLB for the first implementation, $H + \Delta H$ is the average cost for the second; M is the average cost of a TLB refill that misses the STLB for the first implementation, $M + \Delta M$ is the average cost for the second.

Thus the average TLB-refill cost (C_{stlb1}) for the first implementation is,

$$C_{stlb1} = RH + (1 - R)M$$

and the average cost for the second is:

$$C_{stlb2} = R(H + \Delta H) + (1 - (R + \Delta R))(M + \Delta M)$$

For the second implementation to, on average, outperform the first C_{stlb2} must be less than C_{stlb1}. This equates to:

$$R(H + \Delta H) + (1 - (R + \Delta R))(M + \Delta M) < RH + (1 - R)M$$

which after expanding and removing common terms is:

$$\Delta R(H + \Delta H - M - \Delta M) < -R(\Delta H - \Delta M) - \Delta M$$

which after further re-arranging becomes

$$\Delta R > \frac{R(\Delta H - \Delta M) + \Delta M}{M + \Delta M - H - \Delta H}$$

We now make the approximation that the difference between the miss cost ($M + \Delta M$) and the hit cost ($H + \Delta H$), is equal to the lookup cost of the underlying page table (P). Thus, after further re-arrangement, we have the following inequality,

$$\Delta R > \frac{R\Delta H}{P} + \frac{(1 - R)\Delta M}{P} \tag{5.1}$$

The inequality (Equation 5.1) specifies the required improvement in hit ratio (ΔR) of a second STLB, for the second STLB to perform better than the original STLB, given the second STLB has an extra hit penalty (ΔH) and miss penalty (ΔM).

Equation 5.2 optimistically assumes the ΔM component (in Equation 5.1) is negligible. Even if it is not, Equation 5.2 must be true when Equation 5.1 holds true.

$$\Delta R > \frac{R\Delta H}{P} \tag{5.2}$$

ΔR also has the constraint that $R + \Delta R < 1$, as hit ratio cannot exceed 1. This leads to the following constraint on the hit ratio of the original STLB:

$$1 - R > \frac{R\Delta H}{P}$$

Which when re-arranged is

$$R < \frac{P}{P + \Delta H} \tag{5.3}$$

Hence, Equation 5.3 specifies the optimistic limit of hit ratio of the original STLB, for it to be theoretically possible for a second STLB to perform better, given the second STLB has an increased hit penalty of ΔH.

Figure 5.6 shows Equation 5.3 graphically. It plots the limit on hit-ratio for the original STLB against the average TLB-refill cost of the underlying page table (P) for various second STLB hit penalties (ΔH). The penalties are 3, 5, 10, 15, and 20 cycles. 3 cycles is the minimum average penalty for using two-way associativity (given no cache misses) in the STLB designs used in the previous experiments.

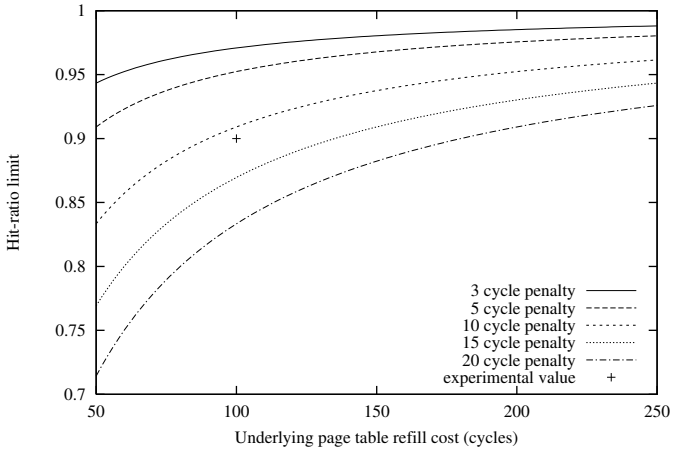


Figure 5.6: Plot of optimistic upper limit of original STLB hit ratio (R), for it to be theoretically possible for a second STLB to compensate for various additional refill penalties (ΔH), versus average underlying page table refill cost.

Figure 5.6 does not reveal whether one STLB configuration is better than another, but it does reveal whether it is theoretically possible for another STLB to better a given reference STLB. In the TLB-refill experiments undertaken, the underlying page table's average TLB-refill cost (P) was 100 cycles (see Section 4.2.2), and the difference in hit cost for the large STLBs (ΔH) was 11 cycles (see Figure 5.2).

The graph reveals for the experiments undertaken that the limit on hit ratio for the one-way STLB is 0.9, for it to be theoretically possible for the two-way to perform better. This point is highlighted in the figure by the plotted *experimental value*.

If one considers that the modest 8K one-way STLB achieves at least a 98% hit ratio in 8 out of the 11 benchmarks tested, and that scaling the one-way STLB to 128K (and beyond) to achieve higher hit ratios for other applications is feasible, there is no scope for adding associativity in pursuit of further performance gains on this architecture.

5.4 Conclusions

The addition of an STLB acting as a software cache of TLB entries improves the TLB-refill performance of GPTs dramatically. This translates to significant improvements in application runtime. Experimental and theoretical results show that STLB refill time is critical for performance; a small increase in TLB-refill penalty may be impossible to compensate. STLB refill-routine designs should be simple and fast. There is no scope for using associativity in a STLB in the environment under study. The environmental situation would need to change significantly for associativity to deserve further consideration.

Adding an STLB had a small detrimental effect on other microkernel primitives. The need to keep the STLB consistent introduced a small penalty for both mapping and task creation-and-deletion primitives. However, the improvement in TLB-refill performance should offset the small penalty.

The GPT+STLB combination appears to be very attractive for use in a microkernel. It features low refill costs, low space overheads, low setup and tear-down costs, and fast mapping operations. However, one needs to compare the combination with other page-table structures to prove conclusively that the GPT+STLB combination does perform better than traditional hashed or hierarchical page-table structures. The next chapter performs this comparison.

Chapter 6

Comparison with Other Page Tables

The previous chapters examined GPTs both without and with an STLB. The combination of a GPT with 16 entries per node (G16), together with a large one-way set-associative STLB, was found to perform better in the completed benchmarks than other structural variations of GPTs and STLBs. However, all quantitative comparisons thus far have involved variations of GPTs of one form or another. There has been no quantitative comparison with more traditional structures.

The goals of this chapter are to quantify the relationship between GPT+STLB performance and the performance of more traditional page-table structures, and also to evaluate the traditional page tables in terms of microkernel primitives. To achieve this goal several page-tables were chosen, implemented, and subsequently tested. The benchmark results are then compared with the GPT+STLB results.

6.1 Page Table Implementations

Two variants of the GPT+STLB combination are used in this comparison. One variant is a G16 GPT together with an 8K one-way set-associative STLB containing 1024 page-table entries(termed G16+S1024 from here on); the other variant is G16 GPT together with an 128K one-way set-associative STLB containing 16384 entries(termed G16+S16384).

The other page tables chosen for comparison were selected from those reviewed in Chapter 3. The three basic types implemented were the multilevel page table (MPT), the hashed page table (HPT), and the clustered page table (CPT). The inverted page table (IPT) was not implemented as it is less flexible and performs

worse than the HPT [HH93]. The virtual linear array (VLA) was also not implemented due to the current microkernel's inability to support cascading TLB-refill faults, i.e. it cannot handle a TLB-refill fault while servicing a TLB-refill fault, which is a requirement for implementing a VLA. The consequence of the VLA's omission from this page-table study is discussed later in Section 6.4.2. As implied by the inability to handle cascading TLB faults, all the page-table implementations are contained in physical memory.

Note that the inability to support cascading TLB refills is merely a limitation of the current implementation. There is nothing inherent in the microkernel's design that prevents adding support for cascading TLB refills in a future version.

6.1.1 Multilevel Page Table

Figure 6.1 illustrates the structure of the implemented MPT. The MPT uses a 4K node size which corresponds to the hardware page size. This size was chosen as it is a reasonable trade-off between tree depth and node granularity, and because it corresponds to the LVA node size, it should provide an estimate of the LVA memory-consumption characteristics. The 4K node size for leaf and internal nodes leaves only 4 bits remaining to index the root node, which is 64 bytes in size.

The page table has six levels indexed by the bit selections illustrated (13-20, 21-30, 31-40, 41-50, 51-60, 61-63). The pointers contained in the root and internal nodes have been shortened to 32 bits, and rely on sign extension to form valid 64-bit pointers. This technique conserves memory, gives a more compact page-table structure, and improves the chances of pointer entries remaining in cache.

Figure 6.2 illustrates the format of the MPT leaf nodes. Instead of 32-bit pointers, they contain 256 page-table entry pairs (16 bytes per pair). The format of a PTE pair is illustrated in the shaded section. It contains two page-table entries and the mapping-tree pointers associated with the page-table entries.

The page-table entries for the kernel-mapped memory (an array of thread control blocks) are shared between all address spaces by sharing a common branch in all page tables. This saves memory and avoids the problem of keeping the kernel-mapped memory consistent in all address spaces.

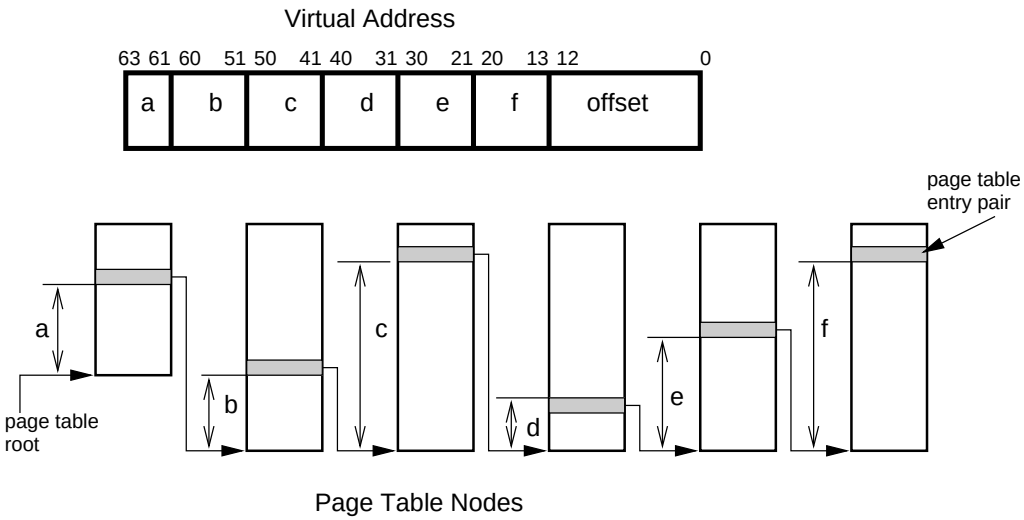


Figure 6.1: Multi-level page table implementation.

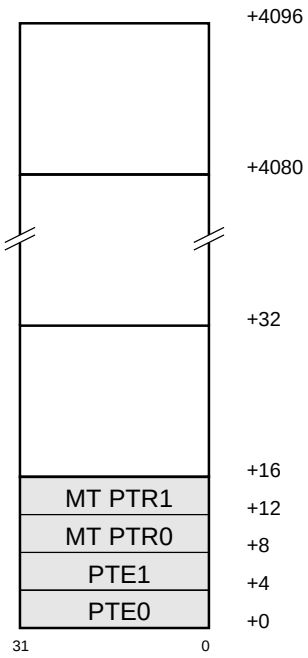


Figure 6.2: Multi-level page table leaf node implementation.

6.1.2 Hashed Page Table

Figure 6.3 illustrates the design of the implemented hashed page table. The HPT consists of consecutively laid out buckets that form the head of the structure, which is a power of 2 in size to enable efficient indexing. The format of each bucket is shown in the shaded section. A bucket contains a tag to check if the bucket matches the faulting virtual address. The tag is the same format as the ENTRYHI system-coprocessor register of the R4x00.

There are two page-table entries following the tag. These entries are placed immediately after the tag to minimise memory latency if the entries are not in the cache. Collisions in the HPT are resolved by following the link field, which is the next entry in the bucket. The link field is placed before the mapping-tree pointers to minimise memory latency when the link is followed in the case of a collision. The mapping-tree pointers play no part in TLB refill and are placed last in the bucket.

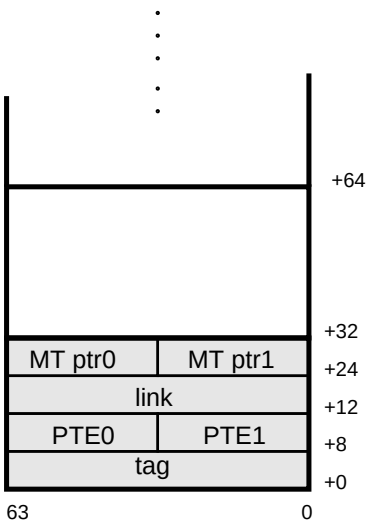


Figure 6.3: Hashed page table implementation.

Collisions in the HPT are resolved using external linear chaining, i.e the link field in the hash bucket forms the head of an external linear linked list. TLB refills resolved in the external list are promoted to the HPT head by swapping the tag, PTEs, and mapping-tree pointers between the matching external bucket and the bucket at the head of the chain in the HPT. This promotion technique is similar to one proposed for IPT-based page tables [RSD81] and exploits temporal locality of TLB misses.

The hash function used to select the initial bucket for a potential match is a simple bit selection of the faulting virtual address. Bit 13 and above are used[1], with the actual number of bits used depending on the HPT size. The two HPT sizes that were implemented and tested were an 8K HPT featuring 512 potential page-table entries (H512) in the hash table, and a 128K HPT with 8192 potential page-table entries (H8192). The hash tables are indexed with bits 20–13 and 24–13 respectively.

Kernel-mapped memory is handled via a second HPT of the same format that is shared between all address spaces. It is searched after a miss in the user HPT. This configuration is a result of the design semantics of sharing kernel PTEs between all tasks. Implementing the sharing as a second HPT provides a simple way to achieve sharing semantics. The kernel PTEs do not play a significant role in the benchmarks as the benchmarks test properties of user-level page tables.

6.1.3 Clustered Page Table

The major design decision to be made when building a clustered page table (CPT) is choice of the subblock factor. The subblock factor has an effect on memory consumption, and the average number of cache-lines accessed during page-table lookup. Increasing the subblock factor decreases memory consumption, assuming a high percentage of valid entries within a cluster. Table 6.1 shows the page-table bucket size for various subblock factors, assuming each subblock contains both a 64-bit tag and link, and n page table and mapping-tree entries (n being the subblock factor). The table shows a dramatic reduction in page-table size with increasing subblock factor. Subblock factors higher than 16 are not considered as the aim is to support sparse environments. Subblock factor 2 is considered the normal case given the MIPS R4x00 TLB is itself subblocked with a factor 2.

Considering memory consumption alone, the table argues for a large subblock factor. However, further considering caching characteristics limits choice of subblock factor. The cache-line size is an important factor in choosing subblock factor. The STLB experiment in Chapter 5 revealed the prevalence of data-cache misses during TLB refill. Access to extra memory within a cache line produced a significant performance penalty. Increasing the average number of cache-lines accessed will incur a greater penalty.

[1] Bits 12-0 are the offset within a page pair.

n	CS	ENCS	%SR
2	32	32	0
4	48	64	25
8	80	128	37.5
16	144	256	43.75

Table 6.1: Clustered (CS) and equivalent non-clustered (ENCS) page-table bucket size, and percentage size reduction (%SR) for various subblock factors (n).

Subblock factor	Average cache lines accessed
2	1
4	1.25
8	1.5
16	1.75

Table 6.2: Average number of cache lines accessed per TLB refill for various subblock factors.

The MIPS R4x00 has a 32-byte cache-line size. The average number of cache-lines accessed during TLB refill can be estimated with the assumptions of a hit in a hash-table bucket, the most compact layout of tag and PTEs, and equal probability of hitting each entry within a bucket. Note that two buckets need examining for this calculation. Bucket size is not a multiple of the cache-line size, so the second bucket in a pair is not aligned as well as the first. Refering to Figure 6.4, we see the average number of cache lines accessed for a hit (noting that a hit requires access to both the tag and PTE), for each of the page-table entries is $\frac{1+1+1+1+1+1+2+2}{8} = 1.25$. The second bucket begins in the middle of a cache line and is not as favourably aligned. The average number of cache line accessed for the second bucket is $\frac{1+1+2+2+2+2+2+2}{8} = 1.75$. Thus, the overall average number of cache line accessed is 1.5. Table 6.2 shows the average number of cache lines accessed per TLB refill for this, and other subblock factors.

Subblocking complicates the TLB-refill routine in two ways. The bucket size is no longer a power of 2, which complicates the calculation of an index for selecting a bucket, and an extra calculation is required to select a pair of entries within a bucket. The extra complexity is equal for subblock-factor 4, 8 and 16, and thus is not a consideration for subblock-factor selection.

When clustering the page table, an interesting question arises as to whether it

is worthwhile to promote hits in the collision chains to the hash table. Promotion from a collision chain to the hash table was relatively cheap for the HPT as both the overflow and hash-table bucket are in the cache as a result of the search. A CPT with buckets spread across multiple cache lines incurs a greater penalty as it can cause extra caches misses, in addition to copying a larger amount of data. There exists the possibility of thrashing, i.e. constant swapping between collision chains and the hash table. *Clustering may rely not only on clustered allocation, but also on clustered access patterns.*

To determine an appropriate subblock factor and investigate the applicability of promotion, the following clustered page tables were constructed.

C-4-512 A clustered page table with 128 buckets, each with 4 page-table entries. Hits in collision resolution chains are promoted to the hash table.

C-8-512 A clustered page table with 64 buckets, each with 8 page-table entries. Hits in collision resolution chains are promoted to the hash table.

C-16-512 A clustered page table with 32 buckets, each with 16 page-table entries. Hits in collision resolution chains are promoted to the hash table.

C-4-512-N A clustered page table with 128 buckets, each with 4 page-table entries. Hits in collision resolution chains are *not* promoted to the hash table.

C-8-512-N A clustered page table with 64 buckets, each with 8 page-table entries. Hits in collision resolution chains are *not* promoted to the hash table.

C-16-512-N A clustered page table with 32 buckets, each with 16 page-table entries. Hits in collision resolution chains are *not* promoted to the hash table.

The layout of a CPT hash bucket of subblock-factor 8 is shown in Figure 6.4. The other subblock-factor hash buckets are similar, except for the appropriate change in the number of page-table entries and mapping-tree entries.

The six page tables were tested using the TLB-refill benchmarks (described in Section 4.2.1) and compared using the normalised elapsed-time metric. The results are shown in Table 6.3.

Comparing the three subblock-factors tested, we see that subblock-factor 4 (C-4-512) performs similar to subblock-factor 8 and 16 (C-8-512 and C-16-512), with the exception of `mm` and `c4`. The C-4-512 CPT, on average, performs 2% better

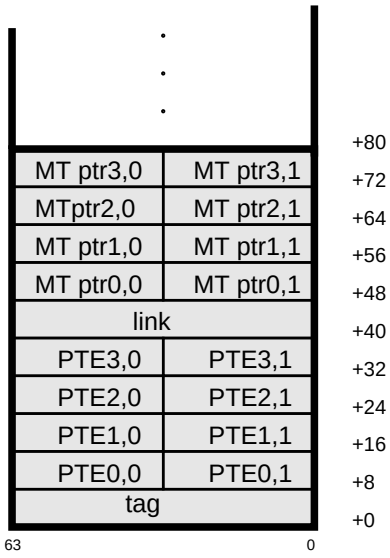


Figure 6.4: Clustered page table implementation for subblock factor 8.

BENCH	C-4-512	C-4-512-N	C-8-512	C-8-512-N	C-16-512	C-16-512-N
go	1.00	1.00	1.00	1.00	1.00	1.00
swim	1.00	1.00	1.00	1.00	1.01	1.00
gcc	1.00	0.96	0.99	0.97	1.00	0.98
compress	1.00	1.27	1.04	1.39	1.09	1.43
apsi	1.00	1.00	1.00	1.00	1.01	1.00
wave5	1.00	1.52	1.02	1.80	1.06	1.88
c4	1.00	0.85	1.16	0.89	1.50	0.92
nsieve	1.00	1.01	1.00	1.01	1.00	1.00
heapsort	1.00	1.00	1.00	1.00	1.00	1.00
mm	1.00	1.06	0.96	1.02	0.93	0.99
tfftdp	1.00	0.92	1.03	0.92	1.09	0.92
Average	1.00	1.05	1.02	1.09	1.06	1.10

Table 6.3: Results of TLB-refill benchmark for various CPT configurations. Results are normalised to the elapsed time of C-4-512.

than the subblock-factor-8 CPT, and 6% better than the subblock-factor-16 CPT. Examining the two notable exceptions, we see that `c4` performs significantly worse with the larger subblock factors, and `mm` performs significantly better. `c4` plays the game of "connect four" and has a sparse memory access pattern, which results in significant amounts of thrashing of bucket contents between collision chains and the hash table. This analysis is confirmed by comparing these results with `c4`'s results without promotion (C-4-512-N, C-8-512-N, and C-16-512-N). Without promotion (i.e. without copying), we see that `c4` performs significantly better.

`mm` performs better with a higher subblock factor. `mm` is a inner-product matrix multiply. The higher subblock-factor page tables prefetch more page-table entries into the hash table that are subsequently used, which results in lower average TLB-refill cost.

The results reveal, on average, that promotion pays off. In particular, promotion is advantageous for both `wave5` and `compress`, which are the two largest benchmarks, and thus have the longest collision-resolution chains. It is likely that promotion will continue to pay off as recent microprocessors feature larger cache lines, thus reducing the promotion costs.

For comparison with the other page tables, I chose C-4-512 as being representative of a clustered page table on the MIPS architecture, it is 6K in size. In addition, a larger clustered page table was constructed (C8192) to compare with the larger HPT. C8192 features 2048 hash buckets, each containing four page-table entries and is 96K in size. From here on in the text, the C-4-512 CPT is referred to simply as C512.

6.2 Benchmarks

The benchmarks used to evaluate the various page-table types are those introduced previously. The benchmarks test TLB-refill performance (Section 4.2.1), memory overhead (Section 4.3.2), (un)mapping performance (Section 4.4.1), and task creation-and-deletion performance (Section 4.5.1). A brief summary of each benchmark follows.

The metrics used to compare TLB-refill performance are application runtime and average TLB-refill cost obtained via instrumentation. Each of the applications used for testing are run three times each, both with and without instrumentation.

Page-table memory overhead is examined in three situations. The first scenario is with the conventional applications, the second with a sparse address space (SPARSE-PAGE), and the third with a clustered address space (SPARSE-FILE). At the end of each benchmark run, the page-table size is recorded together with the number of pages mapped in the address space.

The (un)mapping speed is tested in three scenarios via the MAP1, MAPN, and MAPS benchmarks. They respectively test individual page, region, and sparsely-populated-region (un)mapping. These benchmarks are also run with the addition of a compact 64-megabyte region of mapped memory elsewhere in the address space. The memory is unrelated to the region undergoing testing. The motivation for this is to compare page tables under a more realistic scenario. The hash-based page tables are expected to contain entries from other tasks. The added region increased the loading in the hash-based page tables to a level approximately equivalent to that expected with 100% utilisation of physical memory.

The task benchmark measures the cost of creating and destroying a task. The benchmark measures the elapsed time of 100 iterations of:

- create a child task,
- wait for null IPC from child, and
- delete the child.

6.3 Results

6.3.1 TLB Refill

Before analysing the results, the predicted cost (in cycles) of each of the TLB-refill routines is shown in Table 6.4. These numbers represent the minimum TLB-refill cost of each routine, but does not include processor exception and restart costs. It assumes all loads from memory hit in the cache, and either a 100% hit ratio in the STLB, or a 100% front-bucket hit ratio in the HPT and CPT.

The average TLB-refill cost results for the conventional applications are shown in Figure 6.5. The G16+S16384 has the lowest average TLB-refill cost of 31.5 cycles. The second best performer is the H8192 with an average refill cost of 37.3 cycles. This is an interesting result as both page tables are expected to perform near

Page Table	Cycles
STLB	19
HPT	19
CPT	27
MPT	34

Table 6.4: Theoretical minimum TLB-refill cost (in cycles) assuming cache hits, for each of the TLB-refill routines under test.

their best of 19 cycles each, as both have a high hit ratio in either the STLB or the hash table. The difference between the 31.5 cycles of the STLB and the theoretical cost of 19 cycles is attributable to a poor cache hit ratio in the STLB.

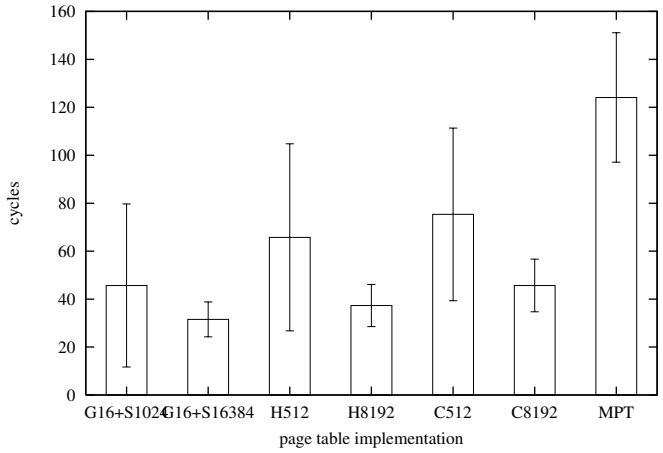


Figure 6.5: Average TLB refill time (cycles) across conventional benchmarks. Error bars represent standard deviations.

The most interesting result is the difference between the average refill cost of the HPT and STLB. Both routines have the same best-case refill cost, and are tested in a near ideal situation. Thus one could expect they perform similarly. The surprisingly large difference in cost can be explained by considering the data-structure density and its effect on cache performance. The STLB is a denser structure with two PTE pairs contained in each cache line. The HPT has a single PTE-pair per cache line, together with data unrelated to TLB refill (mapping-tree pointers, and a link for collision resolution). The density difference favours the STLB for two reasons:

- A TLB refill which results in a cache miss will load the required entry into the

cache together with a neighbouring STLB entry. The hash function used for the STLB is such that spatial locality in the virtual address space translates to spatial locality in the STLB. TLB refills for applications exhibiting spatial locality will pre-fetch and prime the cache for future TLB refills.

- The more compact structure of the STLB allows more entries to fit in a given cache size, increasing the likelihood of entries being in the cache.

The CPT has a best-case TLB-refill cost of 27 cycles, 8 more cycles than both the STLB and HPT page table. Recall that the CPT is 25% more compact in terms of cache lines used, compared to the HPT, but on average accesses 1.25 cache-lines per refill. The reduction in cache usage is expected to improve TLB-refill performance. The clustering is also expected to improve TLB-refill performance via the cache priming effect previously described. The increase in average number of cache-lines accessed per refill is expected to reduce refill performance.

The C8192 average TLB-refill cost was 45.7 cycles, 8.4 cycles more than the average cost for H8192. The difference is slightly more than the difference of 8 cycles expected by comparing instruction counts with the HPT. The result indicates the performance improvements obtained by clustering approximately balances the penalties of clustering. The performance difference between the HPT and CPT comes down to the basic difference in number of instruction cycles needed to execute the refill handler.

The G16+S1024, H512, and C512 all feature higher average TLB-refill costs (46, 66, and 75 cycles respectively) than their larger counter parts. This is attributable to all three implementations having higher first-probe miss ratios due to their smaller hash-table span, when compared to the larger page tables. However for the same hash size, the STLB still performs better than the HPT, which in turn performs better than the CPT.

The MPT has a basic cost of 41 cycles per refill without cache misses, plus it consistently makes 6 memory references per refill. The MPT in this comparison averaged 124 cycles per refill which is poor TLB-refill performance compared to the hash-based schemes. The MPT simply accesses too much memory to be competitive.

Table 6.5 shows the normalised elapsed time for each combination of conventional application and page table. The results are normalised to the elapsed time of the G16+S16384 page table to enable comparison between runs of different lengths.

The table shows the varying contribution of page-table TLB-refill overhead to overall application runtime. The table re-affirms the previous results that on average, the G16+S16384 performs the best followed by H8192, C8192, G16+S1024, H512, C512, and the MPT.

BENCH	G16+S128K	G16+S8K	MPT	H512	H8192	C512	C8192
go	1.00	1.00	1.03	1.00	1.00	1.02	1.02
swim	1.00	1.00	1.00	1.00	1.00	1.00	1.00
gcc	1.00	1.08	1.19	1.15	1.02	1.14	1.03
compress	1.00	1.00	1.10	1.04	1.01	1.05	1.01
apsi	1.00	1.00	1.04	1.00	1.00	1.01	1.01
wave5	1.00	1.00	1.10	1.07	1.01	1.06	1.01
c4	1.00	1.10	1.44	1.18	1.02	1.38	1.10
nsieve	1.00	1.00	1.08	1.01	1.00	1.01	1.00
heapsort	1.00	1.00	1.01	1.00	1.00	1.00	1.00
mm	1.00	1.01	1.58	1.06	1.05	1.03	1.03
tfftdp	1.00	1.00	1.26	1.06	1.01	1.14	1.01
Average	1.00	1.02	1.17	1.05	1.01	1.08	1.02

Table 6.5: Elapsed times normalised to G16+S16384 for each combination of application and page table.

There are some exceptional individual results worth noting. The CPT has the best result for the `mm` application. As described previously, `mm` is a inner-product matrix multiply which exhibits high spatial locality. Clustering of page-table entries is especially advantageous in this scenario as loading multiple entries effectively prefetches entries into the cache, which amortises the cache-miss cost over several TLB refills.

For the small CPT, which has a higher collision ratio than the large CPT, clustering is also advantageous for applications exhibiting spatial locality for another reason. The promotion of entire blocks of PTEs to the hash table from collision resolution chains has the effect of pre-fetching entries to less costly position (in the page table) for refill. This amortises the cost of hash-collision resolution over multiple TLB refills. This phenomenon only occurred for `wave5`.

Unexpectedly, `gcc` also performed better with C512 than with H512. This result is unexpected because in Table 6.3 `gcc` was shown to perform better without promotion than with, which implies sparse access patterns. (Previous TLB studies

[CBJ92] confirm `gcc`'s sparse access patterns.) The H512 should perform better in this situation due to cheaper promotion costs. C512's better performance can be attributed to both page tables having high front-bucket miss ratios, and thus the shorter collision chain lengths of the CPT are an advantage.

As pointed out in Section 6.1.3, the CPT performed poorly for `c4` due its very sparse memory-access patterns, which causes thrashing in the collision chains. In general, the elapsed-time metric confirms that clustering was disadvantageous compared to the simpler HPT.

6.3.2 Page-Table Size

Figure 6.6 shows the page-table memory overhead for the conventional applications. The values illustrated are normalised, i.e. they represent the total page-table size divided by the number of mapped 4K pages. Each page-table configuration has three values associated with it: the lowest, the average, and the highest observed overhead for the applications tested. Note that the STLB is considered kernel-data shared between all tasks, and is not factored into the size overhead for the G16+STLB.

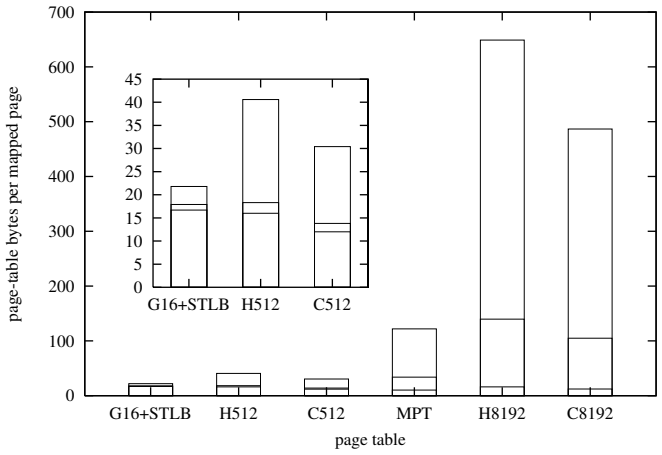


Figure 6.6: Page-table bytes per mapped page for conventional applications. The three values represent the highest, average, and lowest page table size overhead observed in the conventional applications.

The results show that the small CPT (C512) has the lowest average memory overhead of 13.8 bytes per mapped page. The low average overhead is explained

by the small initial page-table size. The initial page table only supports a maximum of 512 page-table entries before overflow (collision resolution) chains are required. The 512 entries correspond to 2 megabytes of virtual address space, which is a size all the applications exceed except `go`. Hence for the conventional applications, the 6K CPT is densely populated and exhibits the least memory overhead.

Similarly, the small HPT (H512) also exhibits a low memory overhead. It too is densely populated and has an overhead of 18.3 bytes per mapped page.

The GPT-based page table (G16+STLB) has an average overhead of 17.9 bytes per mapped page, which places the GPT between the CPT and HPT. It also features the least variation in memory overhead, only varying 5 bytes per mapped page across all the applications.

The MPT average memory overhead was 33.7 bytes per mapped page. The MPT has the potential to be the most compact structure if it is densely populated. It featured the lowest observed overhead of 10.3 bytes per mapped page for one of the benchmark runs. However, for most of the benchmark runs the population density was such that the MPT has higher overhead than the G16+STLB, H512, and C512.

The memory overhead for both the H8192 and C8192 page tables varied greatly. These large structures need a large number of page-table entries to populate them densely. With a sufficient number of entries they have low overhead as shown by the lowest observed overhead of 16.1 and 12.1 bytes per mapped page for the H8192 and C8192 respectively. However, the highest observed overhead of 648.9 and 486.7 bytes per mapped page illustrates what can happen when a large page table is populated by a low number of valid entries. The average overhead for the applications tested was 139 and 104.8 bytes per mapped page, which is significantly higher than the other page table configurations.

Figure 6.7 shows the results for the SPARSE-PAGE benchmark. The figure plots the normalised memory overhead versus number of pages allocated, for each of the page-table configurations. The results reveal the HPT as being the most space efficient, using approximately 33-50 bytes per mapped page, but only when the number of page-table entries is sufficient to populate a good proportion of the available slots in the hash table (512 for the H512, 8192 for the H8192). The H8192, and to a lesser extent the H512, become very space inefficient when partially populated.

The SPARSE-PAGE benchmark does not cluster mapped pages in any way, which is contrary to Talluri's assumption when he proposed the CPT as a space-efficient

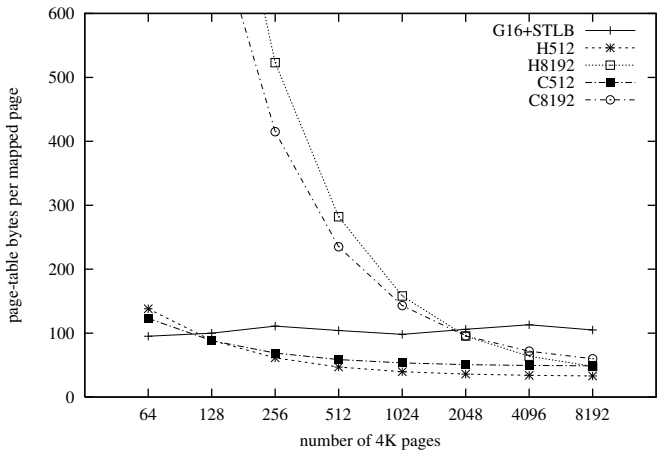


Figure 6.7: Normalised page-table memory overhead for SPARSE-PAGE benchmark for each page-table implementation.

optimisation. When mapping single pages, clustering allocates multiple invalid PTEs together with a single valid PTE within a hash bucket. The CPT follows a similar trend to the HPT, except clustering increases the memory overhead when significant amounts of collision chains are present, i.e. when the number of mapped pages grows beyond 128 for C512, and 2048 for C8192. Prior to these points, the smaller initial page-table size gives clustering a slight advantage. When the CPT is populated, it has an overhead of 50-100 bytes per mapped page. Like the HPT, it becomes very space inefficient when significants amounts of available hash buckets are left unused.

The G16+STLB has the same result as when examined in Chapter 4, it exhibits a relatively stable overhead of 95-115 bytes per mapped page. The *spill-over* effect does not influence a G16 GPT significantly, its memory overhead can be considered independent of the number of entries mapped.

The MPT exhibited a memory overhead of 5–8 kilobytes per mapped page. The result for the MPT is not shown in the figure to avoid having a scale on the y-axis which failed to adequately illustrate the results for the other page tables. The MPT performs extremely poorly in this sparse benchmark.

Figure 6.8 illustrates the results of the SPARSE-FILE benchmark. The figure plots the normalised memory overhead versus number of objects allocated, for each of the page-table configurations. The object size is selected randomly from the file-

size distribution used previously (see Figure 4.5).

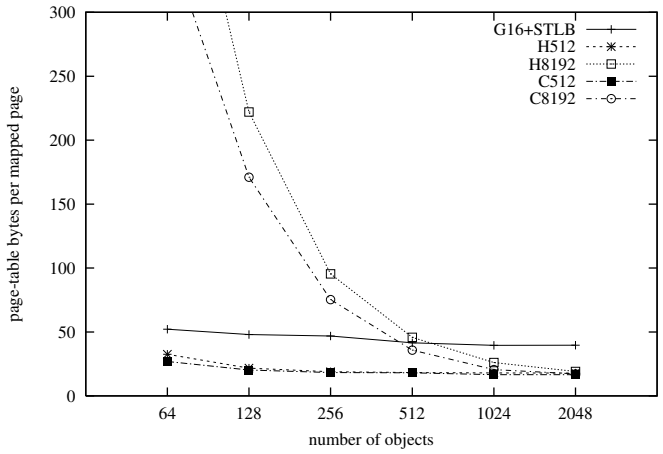


Figure 6.8: Normalised page-table memory overhead for SPARSE-FILE benchmark for each page-table implementation.

In this scenario, a densely populated HPT exhibits a memory overhead of 17–19 bytes per mapped page. The clustering of pages into objects approximately halves the overhead due to the hash buckets containing pairs of PTEs. The clustering also causes a slightly greater reduction in memory overhead of the CPT. When densely populated, it has an approximate overhead of 16–18 bytes per mapped page. This is marginally better than the HPT. Both hash-based page-tables (the HPT and CPT) still exhibit high overheads when partially populated, with the CPT having the advantage of a smaller initial size.

The G16+STLB configuration has an average memory overhead of 39–52 bytes per mapped page. This clustered scenario approximately halves the overhead compared to the SPARSE-PAGE benchmark. Once again, the GPT features relatively constant overhead that is independent of the number of pages mapped.

The MPT also features a reduction in overhead down to 720–1800 bytes per mapped page. This is still 1–2 orders of magnitude above the other configurations.

To summarise, for conventional applications the GPT exhibited stable memory overhead competitive with a densely populated HPT, and slightly worse than the densely populated CPT. Both the HPT and CPT have low memory overhead when their size is well matched to virtual memory consumption. They perform poorly when they are not matched. The MPT exhibited approximately twice the overhead

of the GPT on average. For the sparse benchmarks, the HPT again performed well when its size was appropriately matched with virtual memory consumption. The CPT performed similarly except for exhibiting higher memory overhead for SPARSE-PAGE which maps isolated pages. The memory consumption of the GPT was approximately twice that of a well matched HPT. The GPT exhibited consistent and stable memory overhead independent of virtual memory consumption. The memory overhead of the MPT was 1–2 orders of magnitude worse than the other structures.

6.3.3 Mapping Performance

The results for the MAP1 and MAPN benchmarks are shown in Figure 6.9. For the MAP1 benchmark, C8192 performs the best (costing 13.9 microseconds per iteration), closely followed by H8192 (costing 14.3 microseconds per iteration). Both these large, hashed-based page tables have negligible overflow chaining. Mapping and unmapping involves a simple hash-table probe and page-table manipulation. Comparing H8192 (and C8192) with H512 (and C512), we see that the addition of overflow chaining has a significant detrimental effect on mapping performance. H512 costs 21.3 microseconds per iteration; C512 costs 19.1 microseconds per iteration. An increase of 49% for H512 and 37% for C512. The result shows the advantage of clustered page tables when hash table overflows, because of the reduced search time due to shorter collision chains.

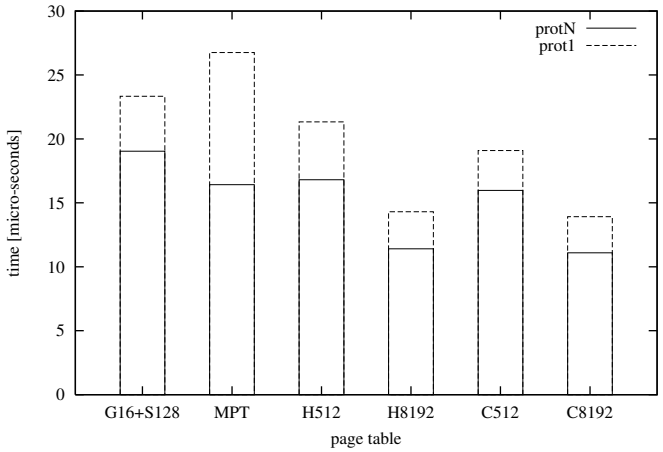


Figure 6.9: Normalised mapping speed for MAP1 and MAPN.

The MPT has the highest MAP1 benchmark cost of 26.8 microseconds per iteration. The high cost is due to the MPT needing to traverse the 6 levels of the page table for each map operation. The G16+S16384 cost is 23.3 microseconds per iteration. G16+S16384 performs better than the MPT as it has less levels to traverse for each map operation. Compared to the hash-based structures, G16+S16384 performs worse due to either the extra lookup complexity (compared to H512 and C512), or the need to traverse more levels (compared to H8192 and C8192).

The MAPN results show the improvement possible when the unmap operation is batched into a single system call, rather than unmapping a single page at a time as in MAP1. All the results improve similarly except for the MPT. The MPT's dramatic cost reduction is due to its efficient processing of page-table regions. For the MPT, processing regions is a simple array scan. Processing regions with the hashed-based structures involves probing for each potential entry in the region (or each potential block for the CPT's case). Theoretically, the GPT-based structure should also perform operations on regions efficiently. Yet the results show a similar cost reduction to the hash-based page tables. As pointed out earlier in Section 4.4.2, the tested GPT implementation performs the operations on regions by page-table scanning, not by operating on the root node of the region. This is due to both the need to check for mappings derived from the pages being operated on (i.e. an L4 specific constraint), and the need to indentify which pages are operated on to subsequently ensure the STLB is kept consistent. Hence, the advantage of in-order traversal shown by the MPT is not visible in the GPT in this benchmark. The software complexity (cost) of traversal in the GPT implementation is similar to that of traversing the hash-based page tables by probing for each entry in the region. However, the cost of probing greatly depends on the length of collision chains in the hash-based page tables. The implications of this are demonstrated next in the MAPS benchmark.

Figure 6.10 shows the results for the MAPS benchmark. It shows the cost of unmapping a region normalised to a per unmapped-page cost, versus the unmapped region size, for each of the page-table configurations. The results show that operations on sparsely populated regions have significantly different costs for the different page-table configurations.

The results reveal the MPT has the highest cost in sparse environments due to it needing larger page tables to represent a sparse region, and thus the unmap

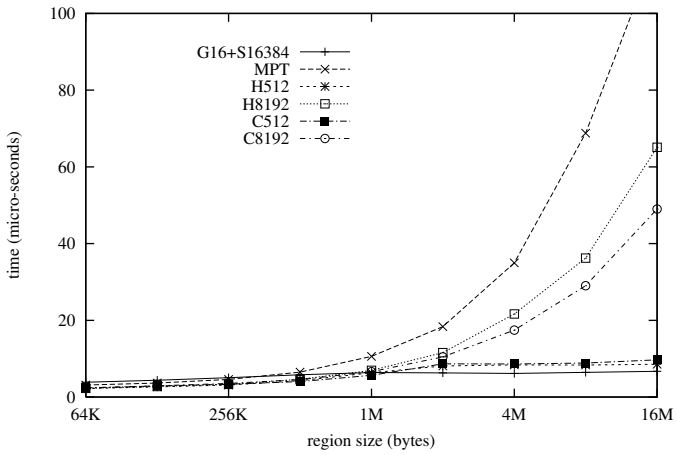


Figure 6.10: Average unmap time for regions of 64K to 16M in size, containing 16 randomly placed pages, for each page table implementation.

code spends more time processing mostly invalid page-table entries in the region. The GPT, H512 and C512 configurations all show relatively constant unmap cost as sparsity varies. However, the H8192 and C8192 both show a dependence on region size. This is a consequence of the unmap algorithm needing to scan a larger proportion of the page table as the region size increases. The smaller HPT and CPT does not exhibit this effect as the unmap algorithm is scanning the entire page table at low sparsity, giving the appearance of constant unmap cost as sparsity increases.

Figure 6.11 shows the results for the same benchmark except that the page table is populated with entries for 64M of memory that is unrelated to the region under test. The results for this show that the performance of the two hierarchical page tables (the GPT and MPT) remains unchanged. The GPT exhibits a constant low overhead independent of the sparsity of the unmap operation, and the MPT remains expensive for sparse operations.

The benchmark results for the HPT and CPT shows that their performance degrades significantly compared to the previous mapping benchmark which did not have the additional mapped memory. The degradation is due to the scanning of the additional memories' page-table entries, even though they are unrelated to the memory region undergoing testing. The large configurations are least affected as the additional page-table entries basically fill-in the invalid entries in the sparsely populated page table with a low amount of additional collision resolution chaining.

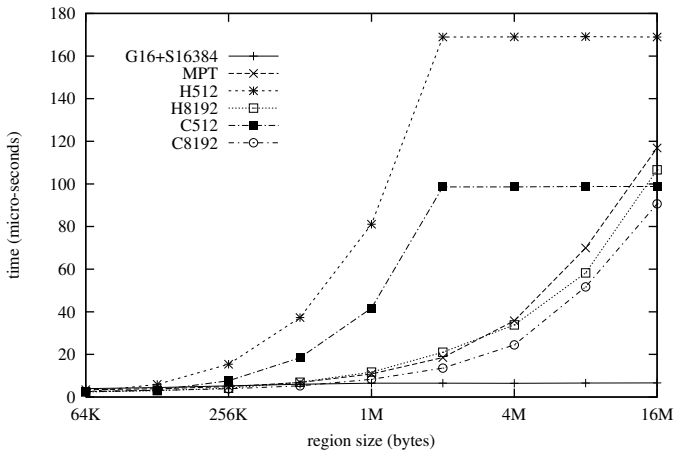


Figure 6.11: Average unmap time for regions of 64K to 16M in size, containing 16 randomly placed pages and an unrelated 64M segment, for each page table implementation.

Thus, for H8192 and C8192, the mapping benchmark does not scan dramatically more page-table entries compared to the previous benchmark. However, H8192 and C8192 remain significantly more expensive than the GPT.

The small hash-based page table (H512 and C512) results are dramatically affected by the additional memory. The page-table entries needed to map the additional memory produce collision resolution chaining. The chains add to the scanning overhead of both densely and sparsely populated regions. This overhead produces the dramatic increase in unmap time as the region size increases. The unmap cost levels off at the point where the entire page table is scanned. This is an important result as it approximates what one can expect in a real system. A task can easily have 64M of memory unrelated to mapping operations, and, more importantly, the benchmark approximates what happens when the hash-based page tables are shared among all tasks, which adds the PTEs of all other tasks to the page table. The large page tables would exhibit the same effect if the unmapped region size was increased.

Comparing the HPT and CPT results, we see that for sparsely populated regions, or regions involving scanning page tables containing significant amounts of unrelated data, the CPT performs better as it compares fewer tags while scanning the same amount of page-table entries.

6.3.4 Task Creation and Deletion

The result of the task benchmark is shown in Figure 6.12. The results show the G16+STLB, H512, and C512 page-table configurations being dramatically faster than the H8192, C8192 and MPT configurations. The G16+STLB configuration took 160 microseconds per iteration of the benchmark. This low overhead can be attributed to several factors: the low GPT initialisation overhead, as a single small root node is initialised; the low cost of incrementally adding a mapping; and the cost of tearing down the GPT being directly related to the number of mappings.

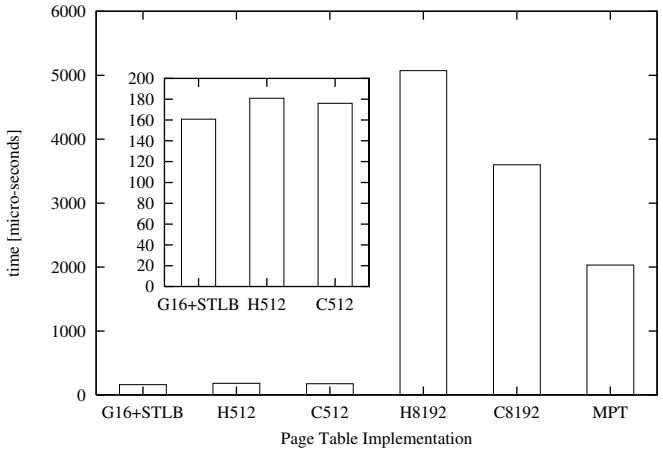


Figure 6.12: Task creation and destruction cost for each page table implementation.

The H512 and C512 took 181 and 176 microseconds per iteration respectively. These configurations have low overhead as they are of modest size and thus have modest initialisation overhead, they have low cost of incrementally adding a mapping, and they have modest demolition cost. These 512-entry ($\approx$ 8K) configurations can be held entirely in the cache, which in this repetitive benchmark favours the smaller configurations compared to the 8192-entry ($\approx$ 100K) configurations. One could expect a significant degradation in these numbers when the page-table size is increased beyond the 16K cache size. Indeed, the H8192 and C8192 configurations show proportionally higher overhead with 5 milliseconds and 3.6 milliseconds per benchmark iteration, respectively.

When comparing the HPT and CPT we see that clustering improves performance. Clustering reduces page table overhead by reducing the initialisation and demolition overhead. A clustered page table has less tags for the same number of

PTEs. Given initialising a single tag to invalid indirectly initialises the group of PTEs associated with it, clustering reduces the amount of work needed to put the page table into a known state. Demolition costs are reduced as tear-down involves scanning the entire page table, which also involves less operations due to fewer tags.

The MPT performed a benchmark iteration in 2 milliseconds. This is a much higher overhead than that exhibited by the GPT, small HPT, and small CPT. The overhead can be attributed to the cost of initialising and subsequently tearing down the 6 levels in the page-table tree.

6.4 Discussion

6.4.1 Sharing the HPT or CPT

All of the performed benchmarks primarily consist of a single task. This has the effect of making the memory overhead of the HPT and CPT sensitive to the benchmark size. In a multi-user multi-tasking system, the HPT (or CPT) is shared between many tasks. This would ensure utilisation of the HPT (or CPT) if it is sized relative to available physical memory. Under-utilisation (high memory overhead) only occurs if physical memory is under-utilised; in this case, memory overhead becomes a non-issue. However, sharing an HPT (or CPT) amongst many tasks will have an effect on the other examined areas of page-table performance. The following is a qualitative look at a sharing the HPT or CPT, describing the expected effect on the various areas examined.

TLB Refill – The average TLB-refill cost is expected to increase with sharing. This is a consequence of storing page-table entries of other tasks in the same data structure. This increases the length of collision resolution chains resulting in longer lookup times.

ASID Management – Like the STLB in Section 5.2.5, a shared HPT (or CPT) has either to maintain its own ASIDs independent of those supplied by the hardware, or use hardware based ASIDs and be encumbered with ASID management problems. Using software based ASIDs independent of the hardware has the advantage of not encountering the problem of what to do when ASIDs

become unavailable and the ensuing address-space switch complexity. Software based ASIDs can be designed such that unavailability never eventuates. However, software based ASIDs complicate TLB refill by requiring some translation between ASIDs used by the hardware and the ASIDs stored in the page table. This complexity reduces TLB-refill performance.

Using hardware derived ASIDs in the page table avoids ASID related complexity in the TLB-refill routine, but when the number of address spaces exceeds available ASIDs, it has a side effect of potentially expensive ASID management costs. The ASID reclamation costs for a HPT or CPT will be much higher than a STLB as the HPT contains all page-table entries, the STLB contains only a subset.

Mapping Performance – Sharing the HPT (or CPT) will have a detrimental effect on mapping performance. The results in Section 6.3.3 showed the adverse effect on mapping performance of adding unrelated page-table entries to the page table. Sharing the HPT will have similar, if not dramatic, effect of reducing mapping performance.

Task Creation and Deletion – Sharing a single HPT or CPT will also have a detrimental effect on task creation-and-deletion overhead compared to a per task data structure. Sharing adds the PTEs of all unrelated tasks to the data structure. The scanning of all these additional unrelated PTEs will increase task creation-and-deletion costs.

6.4.2 Linear Virtual Arrays

As described earlier in this chapter, linear virtual arrays (LVA) were excluded from this comparison because of implementational limitations. It is not clear how LVAs would perform in terms of TLB-refill performance when compared to the other tested page tables. Earlier MIPS processors featured refill times of 10–14 cycles [DMM86]. The minimum refill cost being 10 cycles if everything needed is in the cache. These cycle times were for an 8MHz processor, a dramatically slower processor than the one used in this thesis, and more importantly a processor with a lower ratio of CPU speed to memory speed. Translating the above numbers by applying a similar memory reference penalty to the one observed for the STLB (12.5 cycles) gives an estimated refill time of 22.5 cycles on average. This assumes no

TLB misses in the LVA itself, which is likely to have a significant impact, especially in sparse allocated environments or for applications exhibiting sparse access patterns [NUS+93].

Sparse environments will be a problem for LVAs when one considers memory consumption. An LVA can be thought of as a multilevel page table searched from the bottom up instead of top down. An LVA will have the same memory consumption characteristics as the MPT tested. The results for the MPT revealed it was unusable in sparsely allocated environments.

The LVA will be competitive for traditional contiguous 32-bit environments, i.e. enviroments encouraging low levels of cascading TLB misses and low memory overhead. However, the LVA will be unsuitable for large, sparse address spaces.

6.5 Conclusions

6.5.1 TLB Refill

A tagged, shared STLB is the best choice for good TLB-refill performance. It has been shown to outperform all other page-table implementations in this thesis. Compared to the HPT, the STLB performs better as it has no collision overflow link, which leads to a more compact page-table structure. The compact structure allows more page-table entries to fit in a cache line, thus resulting in more page-table entries being pre-fetched when the cache line is loaded to service a TLB miss.

The cache priming (pre-fetching) effect is strong, and indicates that *neighbouring entries within a cache line should be as closely related as possible* in order to maximise the likelihood that pre-fetched entries will be used. Applications tend to exhibit spatial locality in their address space, hence the STLB hash function should translate spatial locality within the virtual address space into spatial locality within a cache line. The pre-fetching effect will become increasingly important as the gap widens between CPU and memory speed.

Clustered page tables aim to take advantage of the same cache priming effect. However, clustered page tables did not perform as well as the STLB. The cache-line size on the R4700 is only 32 bytes, which means a clustered block of page-table entries is spread across multiple cache-lines. The spanning of cache lines has several disadvantages:

- Spanning increases the average cache-lines referenced per TLB refill.
- Spanning makes promotion of collision-chain blocks to the hash table more expensive. Promotion is an important performance optimisation. Expensive promotion makes clustered page tables susceptible to costly thrashing if accesses are sparse. Thus clustered page tables rely on clustered access patterns in addition clustered memory allocation.

In a software-loaded TLB, clustering has the additional disadvantage of adding significant extra complexity to the TLB-refill handler. The advantages of clustered page tables are more likely to manifest themselves with larger cache-lines. Clustering would be particularly attractive when used in conjunction with multiple page sizes as proposed by Talluri [Tal95].

The cache priming effect for page tables was alluded to in [Tal95], but not demonstrated due to the simulation methodology. To my knowledge, this thesis is the first to demonstrate the strength of the effect on a real system.

For TLB-refill, the MPT performed worse than the other page tables tested due to the depth of the tree.

6.5.2 Page Table Size

The small clustered page table (C512) featured the lowest average memory overhead for conventional applications. It shows what is achievable when a clustered page table is well matched to virtual memory consumption. The small HPT (H512) and GPT both have slightly worse memory overhead when compared to the C512. However, the GPT overhead was stable and relatively independent of virtual memory consumption. Like the CPT, the HPT featured low overhead when well matched to virtual memory consumption. Both the CPT and HPT featured high memory overhead when mismatched. The MPT exhibited an average memory overhead of approximately twice that of C512, G16, and H512.

In sparse environments, the small CPT and HPT feature the lowest overheads with the HPT more suited to really sparse address spaces, and the CPT more suited to clustered address spaces. Again, both page tables have to be well matched to virtual address space consumption, otherwise they exhibit high memory overhead. The GPT has a memory overhead of approximately twice that of the small CPT and HPT, with the overhead again being independent of the number of pages mapped.

The MPT is unusable in sparse environments. It has a memory overhead 1 – 2 orders of magnitude above the hash-based and guarded page tables.

To summarise, if virtual memory consumption is known and consistent, the CPT features the lowest memory overhead. If virtual memory consumption is unknown or inconsistent, the GPT is the better choice.

6.5.3 Mapping Performance

For manipulating single pages in contiguous regions, the hash-based page tables feature the lowest cost of mapping and unmapping, with the CPT performing better than the HPT due to shorter collision chains. The GPT has higher overhead (approximately 20%) when compared to the smaller hash-based structures. The MPT is about 20% more costly than the GPT. The situation is similar when manipulating contiguous regions, the hash-based structures are fastest, however the MPT shows a greater improvement in performance when operating on regions due to its efficient page-table traversal. The MPT performs better than the GPT in this case. Note that the GPT has the potential to support extremely fast power-of-2 region-based operations by manipulating internal nodes of the GPT tree, however the structure of the microkernel prevented such an optimisation.

When operating on sparsely populated regions, the situation changes dramatically. The GPT performs significantly better than all the other page tables. It features consistently low overhead that is related to the number of manipulations needed in the region, not the region size itself, nor the page table size. All the other page tables have much higher costs. The GPTs are the better choice for manipulating sparse populated regions.

6.5.4 Task Performance

Page table size has a significant effect on setup and tear-down costs of tasks. The GPT featured the lowest task overhead as it has a small initial structure with low cost of incrementally adding entries. It also features tear-down costs proportional to the number of mapping present in the page table. The small hash-based structures also feature low overhead as they are also small structures, however their tear-down costs a proportion to the table size, not the number of mappings. When the table size increases, so do tear-down costs. Tear-down costs also increase when sharing

the page table among other tasks in the system. Of the two hash-based structures, the CPT performs better than the HPT. The MPT is also a large structure the has significant setup and tear-down costs.

6.5.5 Overall Performance

The examination of various facets of page-table performance revealed and confirmed that each page table has its own weak and strong points. No page table was a clear winner in every category, though the MPT was a clear loser in nearly all cases. Applications with different performance demands would be best served by different page tables optimised for each application. However, it is not always possible to tune the page table to the expected demands of all relevant applications. The goal of this thesis is to find a solution that makes no "hard" assumptions about application behaviour. With this in mind the GPT+STLB combination is demonstrably the best choice.

The STLB features the best TLB-refill performance when it achieves a high hit ratio. The results show that a high hit ratio is achievable, even for modest size STLBs. Given the STLB, some other data structure is needed to hold the remaining page table entries. A GPT is the best choice as it performs nearly as well as the best page table in some of the categories examined, and significantly better in the remaining categories, especially for operations on sparse regions.

Chapter 7

Conclusion

Virtual memory is an important feature of operating systems. For applications concurrently sharing a single machine, virtual memory provides them with their own protected address space, which enhances security and reliability. Virtual memory also provides for application transparent relocation of programs within physical memory, and between physical memory and other forms of data storage such as disk storage.

Virtual memory is not free to applications, it does come with a cost. In typical virtual memory systems, the page table is a major component that affects the cost of virtual memory to applications. This thesis has examined the effect page-table selection has on virtual memory cost to applications in a 64-bit microkernel environment. The page table not only affects the usually examined areas of TLB-refill performance and kernel memory consumption, but also has a significant effect on kernel mapping primitives, and task creation and destruction overheads.

The 64-bit microkernel environment is particularly challenging for implementing efficient virtual memory. The microkernel only provides the basic virtual-memory primitives to application servers, which in turn present higher-level virtual-memory abstractions to their clients. The kernel itself can make very few assumptions about the behaviour of applications, and thus, the use of virtual memory. The microkernel should not make assumptions about application behaviour for it to be universally applicable. Ideally, the kernel would efficiently support both a conventional system such as Linux, and an unconventional system such as the Mungi single-address-space system, or even both concurrently.

This thesis examines each facet of a page-tables cost to applications: TLB-

refill performance, kernel memory consumption, kernel mapping primitives, and task creation and destruction. It uses benchmarks running on real page-table implementations, and uses real time as a metric for comparison. It does not use trace- or trap-driven simulation which only approximate actual performance. The cache priming effect, quantified in Section 6.3, would not be visible when using metrics such as average cache-lines accessed.

Guarded page tables were one of the page tables examined. There is much theory about guarded page tables. However, the theory is mostly concerned with satisfactory worst-case behaviour. This thesis provides a practical evaluation of guarded page tables. It shows that guarded page tables generally exhibit much lower size overhead than worst case, even in extremely sparse address spaces. The *spill-over* effect was also identified. *Spill-over* theory can be used to explain the behaviour of GPTs in some scenarios. It predicts situations of high memory overhead and high (un)mapping cost.

A detailed examination of software TLBs is presented. The examination confirmed the STLB's ability to improve TLB-refill performance, and it is shown that an STLB adds little to the cost of virtual memory to applications in the areas of memory consumption, (un)mapping performance, and task creation and destruction overhead.

A comparison of most major page-table types is performed. Guarded page tables with a software TLB are compared with implementations of clustered, multilevel, and hashed page tables. The results show that a combination of a medium node size GPT with a STLB performs significantly better in the areas of TLB-refill, operations on sparse regions, and task overhead. In the remaining areas examined, the combination performed satisfactorily.

The comparison between the hash-based page tables shows the importance of the cache priming effect. The cache priming effect argues strongly for neighbouring page-table entries in a cache line to be as strongly related as possible.

Clustered page tables failed to live up to expectation in the test environment. A combination of a software-loaded TLB and a small cache-line size worked against them. They would be expected to perform better on a processor architecture with a larger cache-line size, and multiple instructions issued concurrently, especially if they are used to support multiple page sizes. The value of the promotion of matching entries in collision resolution chains to the hash table was demonstrated.

However, the small cache-line size made promotion more expensive than in the non-clustered case. More expensive promotion revealed that sparse access patterns can *thrash* a block between the collision resolution chain and the hash table. Thus, clustered page tables that span multiple cache lines not only rely on clustered allocation, but on clustered access patterns as well.

The implications of these results are that a hardware-refilled STLB is a good choice for implementing virtual memory. The results show that the hash function used to access the STLB should preserve the spatial locality of an address space among the page-table entries within a cache line. External to the cache line, the hash function is free to implement whatever function that minimises conflict misses in the STLB. A clustered STLB could be used to support two page sizes as long as the block size does not exceed the cache-line size. STLB misses can be handled in software using whatever page-table structure chosen. Guarded page tables are ideal for this role.

7.1 Future Work

There are several unresolved issues that warrant further investigation. Guarded page tables stand to benefit from the use of multiple node sizes that are tuned to application behaviour in some manner. GPTs would benefit via a reduction in both memory consumption and tree depth. A reduction in tree depth would improve both mapping and TLB-refill performance.

GPTs could support operations on large, power-of-2 aligned region via manipulating roots of appropriate subtrees in the GPT. It is not clear how to take advantage of this potential benefit in the current microkernel as the kernel must also ensure that individual page-table entries do not have descendents in other address spaces.

Address space identifier (ASID) management is a significant issue in a microkernel that delivers interrupts via IPC to device drivers running in user level. In a shared page table that relies on the hardware ASIDs to distinguish between address spaces, the problem of how to deal with more address spaces that ASIDs arises. Revocation of an ASID from a page-table is expensive enough to cause unsatisfactory IPC latency. Emulating more ASIDs in software penalises TLB refill. A better solution would take advantage of hardware ASIDs, and at the same time ensure satisfactory IPC latencies.

Bibliography

[ABB+86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. MACH: A new kernel foundation for UNIX development. In *Proc. Summer USENIX*, July 1986.

[ABC+83] M. Atkinson, P. Bailey, K. Chisholm, P. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26, 1983.

[Abu] Al Aburto. Benchmark collection. Available: ftp://ftp.nosc.mil/pub/-aburto. January, 1998.

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proc. 4th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, April 1991.

[AP93] A. Agarwal and S. Pudar. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. In *Proc. 20th International Symposium on Computer Architecture*, 1993.

[Arn86] James Q. Arnold. Shared libraries on UNIX System V. In *Proc. Summer USENIX Conf.*, 1986.

[BCD69] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics virtual memory. In *Proc. 2nd Symp. on Operating Systems Principles*, 1969.

[BHK+91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *13th Symp. on Operating Systems Principles*, Pacific Grove, CA, October 1991. ACM.

[BKW94] Kavita Bala, M. Frans Kaashoek, and William E. Weihl. Software prefetching and caching for translation lookaside buffers. In *First Symposium on Operating Systems Design and Implementations*, pages 243–253. Usenix, 1994.

[Bro96] Mark R. Brown. FastCGI: A high-performance gateway interface. In *Fifth International World Wide Web Conference, 6 May 1996, Paris, France*. Paris, France, May 1996. Position Paper. Available: http://www.fastcgi.com/kit/doc/www5-api-workshop.html 17th April, 1998.

[CBJ92] J. Bradley Chen, Anita Borg, and Norman P. Jouppi. A simulation based study of TLB performance. In *19th International Symposium on Computer Architecture*, May 1992.

[CE85] Douglas Clark and Joel S. Emer. Performance of the VAX-11/780 translation buffer: Simulation and measurement. *ACM Transactions on Computer Systems*, 3(1):31–62, February 1985.

[CFL93] Jeff Chase, Mike Feeley, and Hank Levy. Some Issues for Single Address Space Systems. In *4th Int'l Workshop on Workstation Operating Systems*, Napa, California, October 1993. IEEE.

[CH81] Richard W. Carr and John L. Hennessy. WSCLOCK – a simple and effective algorithm for virtual memory management. In *Proc. 8th Symp. on Operating Systems Principles*, 1981.

[Cha95] Jeffrey S. Chase. *An Operating System Structure for Wide-Address Architectures*. PhD thesis, Dept. of Computer Science and Engineering, University of Washington, August 1995. Technical Report 95-08-06.

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

[CLK97] David Channon, Raymond Lai, and David Koch. Associativity revisited - a study of set, column and skewed associative TLBs using SPEC95. In Ronald Pose, editor, *2nd Australasian Computer Architecture Conference*. Springer, 1997.

[CM88] Albert Chang and Mark F. Mergen. 801 storage: Architecture and programming. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[Coc81] John Cocke. Virtual to real address translation using hashing. *IBM Technical Disclosure Bulletin*, 24(6), November 1981.

[CP78] Richard P. Case and Andris Padegs. Architecture of the IBM System/370. *Communications of the ACM*, 21(1), January 1978.

[CSD86] David Cheriton, Gert Slavenburg, and Patrick Doyle. Software-controlled caches in the VMP multiprocessor. In *Proc. 13th International Symposium on Computer Architecture*, 1986.

[DEC96] Digital Equipment Corporation, Maynard, Massachusetts. *Alpha 21164 Microprocessor Hardware Reference Manual*, July 1996. Order Number: EC-QAEQD-TE.

[Den70] Peter J. Denning. Virtual memory. *Computing Surveys*, 2(3), September 1970.

[DMM86] M. DeMoney, J. Moore, and J. Mashey. Operating system support on a RISC. *Proc. COMPCON Spring 1986*, 1986.

[EHL97] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. L4 reference manual – MIPS R4x00. Technical Report UNSW-CSE-TR-9707, School of Computer Science and Engineering, University of New South Wales, December 1997. Latest version available from http://www.cse.unsw.edu.au/~disy/.

[Elp93] Kevin Elphinstone. Address space management issues in the Mungi operating system. Technical Report 9312, School of Computer Science and Engineering, University of New South Wales, November 1993.

[Fot61] J. Fotheringham. Dynamic storage allocation in the Atlas computer, including an automatic use of backing store. *Communications of the ACM*, 4(10), October 1961.

[FR86] Robert Fitzgerald and Richard F. Rashid. The Integration of Virtual Memory Management and Interprocess Communication in Accent. *ACM Transactions on Computer Systems*, 4(2), May 1986.

[Gal96] Galileo Technology. *GT-64010A System controller with PCI interface for R4xxx and R5000 family CPUs*, December 1996. Available: http://www.galileot.com/library/dsheet/64010ads.pdf, February, 1999.

[GMS87] Robert A. Gingell, Joseph P. Moran, and William A. Shannon. Virtual memory architecture in SunOS. In *Proc. of Summer USENIX Conf.*, 1987.

[Han70] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4), April 1970.

[HCL95] W. Wilson Ho, Wei-Chau Chang, and Lilian H. Leung. Optimizing the performance of dynamically-linked programs. In *1995 USENIX Technical Conf.*, New Orleans, LA, January 1995.

[Hei93] Joe Heinrich. *MIPS R4000 Microprocessor User's Manual*. Prentice-Hall, 1993.

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

[HH93] Jerry Huck and Jim Hays. Architectural Support for Translation Table Management in Large Address Space Machines. In *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proc. 16th Symp. on Operating Systems Principles*. ACM, 1997.

[IBM78] IBM. *IBM System/38 technical developments*. Order no. G580-0237. IBM, Atlanta, Ga., 1978.

[JM97] Bruce Jacob and Trevor Mudge. Software-managed address translation. In *Proc. 3rd High-Performance Computer Architecture Conference*. IEEE, 1997.

[JM98a] Bruce L. Jacob and Trevor N. Mudge. A look at several memory management units, TLB-refill mechanisms, and page table organizations. In *Proc8th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, October 1998.

[JM98b] Bruce L. Jacob and Trevor N. Mudge. Virtual memory in contemporary microprocessors. *IEEE Micro*, 18(4), July/August 1998.

[KH91] Gerry Kane and Joe Heinrich. *MIPS Risc Architecture*. Prentice Hall, 1991.

[KH92] R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-index caches. *ACM Transactions on Computer Systems*, 10(4):338–359, November 1992.

[Kin90] Jeffrey H. Kingston. *Algorithms and Data Structures — Design, Correctness, Analysis*. Addison Wesley, 1990.

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

[Knu73] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 1973.

[KTNW93] Yousef A. Khalidi, Madhusudhan Talluri, Michael N. Nelson, and Dock Williams. Virtual Memory Support for Multiple Page Sizes. In *4th Int'l Workshop on Workstation Operating Systems*, Napa, California, October 1993. IEEE.

[LB89] T. Paul Lee and R. E. Barkley. A watermark-based lazy buddy system for kernel memory allocation. In *Proc. Summer USENIX Conf.*, 1989.

[LE95] Jochen Liedtke and Kevin Elphinstone. Guarded page tables on the MIPS R4600. Technical Report UNSW-CSE-TR-9503, School of Computer Science and Engineering, University of New South Wales, November 1995.

[Lee89] Ruby B. Lee. Precision architecture. *Computer*, January 1989.

[LES+97] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Hermann Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance. In *6th Workshop on Hot Topics in Operating Systems (HotOS)*, Chatham, Massachusetts, May 1997.

[Lie93] Jochen Liedtke. A persistent system in real use: Experiences of the first 13 years. In *1993Int'l Workshop on Object-Orientation in Operating Systems*, Asheville, North Carolina, December 1993.

[Lie94] Jochen Liedtke. A short note on implementing thread exclusiveness and address space locking. *Operating Systems Review*, 28(3):38–42, July 1994.

[Lie95a] Jochen Liedtke. Address space sparsity and fine granularity. *Operating Systems Review*, 29(1), January 1995. Also appeared in *6th SIGOPS European Workshop*, Schloß Dagstuhl, Germany, 1994.

[Lie95b] Jochen Liedtke. On micro-kernel construction. In *Proc. 15th Symp. on Operating Systems Principles*. ACM, December 1995.

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

[LL82] Henry M. Levy and Peter H. Lipman. Virtual memory management in the VAX/VMS operating system. *IEEE Computer*, 15(3), March 1982.

[Mil90] Milan Milenkovic. Microprocessor memory management units. *IEEE Micro*, 10(2), April 1990.

[MKL+94] M. K. McKusick, M. J. Karels, S. J. Leffler, W. N. Joy, and R. S. Fabry. *Berkeley Software Architecture Manual, 4.4 BSD Edition*, volume 4.4BSD Programmer's Supplementary Documents, pages 5:1–42. O'Reilly & Associates, 1994.

[Mog93] Jeffrey C. Mogul. Big Memories on the Desktop. In *4th Int'l Workshop on Workstation Operating Systems*, Napa, California, October 1993. IEEE.

[MWO+93] Kevin Murray, Tim Wilkinson, Peter Osmon, Ashly Saulsbury, Tom Stiemerling, and Paul Kelly. Design and implementation of an object-oriented 64-bit single address space microkernel. In *Proc. USENIX Microkernels and Other Kernel Architectures Symposium*. USENIX, September 1993.

[NUS+93] David Nagle, Richard Uhlig, Tim Stanely, Stuart Sechrest, Trevor Mudge, and Richard Brown. Design tradeoffs for software-managed TLBs. In *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993.

[OCH+85] John K. Ousterhout, Hervé De Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *10th Symp. on Operating Systems Principles*, 1985.

[Org72] E. Organick. *The Multics System: An Examination of its Structure*. MIT Press, Cambridge, MA, 1972.

[Pet] W. P. Peterson. Random number generator C source code. Available: http://www.netlib.org/random/zufall.shar. 30th March, 1998.

[PH90] David A. Patterson and John L. Hennessy. *Computer architecture: A quantitative approach*. Morgan Kaufmann, 1990.

[Pow97] IBM / Motorola. *PowerPC Microprocessor Family: The Programming Environments*, January 1997. Available: http://www.chips.ibm.com/products/ppc/documents/datasheets/products-.html. 13th February, 1998.

[PT77] J. L. Peterson and N. Theodore. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

[R4795] Integrated Device Technology. *IDT79R4600 and IDT79R4700 RISC Processor Hardware User's Manual*, April 1995.

[RA85] John Rosenberg and David Abramson. MONADS-PC - A capability-based workstation to support software engineering. In *Proc 18th Hawaii Int'l Conf. on System Sciences*, 1985.

[ROKB95] Theodore H. Romer, Wayne H. Ohlrich, Anna R. Karlin, and Brian N. Bershad. Reducing TLB and memory overhead using online superpage promotion. In *Proc. 22nd International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, 1995. ACM.

[RSD81] K. Ramamohanarao and R. Sacks-Davis. Hardware address translation for machines with a large virtual memory. *Information Processing Letters*, 13(1), October 1981.

[RTY+88] Richard Rashid, Avadis Tevavian, Jr., Michael Young, David Golub, Robert Baron, David Black, William j. Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, 37(8), August 1988.

[Sat81] M. Satyanarayanan. A study of file sizes and functional lifetimes. In *Proc. 8th Symp. on Operating Systems Principles*, 1981.

[Sez93] André Seznec. A case for two-way skewed-associative caches. In *Proc. 20th International Symposium on Computer Architecture*, San Diego, California, 1993. ACM.

[Smi78a] A. J. Smith. Bibliography on paging and related topics. *Operating Systems Review*, October 1978.

[Smi78b] A. J. Smith. A comparative study of of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Transactions on Software Engineering*, SE-4(2), March 1978.

[Smi86] Alan Jay Smith. Bibliography and readings on cache memories. *Computer Architecture News*, 11(1), January 1986.

[Smi91] Alan Jay Smith. Second bibliograpy on cache memories. *Computer Architecture News*, 19(4), June 1991.

[SPE95] Standard Performance Evaluation Corporation, Manassas, VA, USA. *SPECint95 Benchmark/SPECfp95 Benchmark*, August 1995.

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

[Tel90] Patricia J. Teller. Translation-lookaside buffer consistency. *IEEE Computer*, 23(6), June 1990.

[THK95] Madhusudhan Talluri, Mark D. Hill, and Yousef A. Khalidi. A new page table for 64-bit address spaces. In *15th Symp. on Operating Systems Principles*, Copper Mountain Resort, CO, 1995.

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

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

[VERH96] Jerry Vochteloo, Kevin Elphinstone, Stephen Russell, and Gernot Heiser. Protection domain extensions in Mungi. In *Proc. 5th Int'l Workshop on Object Orientation in Operating Systems*, Seattle, WA, USA, October 1996.

[Voc99] Jeroen Vochteloo. *The Design, Implementation and Performance of Protection in the Mungi Single Address Space Operating System*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, 1999.

[WCC+74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: the kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6), June 1974.

[WEG+86] David Wood, Susan Eggers, Garth Gibson, Mark Hill, Joan Pendleton, Scott Ritchie, George Taylor, Randy Katz, and David Patterson. An in-cache address translation mechanism. In *Proc. 13th International Symposium on Computer Architecture*, 1986.

[You94] Robert Young. *Spitfire Programmer's Reference*, February 1994. Unpublished, obtained from author.

[YR93] Hyuck Yoo and Tom Rogers. UNIX kernel support for OLTP performance. In *USENIX Winter Conference*, San Diego, January 1993.

Appendix A

Raw Results

A.1 GPT Evaluation Raw Results

This section contains the raws results for all the experiments carried out in the GPT evaluation (Chapter 4). It is provided for the reader to confirm the results presented, or to perform one's own analysis. The following definitions are required to interpret the results.

BENCH The name of the benchmark run.

PAGE The number of mapped 4K pages needed by the benchmark to run.

SIZE The size (in bytes) of the GPT (including the extra PTE pair nodes) required to map the needed pages.

NULL The number of null guards in the GPT.

LF The number of leaf gpt entries, i.e. gpt entries that point to a PTE pair node.

INT The number of internal entries in the GPT, ie entries which point to another gpt node.

AVD The average depth of leaf entries in the gpt tree. This does not include the extra load required to access the leaf PTE pair.

NSTD Is the normalised standard deviation relative to the quantity associated with it.

BENCH	G2	G4	G8	G16
wave5	178.8	111.7	92.4	82.4
apsi	144.4	108.6	87.0	83.1
swim	378.5	233.5	183.5	164.1
tfftdp	172.5	107.1	93.4	82.1
mm	229.6	121.8	103.2	94.4
heapsort	267.6	145.3	114.9	114.5
nsieve	206.8	117.6	100.9	93.8
c4	308.3	140.6	113.4	99.7
gcc	233.8	117.0	101.0	93.2
compress	155.7	109.4	94.5	89.0
go	191.9	141.4	123.9	111.4

BENCH	G32	G64	G128	G256
wave5	72.6	74.5	65.8	65.8
apsi	73.3	82.3	79.2	64.7
swim	141.8	143.6	128.0	132.9
tfftdp	76.8	73.6	68.4	64.4
mm	86.3	86.3	78.2	77.9
heapsort	106.1	105.5	84.8	86.3
nsieve	86.1	85.6	77.4	80.1
c4	92.2	91.4	83.1	85.2
gcc	82.3	84.0	74.5	75.8
compress	77.6	78.1	69.7	69.4
go	113.0	156.3	100.0	103.2

Table A.1: Average TLB refill time for conventional application benchmarks (cycles) for each GPT implementation.

BENCH	G2	G4	G8	G16
wave5	3691.0	3437.0	3377.1	3331.3
apsi	1902.9	1872.2	1836.2	1856.5
swim	2142.5	2135.1	2133.0	2131.3
tfftdp	16.4	14.4	13.9	13.6
mm	207.1	127.7	110.8	110.5
heapsort	30.2	29.6	29.7	29.5
nsieve	174.6	163.8	162.1	161.4
c4	88.6	59.4	51.8	51.4
gcc	67.0	55.2	53.5	52.7
compress	1117.1	1046.7	1026.0	1007.8
go	969.1	963.9	954.2	960.7

BENCH	G32	G64	G128	G256
wave5	3289.9	3294.7	3275.5	3275.2
apsi	1850.7	1850.1	1859.6	1836.6
swim	2129.6	2129.6	2129.2	2129.3
tfftdp	13.5	13.4	13.3	13.1
mm	105.2	105.7	97.1	94.7
heapsort	29.5	29.5	29.5	29.5
nsieve	160.5	160.5	158.9	159.3
c4	49.9	50.0	48.3	48.4
gcc	51.4	51.6	50.9	50.7
compress	995.1	998.6	983.9	983.6
go	960.8	964.7	959.2	959.5

Table A.2: Conventional application elapsed run time (seconds) for each GPT implementation.

BENCH	G2	G4	G8	G16
wave5	663.3	414.1	354.3	309.7
apsi	112.5	76.05	58.26	52.14
swim	22.4	13.92	11.45	9.986
tfftdp	5.272	3.288	2.596	2.26
mm	156.4	71.97	56.1	55.3
heapsort	1.045	0.4858	0.4932	0.4117
nsieve	25.16	13.96	11.92	11.44
c4	53.01	24.19	18.92	16.67
gcc	23.6	11.78	10.26	9.452
compress	239.6	168.6	150.1	141.2
go	17.93	13.23	21.07	18.95

BENCH	G32	G64	G128	G256
wave5	267.6	274.7	245.9	245.8
apsi	50.41	56.61	56.14	45.84
swim	8.569	8.678	7.851	8.166
tfftdp	2.354	2.253	2.09	1.971
mm	50.45	50.5	42.45	42.25
heapsort	0.3533	0.3519	0.3494	0.3569
nsieve	10.51	10.46	8.542	8.84
c4	15.78	15.64	14.21	14.59
gcc	8.226	8.394	7.564	7.693
compress	119.4	120.2	107.1	106.7
go	10.55	14.6	9.354	9.645

Table A.3: Time spent in TLB miss handling for conventional applications (seconds) for each GPT implementation.

BENCH	G2	G4	G8	G16
wave5	3.709e+008	3.707e+008	3.834e+008	3.76e+008
apsi	7.788e+007	7e+007	6.694e+007	6.273e+007
swim	5.919e+006	5.961e+006	6.24e+006	6.086e+006
tfftdp	3.056e+006	3.069e+006	2.781e+006	2.753e+006
mm	6.81e+007	5.911e+007	5.433e+007	5.857e+007
heapsort	3.905e+005	3.343e+005	4.292e+005	3.597e+005
nsieve	1.217e+007	1.186e+007	1.182e+007	1.219e+007
c4	1.719e+007	1.721e+007	1.668e+007	1.673e+007
gcc	1.01e+007	1.007e+007	1.016e+007	1.015e+007
compress	1.539e+008	1.541e+008	1.588e+008	1.586e+008
go	9.344e+006	9.362e+006	1.701e+007	1.7e+007

BENCH	G32	G64	G128	G256
wave5	3.687e+008	3.686e+008	3.735e+008	3.735e+008
apsi	6.878e+007	6.878e+007	7.092e+007	7.087e+007
swim	6.043e+006	6.044e+006	6.134e+006	6.145e+006
tfftdp	3.063e+006	3.062e+006	3.057e+006	3.058e+006
mm	5.848e+007	5.849e+007	5.428e+007	5.425e+007
heapsort	3.33e+005	3.335e+005	4.122e+005	4.135e+005
nsieve	1.221e+007	1.221e+007	1.103e+007	1.103e+007
c4	1.712e+007	1.711e+007	1.711e+007	1.711e+007
gcc	9.993e+006	9.991e+006	1.015e+007	1.015e+007
compress	1.539e+008	1.539e+008	1.538e+008	1.538e+008
go	9.336e+006	9.342e+006	9.353e+006	9.346e+006

Table A.4: Number of TLB misses for conventional applications for each GPT implementation.

BENCH	G16 Pair	G16 Single
wave5	3331.3	3384.2
apsi	1856.5	1862.9
swim	2131.3	2131.3
tfftdp	13.6	14.2
mm	110.5	111.2
heapsort	29.5	29.5
nsieve	161.4	163.2
c4	51.4	53.1
gcc	52.7	54.0
compress	1007.8	1029.2
go	960.6	962.1

Table A.5: Elapsed time (seconds) comparison between a G16 with specialised pair leaf nodes, and a G16 that loads from leaves with single PTEs.

BENCH	PAGE	SIZE	NULL	LF	INT	AVD
wave5	10330	248592	0	5184	5182	13.9
apsi	557	13632	0	289	287	11.3
swim	3626	87120	0	1820	1818	12.9
tfftdp	1035	24960	0	525	523	11.5
mm	1967	47376	0	992	990	12.3
heapsort	1010	24384	0	513	511	11.6
nsieve	1243	29952	0	629	627	11.5
c4	1299	31344	0	658	656	11.5
gcc	2391	57648	0	1206	1204	12.5
compress	8931	214560	0	4475	4473	13.8
go	202	5088	0	111	109	8.8

Table A.6: Conventional application GPT size statistics for G2.

BENCH	PAGE	SIZE	NULL	LF	INT	AVD
wave5	10330	194064	38	5184	1740	8.0
apsi	557	11168	31	289	106	7.8
swim	3626	68048	18	1820	612	8.0
tfftdp	1035	19616	14	525	179	6.8
mm	1967	37136	18	992	336	7.7
heapsort	1010	19168	14	513	175	6.9
nsieve	1243	23520	15	629	214	7.0
c4	1299	24624	16	658	224	7.0
gcc	2391	45168	20	1206	408	7.8
compress	8931	167488	33	4475	1502	8.0
go	202	4224	17	111	42	5.8

Table A.7: Conventional application GPT size statistics for G4.

BENCH	PAGE	SIZE	NULL	LF	INT	AVD
wave5	10330	179120	87	5192	753	6.0
apsi	557	10672	70	288	50	5.9
swim	3626	62704	43	1820	265	6.0
tfftdp	1035	18304	43	525	80	5.9
mm	1967	34592	60	991	149	6.0
heapsort	1010	17968	49	512	79	5.9
nsieve	1243	22000	52	628	96	6.0
c4	1299	23104	58	657	101	6.0
gcc	2391	42016	61	1207	180	6.0
compress	8931	154448	84	4474	650	6.0
go	202	4400	57	112	23	4.9

Table A.8: Conventional application GPT size statistics for G8.

BENCH	PAGE	SIZE	NULL	LF	INT	AVD
wave5	10330	172976	111	5192	353	5.0
apsi	557	11312	140	288	28	4.9
swim	3626	60912	78	1820	126	5.0
tfftdp	1035	18176	83	525	40	5.0
mm	1967	33824	97	991	72	5.0
heapsort	1010	17712	81	512	39	5.0
nsieve	1243	21360	70	628	46	5.0
c4	1299	22592	86	657	49	5.0
gcc	2391	40864	91	1207	86	5.0
compress	8931	149712	139	4474	307	5.0
go	202	4400	76	112	12	3.9

Table A.9: Conventional application GPT size statistics for G16.

BENCH	PAGE	SIZE	NULL	LF	INT	AVD
wave5	10330	170960	212	5184	174	4.0
apsi	557	12256	240	289	17	4.0
swim	3626	60304	135	1820	63	4.0
tfftdp	1035	18080	128	525	21	4.0
mm	1967	33744	157	992	37	4.0
heapsort	1010	17888	140	513	21	4.0
nsieve	1243	21280	117	629	24	4.0
c4	1299	22768	150	658	26	4.0
gcc	2391	40752	160	1206	44	4.0
compress	8931	148864	270	4475	153	4.0
go	202	5312	170	111	9	3.9

Table A.10: Conventional application GPT size statistics for G32.

BENCH	PAGE	SIZE	NULL	LF	INT	AVD
wave5	10330	171056	431	5184	89	4.0
apsi	557	14912	538	289	13	4.0
swim	3626	61936	393	1820	35	4.0
tfftdp	1035	19712	365	525	14	4.0
mm	1967	35376	402	992	22	4.0
heapsort	1010	19520	377	513	14	4.0
nsieve	1243	23424	387	629	16	4.0
c4	1299	24912	421	658	17	4.0
gcc	2391	42896	440	1206	26	4.0
compress	8931	150496	573	4475	80	4.0
go	202	7968	464	111	9	3.9

Table A.11: Conventional application GPT size statistics for G64.

BENCH	PAGE	SIZE	NULL	LF	INT	AVD
wave5	10330	170992	662	5184	46	3.0
apsi	557	16896	858	289	9	3.0
swim	3626	61872	597	1820	19	3.0
tfftdp	1035	20672	622	525	9	3.0
mm	1967	36336	663	992	13	3.0
heapsort	1010	20480	634	513	9	3.0
nsieve	1243	24384	645	629	10	3.0
c4	1299	26896	743	658	11	3.0
gcc	2391	43856	703	1206	15	3.0
compress	8931	149408	736	4475	41	3.0
go	202	9952	782	111	7	3.0

Table A.12: Conventional application GPT size statistics for G128.

BENCH	PAGE	SIZE	NULL	LF	INT	AVD
wave5	10330	173104	944	5184	24	3.0
apsi	557	25152	1504	289	7	3.0
swim	3626	66032	993	1820	11	3.0
tfftdp	1035	24832	1013	525	6	3.0
mm	1967	40496	1056	992	8	3.0
heapsort	1010	24640	1025	513	6	3.0
nsieve	1243	26496	909	629	6	3.0
c4	1299	31056	1135	658	7	3.0
gcc	2391	48016	1097	1206	9	3.0
compress	8931	153568	1143	4475	22	3.0
go	202	14112	1172	111	5	3.0

Table A.13: Conventional application GPT size statistics for G256.

BENCH	PAGE	SIZE	NSTD	NULL	NSTD	LF	INT	NSTD	AVD
64	75	3408	0	0	0	77	75	0	8.4
128	139	6480	0	0	0	141	139	0	9.3
256	267	12624	0	0	0	269	267	0	10.3
512	523	24912	0	0	0	525	523	0	11.3
1024	1035	49488	0	0	0	1037	1035	0	12.3
2048	2059	98640	0	0	0	2061	2059	0	13.3
4096	4107	196944	0	0	0	4109	4107	0	14.3
8192	8202	393537	$6e-005$	0	0	8204	8202	$6e-005$	15.3

Table A.14: SPARSE-PAGE benchmark GPT size statistics for G2.

BENCH	PAGE	SIZE	NSTD	NULL	NSTD	LF	INT	NSTD	AVD
64	75	3990	0.04	69	0.1	77	48	0.06	6.3
128	139	7632	0.03	128	0.07	141	89	0.04	6.8
256	267	14748	0.03	237	0.08	269	168	0.04	7.2
512	523	29270	0.01	470	0.04	525	331	0.02	7.7
1024	1035	57603	0.007	902	0.02	1037	645	0.01	8.2
2048	2059	115113	0.006	1806	0.02	2061	1288	0.009	8.7
4096	4106	229089	0.005	3565	0.02	4108	2557	0.007	9.2
8192	8202	458265	0.002	7139	0.007	8204	5114	0.003	9.7

Table A.15: SPARSE-PAGE benchmark GPT size statistics for G4.

BENCH	PAGE	SIZE	NSTD	NULL	NSTD	LF	INT	NSTD	AVD
64	75	5260	0.06	175	0.09	78	35	0.07	4.9
128	139	10137	0.04	322	0.07	142	65	0.05	5.1
256	267	19404	0.04	589	0.07	270	121	0.05	5.4
512	523	37734	0.02	1111	0.03	526	232	0.02	5.7
1024	1035	77414	0.01	2321	0.02	1038	478	0.02	6.1
2048	2059	151153	0.01	4434	0.02	2061	926	0.01	6.4
4096	4107	297740	0.006	8610	0.01	4110	1816	0.008	6.7
8192	8202	610505	0.007	18035	0.01	8205	3747	0.009	7.0

Table A.16: SPARSE-PAGE benchmark GPT size statistics for G8.

BENCH	PAGE	SIZE	NSTD	NULL	NSTD	LF	INT	NSTD	AVD
64	75	7142	0.08	303	0.1	78	24	0.09	4.6
128	139	13926	0.06	577	0.08	142	47	0.07	4.7
256	267	29568	0.04	1245	0.05	270	100	0.04	5.0
512	523	54451	0.03	2207	0.04	526	181	0.03	5.2
1024	1035	101529	0.02	3974	0.03	1038	333	0.02	5.4
2048	2059	217420	0.02	8780	0.02	2062	722	0.02	5.7
4096	4106	463918	0.005	19256	0.007	4109	1557	0.006	6.0
8192	8202	865404	0.007	34844	0.01	8205	2869	0.008	6.2

Table A.17: SPARSE-PAGE benchmark GPT size statistics for G16.

BENCH	PAGE	SIZE	NSTD	NULL	NSTD	LF	INT	NSTD	AVD
64	75	10640	0.06	560	0.07	77	20	0.06	4.5
128	139	23952	0.07	1240	0.08	141	44	0.07	4.7
256	267	50422	0.03	2591	0.03	269	92	0.03	4.9
512	523	87491	0.05	4331	0.06	525	156	0.05	5.1
1024	1035	142121	0.02	6631	0.03	1037	247	0.02	5.2
2048	2059	296028	0.02	13933	0.02	2061	515	0.02	5.4
4096	4107	712438	0.01	35114	0.01	4109	1265	0.01	5.7
8192	8202	1555596	0.008	78101	0.01	8204	2784	0.009	5.9

Table A.18: SPARSE-PAGE benchmark GPT size statistics for G32.

BENCH	PAGE	SIZE	NSTD	NULL	NSTD	LF	INT	NSTD	AVD
64	75	15600	0.1	1065	0.09	77	18	0.08	4.3
128	139	26044	0.08	1580	0.08	141	27	0.08	4.3
256	267	65264	0.07	3739	0.07	269	63	0.07	4.5
512	523	158448	0.03	8964	0.03	525	150	0.03	4.7
1024	1035	354134	0.02	19987	0.02	1037	333	0.02	4.9
2048	2059	582076	0.02	31979	0.02	2061	540	0.02	5.0
4096	4106	839713	0.02	43766	0.02	4108	759	0.02	5.1
8192	8202	1547088	0.02	79159	0.02	8204	1386	0.02	5.2

Table A.19: SPARSE-PAGE benchmark GPT size statistics for G64.

BENCH	PAGE	SIZE	NSTD	NULL	NSTD	LF	INT	NSTD	AVD
64	75	41340	0.1	2924	0.08	77	23	0.08	3.9
128	139	88035	0.05	5692	0.05	141	45	0.05	4.0
256	267	136982	0.03	8472	0.03	269	68	0.03	4.1
512	523	179990	0.07	10629	0.07	525	87	0.07	4.1
1024	1035	270716	0.07	15235	0.08	1037	128	0.07	4.1
2048	2059	598806	0.06	33541	0.06	2061	280	0.06	4.2
4096	4106	1718036	0.03	98866	0.03	4108	810	0.03	4.4
8192	8203	4738016	0.01	277981	0.02	8204	2253	0.02	4.6

Table A.20: SPARSE-PAGE benchmark GPT size statistics for G128.

BENCH	PAGE	SIZE	NSTD	NULL	NSTD	LF	INT	NSTD	AVD
64	75	50416	0.09	3501	0.08	77	14	0.08	4.2
128	139	63318	0.1	4176	0.1	141	16	0.1	4.1
256	267	107964	0.1	6700	0.1	269	27	0.1	4.2
512	523	280816	0.08	16950	0.09	525	68	0.08	4.2
1024	1035	839920	0.05	50736	0.05	1037	203	0.05	4.4
2048	2059	2313659	0.02	140441	0.02	2060	558	0.02	4.6
4096	4107	5109590	0.008	310416	0.008	4109	1233	0.008	4.9
8192	8202	8028412	0.007	483952	0.008	8205	1930	0.008	5.0

Table A.21: SPARSE-PAGE benchmark GPT size statistics for G256.

BENCH	PAGE	NSTD	SIZE	NSTD	NULL	NSTD	LF	NSTD	INT	NSTD	AVD
64	382	1	10527	0.9	0	0	225	0.9	223	0.9	10.4
128	1327	1	34417	1	0	0	723	1	721	1	12.5
256	1886	0.8	50226	0.7	0	0	1052	0.7	1050	0.7	13.6
512	4051	0.6	107040	0.5	0	0	2236	0.5	2234	0.5	15.3
1024	8438	0.4	221959	0.4	0	0	4630	0.4	4628	0.4	16.9
2048	16122	0.3	425836	0.3	0	0	8877	0.3	8875	0.3	18.0

Table A.22: SPARSE-FILE benchmark GPT size statistics for G2.

BENCH	PAGE	NSTD	SIZE	NSTD	NULL	NSTD	LF	NSTD	INT	NSTD	AVD
64	505	2	12768	2	114	0.1	286	2	132	1	7.1
128	1014	1	25384	1	215	0.09	566	1	260	0.9	8.0
256	1893	1	47876	0.7	412	0.06	1056	0.9	488	0.6	8.8
512	3854	0.5	97006	0.4	824	0.05	2137	0.5	986	0.3	9.6
1024	8050	0.5	200017	0.4	1626	0.03	4437	0.4	2020	0.3	10.3
2048	16093	0.3	399852	0.2	3254	0.03	8860	0.3	4037	0.2	11.0

Table A.23: SPARSE-FILE benchmark GPT size statistics for G4.

BENCH	PAGE	NSTD	SIZE	NSTD	NULL	NSTD	LF	NSTD	INT	NSTD	AVD
64	580	2	15919	1	297	0.1	323	2	87	0.8	5.3
128	906	1	27187	0.7	559	0.09	512	1	152	0.5	5.8
256	1816	0.9	53691	0.5	1060	0.06	1018	0.8	295	0.4	6.3
512	3913	0.6	111321	0.3	2058	0.05	2167	0.5	602	0.3	6.8
1024	8396	0.5	234838	0.3	4235	0.03	4608	0.4	1262	0.2	7.4
2048	16455	0.3	461307	0.2	8307	0.02	9042	0.3	2477	0.2	7.8

Table A.24: SPARSE-FILE benchmark GPT size statistics for G8.

BENCH	PAGE	NSTD	SIZE	NSTD	NULL	NSTD	LF	NSTD	INT	NSTD	AVD
64	597	2	19985	1	562	0.1	332	2	59	0.7	4.7
128	974	1	36721	0.5	1129	0.08	546	1	111	0.3	5.1
256	1772	0.7	71224	0.3	2279	0.06	996	0.6	217	0.2	5.5
512	3973	0.6	144904	0.3	4270	0.05	2196	0.6	430	0.2	5.9
1024	7415	0.4	272984	0.2	8050	0.04	4118	0.4	810	0.1	6.2
2048	14689	0.3	558725	0.1	16965	0.02	8159	0.3	1674	0.09	6.4

Table A.25: SPARSE-FILE benchmark GPT size statistics for G16.

BENCH	PAGE	NSTD	SIZE	NSTD	NULL	NSTD	LF	NSTD	INT	NSTD	AVD
64	488	2	25830	0.5	1083	0.1	278	1	43	0.3	4.6
128	1003	1	54321	0.4	2253	0.08	560	1	90	0.2	4.9
256	2039	0.9	110842	0.3	4555	0.07	1129	0.8	183	0.2	5.3
512	4011	0.6	208172	0.2	8309	0.05	2216	0.5	339	0.1	5.6
1024	7888	0.5	383541	0.2	14716	0.03	4354	0.4	615	0.1	5.8
2048	15223	0.3	766765	0.1	29904	0.02	8426	0.3	1236	0.07	5.9

Table A.26: SPARSE-FILE benchmark GPT size statistics for G32.

BENCH	PAGE	NSTD	SIZE	NSTD	NULL	NSTD	LF	NSTD	INT	NSTD	AVD
64	474	2	38430	0.4	2084	0.1	271	2	37	0.2	4.1
128	752	1	68428	0.2	3605	0.1	434	1	64	0.2	4.3
256	1755	0.9	152335	0.2	7670	0.08	987	0.8	137	0.1	4.6
512	3421	0.5	333858	0.1	16985	0.04	1921	0.5	299	0.06	4.9
1024	8138	0.4	722039	0.08	35785	0.03	4482	0.4	639	0.05	5.2
2048	16157	0.3	1315197	0.07	63527	0.02	8892	0.3	1149	0.04	5.4

Table A.27: SPARSE-FILE benchmark GPT size statistics for G64.

BENCH	PAGE	NSTD	SIZE	NSTD	NULL	NSTD	LF	NSTD	INT	NSTD	AVD
64	461	2	81288	0.2	5027	0.1	264	2	41	0.1	3.6
128	969	1	162751	0.2	9521	0.08	544	1	79	0.1	3.9
256	1901	0.8	289762	0.1	16370	0.07	1060	0.8	137	0.08	4.1
512	3777	0.5	474202	0.09	25738	0.06	2099	0.5	219	0.07	4.2
1024	7866	0.5	876683	0.08	46222	0.04	4344	0.4	398	0.06	4.3
2048	15623	0.3	1812764	0.06	95743	0.03	8625	0.3	821	0.04	4.4

Table A.28: SPARSE-FILE benchmark GPT size statistics for G128.

BENCH	PAGE	NSTD	SIZE	NSTD	NULL	NSTD	LF	NSTD	INT	NSTD	AVD
64	555	2	123801	0.2	7601	0.1	311	2	31	0.1	3.9
128	967	1	203877	0.1	12126	0.1	541	0.9	49	0.1	4.0
256	1875	0.9	374989	0.1	21770	0.08	1047	0.8	89	0.09	4.1
512	3687	0.5	821661	0.07	47569	0.07	2053	0.5	194	0.07	4.2
1024	7563	0.5	1879284	0.05	109141	0.04	4193	0.4	444	0.04	4.4
2048	16246	0.3	4446624	0.03	259503	0.02	8937	0.3	1052	0.03	4.6

Table A.29: SPARSE-FILE benchmark GPT size statistics for G256.

Page Table	MAP1	NSTD	MAPN	NSTD	MAP1 NC	NSTD	MAPN NC	NSTD
G1	118.1	0.006	90.3	0.002	1811.5	0.007	1534.7	0.011
G2	97.7	0.001	67.3	0.001	1697.2	0.005	1414.8	0.011
G3	103.8	0.007	72.6	0.010	1664.4	0.010	1377.2	0.005
G4	104.0	0.000	77.0	0.001	1609.6	0.006	1351.0	0.010
G5	73.8	0.002	56.0	0.004	1621.6	0.009	1356.0	0.013
G6	81.0	0.006	56.1	0.009	1562.8	0.003	1320.6	0.008
G7	66.4	0.001	52.5	0.003	1562.8	0.005	1321.5	0.010
G8	75.2	0.001	53.5	0.001	1524.2	0.011	1293.3	0.009

Table A.30: Elapsed time results (in milliseconds) for MAP1 and MAPN benchmark with and without (NC) caching, for G2 – G256.

BENCH	G2	NSTD	G4	NSTD	G8	NSTD	G16	NSTD
64K	63.8	0.008	57.9	0.009	56.7	0.01	53.0	0.02
128K	73.4	0.04	67.3	0.03	66.5	0.02	59.8	0.03
256K	79.6	0.03	74.5	0.05	75.1	0.04	70.2	0.03
512K	82.0	0.04	78.0	0.03	82.6	0.05	83.0	0.04
1M	83.4	0.03	80.6	0.04	83.4	0.04	88.1	0.07
2M	84.7	0.02	80.4	0.02	87.0	0.05	87.7	0.06
4M	85.0	0.02	81.5	0.03	86.6	0.05	89.5	0.06
8M	85.5	0.01	82.9	0.03	86.5	0.05	94.3	0.08
16M	85.6	0.01	82.9	0.02	87.6	0.04	95.3	0.08
32M	86.1	0.01	82.8	0.03	87.7	0.04	91.8	0.07
64M	86.8	0.01	83.2	0.02	86.2	0.04	93.8	0.08
128M	86.1	0.01	82.2	0.03	88.7	0.03	99.6	0.06
256M	86.1	0.01	81.4	0.02	87.3	0.05	97.4	0.07
512M	87.9	0.01	84.6	0.03	88.9	0.04	93.5	0.08

BENCH	G32	NSTD	G64	NSTD	G128	NSTD	G256	NSTD
64K	54.1	0.01	54.2	0.01	54.1	0.01	56.1	0.01
128K	60.5	0.02	60.3	0.02	61.3	0.02	62.9	0.02
256K	67.4	0.03	67.4	0.03	68.4	0.02	69.8	0.03
512K	81.1	0.02	78.1	0.02	78.8	0.02	80.7	0.03
1M	100.9	0.04	101.6	0.01	98.6	0.02	99.5	0.01
2M	112.7	0.08	134.2	0.07	138.7	0.01	136.1	0.009
4M	111.2	0.1	160.5	0.1	212.4	0.07	224.3	0.03
8M	97.8	0.1	149.1	0.1	252.3	0.1	378.6	0.1
16M	97.3	0.09	125.5	0.2	221.9	0.2	479.9	0.2
32M	113.4	0.1	107.5	0.2	182.1	0.2	449.3	0.2
64M	117.5	0.07	118.9	0.2	154.6	0.3	287.6	0.4
128M	115.0	0.1	150.0	0.1	136.1	0.2	218.3	0.3
256M	101.1	0.1	165.0	0.09	154.8	0.2	198.4	0.4
512M	102.2	0.09	153.6	0.1	232.8	0.1	188.3	0.3

Table A.31: Elapsed time results (in microseconds) for MAPS benchmark for various region sizes, for G2 – G256.

Page Table	Time	NSTD
G1	15.0	0.0007
G2	15.6	0.001
G3	15.2	0.0003
G4	15.2	0.002
G5	16.6	0.001
G6	19.7	0.001
G7	24.6	0.0006
G8	32.0	0.0005

Table A.32: Elapsed time results (in milliseconds) for task benchmark for G2 – G256.

A.2 STLB Raw Results

BENCH	1×8K	1×128K	2×4K	2×64K
wave5	26.8	24.8	33.4	33.3
apsi	26.0	25.3	31.5	31.6
swim	144.1	47.6	60.4	60.3
tfftdp	23.0	22.6	29.9	30.0
mm	31.6	31.5	46.3	46.3
heapsort	32.1	32.1	49.1	49.2
nsieve	33.8	33.2	43.9	43.9
c4	52.6	32.3	43.4	43.4
gcc	60.7	26.3	41.7	41.9
compress	29.5	28.7	38.0	38.1
go	42.2	42.2	55.4	55.3

Table A.33: Average TLB refill time (in cycles) for conventional application benchmarks, for each STLB configuration.

BENCH	1×8K	1×128K	2×4K	2×64K
wave5	3115.2	3108.6	3135.1	3140.1
apsi	1795.3	1793.9	1795.6	1796.2
swim	2130.8	2124.2	2131.6	2125.0
tfftdp	12.0	12.0	12.1	12.2
mm	74.1	73.0	82.8	82.8
heapsort	29.4	29.4	29.4	29.4
nsieve	155.1	155.1	155.8	156.0
c4	41.2	37.4	41.9	38.9
gcc	49.3	45.4	49.9	47.0
compress	921.8	920.5	932.1	932.5
go	938.2	938.2	941.0	941.0

Table A.34: Conventional application elapsed run time (in seconds) for each STLB configuration.

BENCH	1×8K	1×128K	2×4K	2×64K
wave5	102.9	95.2	126.6	126.3
apsi	17.12	16.67	21.42	21.4
swim	8.834	2.967	3.648	3.645
tfftdp	0.6346	0.6262	0.8377	0.8382
mm	17.09	17.02	30.77	30.72
heapsort	0.1482	0.1476	0.1746	0.1752
nsieve	3.725	3.662	5.761	5.765
c4	8.761	5.381	7.228	7.232
gcc	6.108	2.672	4.226	4.24
compress	46.75	45.47	60.17	60.36
go	7.173	7.175	9.432	9.42

Table A.35: The time spent in TLB miss handling for conventional applications (in seconds), for each STLB configuration.

BENCH	1×8K	1×128K	2×4K	2×64K
wave5	3.832e+008	3.831e+008	3.789e+008	3.79e+008
apsi	6.589e+007	6.593e+007	6.795e+007	6.783e+007
swim	6.132e+006	6.23e+006	6.043e+006	6.048e+006
tfftdp	2.763e+006	2.769e+006	2.797e+006	2.795e+006
mm	5.405e+007	5.397e+007	6.645e+007	6.64e+007
heapsort	4.611e+005	4.594e+005	3.56e+005	3.559e+005
nsieve	1.104e+007	1.102e+007	1.313e+007	1.312e+007
c4	1.667e+007	1.667e+007	1.666e+007	1.665e+007
gcc	1.007e+007	1.015e+007	1.013e+007	1.013e+007
compress	1.582e+008	1.582e+008	1.585e+008	1.585e+008
go	1.699e+007	1.7e+007	1.702e+007	1.704e+007

Table A.36: Number of TLB misses for conventional applications for each of the STLB configurations.

BENCH	1×8K	2×4K	1×128K	2×64K
wave5	6.51e+006	1.43e+006	1.09e+004	1.17e+004
apsi	7.33e+005	614	4.44e+003	587
swim	3.34e+006	3.55e+006	9.25e+003	3.66e+003
tfftdp	1.87e+004	2.15e+004	5.07e+003	1.08e+003
mm	3.29e+004	5.02e+003	1.56e+004	2.02e+003
heapsort	1.53e+003	1.05e+003	1.47e+003	1.04e+003
nsieve	3.49e+004	1.72e+004	1.23e+004	1.27e+003
c4	3.06e+006	3.28e+006	3.39e+004	1.33e+003
gcc	3.23e+006	2.87e+006	5.69e+003	2.43e+003
compress	8.45e+005	2.43e+005	9.97e+003	9.9e+003
go	3.94e+003	2.87e+003	3.03e+003	2.15e+003

Table A.37: Number of STLB misses for conventional applications for each STLB configuration.

BENCH	No STLB	1×8K	1×128K	2×4K	2×64K
4K	5.5	4.9	5.0	4.6	5.0
8K	10.4	9.7	9.8	9.4	10.0
16K	16.5	16.0	16.3	16.2	16.6
32K	22.2	22.4	22.8	22.7	22.7
64K	35.2	37.0	36.9	38.1	37.5
128K	59.5	61.9	62.1	62.5	63.1
256K	105.7	115.3	116.5	120.0	119.8
512K	202.6	215.2	217.0	229.7	232.0
1M	492.1	539.1	538.3	574.6	575.1
2M	1017.0	1111.9	1111.2	1170.3	1172.0
4M	2030.5	2222.4	2224.4	2330.7	2334.9
8M	4045.0	4436.9	4440.7	4651.2	4669.8
16M	8099.7	8828.0	8825.0	9256.5	9310.0

Table A.38: Elapsed time for unmap benchmark (in microseconds), for each STLB configuration.

BENCH	No STLB	1×8K	1×128K	2×4K	2×64K
4K	49.0	50.3	50.3	51.0	51.0
8K	147.2	149.8	149.8	151.2	151.2
16K	232.0	237.2	237.2	240.0	240.1
32K	355.1	365.4	365.3	371.2	371.2
64K	601.3	621.8	621.8	633.5	633.5
128K	1102.5	1143.6	1143.5	1166.8	1166.8
256K	2134.7	2216.5	2216.6	2262.8	2263.1
512K	4157.7	4321.1	4321.2	4414.3	4414.1
1M	8203.6	8530.4	8530.3	8716.4	8716.5
2M	16304.0	16957.7	16957.5	17330.0	17330.2
4M	32514.1	33822.8	33822.7	34567.0	34566.9
8M	64934.1	67551.7	67551.5	69040.0	69040.1
16M	129786.0	135020.3	135020.2	137998.0	137997.3

Table A.39: Elapsed time for unmap benchmark (in microseconds) without caching, for each STLB configuration.

Page Table	Time	NSTD
No STLB	15.20	0.002
1×8K	16.02	0.0004
1×128K	16.08	0.0003
2×4K	15.77	0.001
2×64K	15.96	0.003

Table A.40: Elapsed time for task benchmark (in milliseconds) for each STLB configuration.

A.3 Raw results for other page tables

BENCH	MPT	H512	H8192	C512	C8192
wave5	108.0	77.6	28.3	75.7	36.7
apsi	115.7	28.3	27.6	40.6	37.4
swim	192.7	174.7	54.8	171.9	65.9
tfftdp	117.0	49.8	25.4	77.5	33.7
mm	116.0	41.0	39.8	44.5	43.0
heapsort	128.3	43.9	43.4	52.5	52.0
nsieve	127.0	48.7	38.2	55.4	43.1
c4	119.5	67.6	36.5	98.1	46.5
gcc	111.6	89.9	35.4	84.6	39.6
compress	80.6	53.0	32.4	62.1	40.8
go	148.4	48.8	48.8	64.1	63.8

Table A.41: Average TLB refill time for conventional application benchmarks (cycles) for various page-table implementations.

BENCH	MPT	H512	H8192	C512	C8192
wave5	3424.7	3317.7	3127.3	3300.0	3152.7
apsi	1867.2	1794.4	1793.2	1815.2	1809.0
swim	2133.4	2132.1	2124.0	2132.5	2124.9
tfftdp	15.1	12.7	12.1	13.7	12.1
mm	115.4	77.1	77.0	75.1	75.0
heapsort	29.6	29.4	29.4	29.3	29.3
nsieve	166.7	156.4	155.5	157.1	155.6
c4	53.9	44.0	38.1	51.4	41.0
gcc	54.2	52.4	46.4	52.0	46.7
compress	1016.1	961.8	926.1	969.8	934.0
go	963.3	939.8	939.8	955.1	955.1

Table A.42: Conventional application elapsed run time (seconds) for for various page-table implementations.

BENCH	MPT	H512	H8192	C512	C8192
wave5	406.7	298.7	109.6	283.4	138.4
apsi	86.9	19.4	18.8	30.8	28.5
swim	12.1	10.6	3.4	10.4	4.0
tfftdp	3.6	1.4	0.7	2.4	1.0
mm	65.4	27.0	26.3	29.7	28.5
heapsort	0.5	0.2	0.2	0.2	0.2
nsieve	14.9	6.5	5.2	7.4	5.8
c4	20.5	11.3	6.1	16.8	8.0
gcc	11.4	9.2	3.6	8.5	4.1
compress	123.9	84.4	51.5	95.5	62.5
go	13.9	8.3	8.3	6.0	6.0

Table A.43: Time spent in TLB miss handling for conventional applications (seconds) for various page-table implementations.

BENCH	MPT	H512	H8192	C512	C8192
wave5	376504695	384802864	386781167	374208534	376674798
apsi	75044581	68357745	68349728	75927306	76089372
swim	6288219	6085202	6128383	6072149	6068797
tfftdp	3046876	2786379	2804925	3131140	3046739
mm	56430041	65837305	66086084	66683750	66328619
heapsort	389854	353938	352350	330598	330330
nsieve	11700733	13413347	13502236	13368370	13496671
c4	17119212	16698965	16652870	17126672	17145616
gcc	10247805	10261724	10179905	10079733	10240692
compress	153727121	159163064	158939744	153636438	153191722
go	9344026	16995695	17008224	9370528	9370942

Table A.44: Number of TLB misses for conventional applications for various page-table implementations.

BENCH	PAGE	SIZE	NULL
wave5	10330	106528	5034
apsi	557	32800	5591
swim	3626	53280	5082
tfftdp	1035	32800	5113
mm	1967	40992	5205
heapsort	1010	32800	5138
nsieve	1243	32800	4905
c4	1299	32800	4849
gcc	2391	40992	4781
compress	8931	98336	5409
go	202	24608	4922

Table A.45: Conventional application page-table size statistics for MPT.

BENCH	PAGE	SIZE	NULL
wave5	10330	165984	44
apsi	557	9056	9
swim	3626	58080	4
tfftdp	1035	16640	5
mm	1967	31552	5
heapsort	1010	16224	4
nsieve	1243	19936	3
c4	1299	20864	5
gcc	2391	38464	13
compress	8931	143008	7
go	202	8192	310

Table A.46: Conventional application page-table size statistics for H512.

BENCH	PAGE	SIZE	NULL
wave5	10330	166048	48
apsi	557	131072	7635
swim	3626	131072	4566
tfftdp	1035	131072	7157
mm	1967	131072	6225
heapsort	1010	131072	7182
nsieve	1243	131072	6949
c4	1299	131072	6893
gcc	2391	131072	5801
compress	8931	146848	247
go	202	131072	7990

Table A.47: Conventional application page-table size statistics for H8192.

BENCH	PAGE	SIZE	NULL
wave5	10330	124752	66
apsi	557	6960	23
swim	3626	43680	14
tfftdp	1035	12576	13
mm	1967	23808	17
heapsort	1010	12288	14
nsieve	1243	15072	13
c4	1299	15792	17
gcc	2391	28992	25
compress	8931	107520	29
go	202	6144	310

Table A.48: Conventional application page-table size statistics for C512.

BENCH	PAGE	SIZE	NULL
wave5	10330	124752	66
apsi	557	98304	7635
swim	3626	98304	4566
tfftdp	1035	98304	7157
mm	1967	98304	6225
heapsort	1010	98304	7182
nsieve	1243	98304	6949
c4	1299	98304	6893
gcc	2391	98304	5801
compress	8931	110304	261
go	202	98304	7990

Table A.49: Conventional application page-table size statistics for C8192.

BENCH	PAGE	SIZE	NULL
64	74	540704	101818
128	138	1064992	200058
256	266	2113568	396538
512	522	4210720	789498
1024	1034	6307872	1051130
2048	2058	10502176	1574394

Table A.50: SPARSE-PAGE benchmark page-table size statistics for MPT.

BENCH	PAGE	SIZE	NULL
64	74	10176	562
128	138	12224	626
256	266	16320	754
512	522	24512	1010
1024	1034	40896	1522
2048	2058	73664	2546
4096	4106	139200	4594
8192	8202	270272	8690

Table A.51: SPARSE-PAGE benchmark page-table size statistics for H512.

BENCH	PAGE	SIZE	NULL
64	74	133056	8242
128	138	135104	8306
256	266	139200	8434
512	522	147392	8690
1024	1034	163776	9202
2048	2058	196544	10226
4096	4106	262080	12274
8192	8202	393152	16370

Table A.52: SPARSE-PAGE benchmark page-table size statistics for H8192.

BENCH	PAGE	SIZE	NULL
64	74	9120	686
128	138	12192	878
256	266	18336	1262
512	522	30624	2030
1024	1034	55200	3566
2048	2058	104352	6638
4096	4106	202656	12782
8192	8202	399264	25070

Table A.53: SPARSE-PAGE benchmark page-table size statistics for C512.

BENCH	PAGE	SIZE	NULL
64	74	101280	8366
128	138	104352	8558
256	266	110496	8942
512	522	122784	9710
1024	1034	147360	11246
2048	2058	196512	14318
4096	4106	294816	20462
8192	8202	491424	32750

Table A.54: SPARSE-PAGE benchmark page-table size statistics for C8192.

BENCH	PAGE	SIZE	NULL
64	473	535379	99145
128	873	1016618	186066
256	1976	1906064	340857
512	3602	3467009	597146
1024	7755	6074932	980024
2048	15995	10561649	1563323

Table A.55: SPARSE-FILE benchmark page-table size statistics for MPT.

BENCH	PAGE	SIZE	NULL
64	547	13095	271
128	922	18038	205
256	2036	36268	230
512	3690	65571	407
1024	7680	135838	809
2048	15511	274028	1615

Table A.56: SPARSE-FILE benchmark page-table size statistics for H512.

BENCH	PAGE	SIZE	NULL
64	447	131291	7758
128	761	131720	7471
256	1990	134765	6432
512	4032	147763	5203
1024	7253	176770	3794
2048	15522	290040	2605

Table A.57: SPARSE-FILE benchmark page-table size statistics for H8192.

BENCH	PAGE	SIZE	NULL
64	579	11100	345
128	872	15569	424
256	2070	33555	725
512	3480	58968	1433
1024	8601	137546	2861
2048	16197	262917	5712

Table A.58: SPARSE-FILE benchmark page-table size statistics for C512.

BENCH	PAGE	SIZE	NULL
64	514	98634	7705
128	802	99172	7461
256	1799	102426	6736
512	3980	115722	5663
1024	8590	161124	4836
2048	16186	269355	6260

Table A.59: SPARSE-FILE benchmark page-table size statistics for C8192.

Page Table	MAP1	NSTD	MAPN	NSTD
G16+S8K	97.7	0.001	79.7	0.001
G16+S128K	95.6	0.000	78.0	0.001
MPT	109.6	0.002	67.3	0.002
H512	87.4	0.002	68.8	0.002
H8192	58.6	0.001	46.7	0.001
C512	78.2	0.002	65.4	0.003
C8192	57.0	0.002	45.4	0.001

Table A.60: Elapsed time results (in milliseconds) for MAP1 and MAPN benchmark, for various page tables.

BENCH	G16+S128K	NSTD	MPT	NSTD	H512	NSTD
64K	61.7	0.01	50.3	0.03	35.5	0.02
128K	70.3	0.02	58.2	0.02	42.5	0.02
256K	80.5	0.04	73.6	0.02	50.1	0.02
512K	92.7	0.05	104.3	0.01	68.6	0.03
1M	102.2	0.06	169.5	0.008	101.3	0.01
2M	100.9	0.08	293.7	0.004	129.8	0.009
4M	98.5	0.06	559.4	0.008	133.9	0.02
8M	102.6	0.05	1100.5	0.06	133.5	0.02
16M	107.6	0.05	1871.3	0.1	136.6	0.02
BENCH	H8192	NSTD	C512	NSTD	C8192	NSTD
64K	36.2	0.03	37.9	0.02	38.3	0.02
128K	44.0	0.03	46.2	0.03	46.6	0.03
256K	53.1	0.02	53.2	0.04	56.9	0.04
512K	74.2	0.02	64.5	0.04	73.0	0.03
1M	111.3	0.01	90.9	0.03	105.7	0.03
2M	185.3	0.007	138.7	0.05	167.5	0.02
4M	346.2	0.01	137.9	0.05	279.1	0.009
8M	579.4	0.002	141.9	0.03	464.6	0.005
16M	1041.5	0.002	155.0	0.01	784.3	0.003

Table A.61: Elapsed time results (in microseconds) for MAPS benchmark for various region sizes, for various page tables.

BENCH	G16+S128K	NSTD	MPT	NSTD	H512	NSTD
64K	63.5	0.01	55.0	0.02	37.9	0.08
128K	70.8	0.03	62.4	0.02	95.5	0.1
256K	84.2	0.02	78.1	0.01	245.9	0.05
512K	96.2	0.04	108.8	0.009	597.0	0.02
1M	104.0	0.06	173.4	0.006	1297.0	0.008
2M	104.3	0.07	296.4	0.004	2702.6	0.001
4M	102.8	0.06	572.9	0.008	2704.0	0.002
8M	105.5	0.06	1119.3	0.005	2705.1	0.002
16M	106.5	0.08	1871.0	0.1	2702.2	0.0006
BENCH	H8192	NSTD	C512	NSTD	C8192	NSTD
64K	38.4	0.02	41.3	0.1	40.2	0.01
128K	47.8	0.02	52.2	0.1	50.5	0.08
256K	65.8	0.02	121.7	0.1	62.6	0.04
512K	111.1	0.07	295.6	0.06	84.5	0.02
1M	186.9	0.008	667.1	0.02	132.8	0.01
2M	336.8	0.02	1577.5	0.003	217.0	0.01
4M	541.9	0.01	1577.5	0.003	391.8	0.004
8M	931.8	0.007	1580.2	0.003	826.6	0.007
16M	1705.8	0.003	1580.8	0.005	1450.6	0.006

Table A.62: Elapsed time results (in microseconds) for MAPS benchmark (with the extra 64M of mapped memory) for various region sizes, for various page tables.

Page Table	Time	NSTD
MPT	203.2	0.0002
H512	18.1	0.001
H8192	507.2	0.0001
C512	17.6	0.001
C8192	359.9	0.0001

Table A.63: Elapsed time results (in milliseconds) for task benchmark for various page tables.

Appendix B

A Detailed Look at GPT Implementation

The following detailed examination of GPT implementation is an excerpt from previous work [LE95].

B.1 GPT Parser

At first, we describe a GPT translation step in general, independent of concrete hardware (see Figure B.1). Here, v is the part of the original virtual address that is still subject to translation, and the pair (p, s) determines the page table (p: physical address, s: $\log_2$ of table size) that has to be used for the current translation step. The result of this step is either a new page table (p', s') and a postfix v' of v, or the data page (p', s') and offset v'.

The translations step starts by extracting u, the uppermost s bits of v. u is used for indexing the page table. The addressed entry specifies a guard g of variable size, i.e. possibly empty, which is checked against the remaining bits of the virtual address ($w = g$). When equal, the remaining v' is either used for the next level translation, or as the offset part. This operates as a shortcut, since not only u, but both u and w are stripped off the virtual address in one step; no table is necessary to decode w.

Note that the width of u, (determined by the page table size), may vary from step to step and that the size of w may differ from entry to entry.

In the following parts, we use $|x|$ to denote the bit length of a flexible bit string x. For improved clarity, we always use x' for an item that belongs to the next translation step (i.e., refers to the next lower level page table) and x for an item belonging to the current level. The operator $[a]$ refers to the guarded page-table entry at address a.

Assuming at first 32-byte page table entries (we hope to later reduce this to 16 bytes), one GPT translation step is:

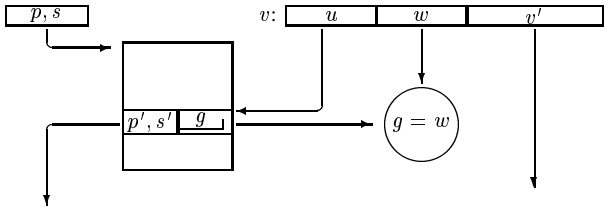


Figure B.1: Guarded Translation Step

$u := v \gg (|v| - s)$;
$g := [p + 32u]$.guard ;
if $g = ((v \gg (|v| - s - |g|))$ AND $(2^{|g|} - 1)$
  then $v' := v$ AND $2^{|v|-s-|g|} - 1$;
    $s' := [p + 32u]$.size′ ;
    $p' := [p + 32u]$.table′ ;
  else page_fault
fi .

This algorithm cannot be implemented 'as is', because the R4600 processor does not support flexible bit strings as a basic data type. Therefore, we have to hold $|v|$ and $|g|$ in additional variables v_{len} and g_{len}:

$u := v \gg (v_{len} - s)$;
$g := [p + 32u]$.guard ;
$g_{len} := [p + 32u]$.guard_len ;
if $g = (v \gg (v_{len} - s - g_{len}))$ AND $(2^{g_{len}} - 1)$
  then $v'_{len} := v_{len} - s - g_{len}$;
    $v' := v$ AND $2^{v_{len}} - 1$;
    $s' := [p + 32u]$.size′ ;
    $p' := [p + 32u]$.table′ ;
  else page_fault
fi .

After eliminating common subexpressions, this algorithm requires 17 arithmetic and load operations.

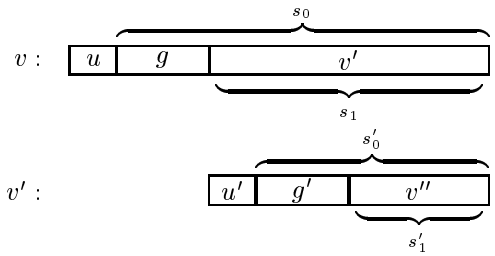
B.1.1 From 17 To 10 Operations

Note that although v is an input variable of the translation process, the length $|v|$ is a constant which is determined by the depth of the table in the GPT tree. Furthermore, the table size s and the guard length $|g|$ are fixed per page table entry. So the

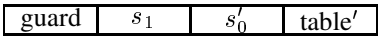
values

$$\begin{aligned} s_0 &= v_{len} - s \\ s_1 &= v_{len} - s - g_{len} \\ g_{mask} &= 2^{g_{len}} - 1 \end{aligned}$$

meaning



can be computed when constructing a GPT entry and can be stored per entry. Note that we have to store the actual level's s_1 but the *next level's* s'_0 in a page table entry:

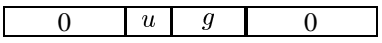


Fortunately, s'_0 can be as easily determined as s_0, as $s'_0 = v'_{len} - s' = v_{len} - s - g_{len} - s' = s_1 - s'$. The improved algorithm

$u := v \gg s_0$;
$g := [p + 32u]$.guard ;
$g_{mask} := [p + 32u].g_{mask}$;
$s_1 := [p + 32u].s_1$;
if $g = (v \gg s_1)$ AND g_{mask}
    then $v' := v$ AND $2^{s_1} - 1$;
        $s'_0 := [p + 32u].s'_0$;
        $p' := [p + 32u]$.table′ ;
    else page_fault
fi .

requires only 14 arithmetic/load operations and no longer needs the variable v_{len}.

The next optimisation is based on the idea of adjusting the guard bits in the GPT entry variable and extending it by the number u of this entry



so that XORing v by this field removes u and g in one step and avoids one shift and one add operation. More precisely, we store the *extended guard*

$$G = ((u \ll |g|) + g) \ll (|v| - s - |g|)$$

in each page table entry instead of the guard g. The resulting algorithm

$u := v \gg s_0$;
$G := [p + 32u]$.extended_guard ;
$s_1 := [p + 32u].s_1$;
if $(v$ XOR $G) \gg s_1 \ = \ 0$
    then $v' := v$ XOR G ;
        $s_0' := [p + 32u].s_0'$;
        $p' := [p + 32u]$.table ;
    else page_fault
fi .

requires only 10 arithmetic/load operations and avoids the per entry field g_{mask}.

Up to this point, we have looked at only one translation step. For a complete translation, a loop is required. To approximate an until-loop, we first move the then-part statements before the if statement. This is possible because these three statements do not destroy yet required data:

$u := v \gg s_0$;
$G := [p + 32u]$.extended_guard ;
$s_0' := [p + 32u].s_0'$;
$s_1 := [p + 32u].s_1$;
$p' := [p + 32u]$.table ;
$v' := v$ XOR G ;
if $v' \gg s_1 \ \neq \ 0$
    then page_fault
fi .

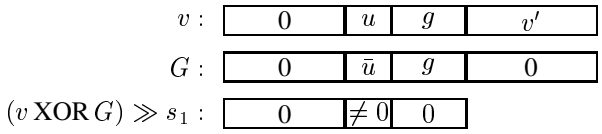
Unifying p', v' and s_0' with p, v and s_0, we get a very simple loop:

do
    $u := v \gg s_0$;
    $G := [p + 32u]$.extended_guard ;
    $s_0 := [p + 32u].s_0'$;
    $s_1 := [p + 32u].s_1$;
    $p := [p + 32u]$.table ;
    $v := v$ XOR G ;
until $v \gg s_1 \ \neq \ 0$ **od** ;

The loop terminates when a page fault, i.e. a guard mismatch, is detected. Of course, the translation process must also terminate in the positive case, i.e. if the

translation finishes without page fault. Adding a further termination condition to the loop would increase our costs per translation step.

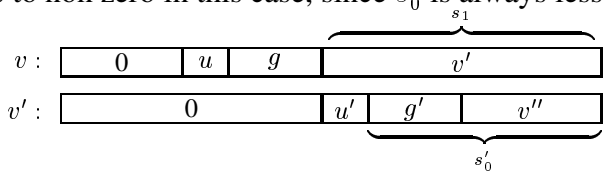
A better solution is to introduce a *pseudo mismatch* at leaf page table entries. We need an extended guard G, which includes the matching guard g, which in all cases leads to a mismatch, i.e. $(v \text{ XOR } G) \gg s_1 \neq 0$. Now recall that the extended guard of the uth entry of a page table always contains the index u. Therefore, we can achieve a pseudo mismatch by using an "incorrect" u for building the extended guard. $G = ((\bar{u} \ll |g|) + g) \ll s_1$ with $\bar{u} \neq u$ always leads to a mismatch:



The loop terminates either due to detecting a page fault or a leaf entry. In the case of

$$(v \gg s_1) \ll (64 - |g|) \;=\; 0\,,$$

we have a pseudo mismatch, i.e. a successful translation. For the mentioned check, we need a field holding the value $64 - |g|$. In leaf entries, the s_0'-field is free and can be used for this purpose. Then, $(v \gg s_1) \ll s_0'$ differentiates between true mismatch and pseudo mismatch, *if the current entry is a leaf entry*. We have to check, whether a mismatch at an higher level entry (which does not hold $64 - |g|$ in its s_0'-field) is also classified as a true mismatch. Fortunately, $(v \gg s_1) \ll s_0'$ evaluates always to non zero in this case, since s_0' is always less than s_1:



Concluding, the loop can be complemented by

if $(v \gg s_1) \ll s_0 = 0$
  then page_frame_addr := p ;
    page_frame_size := s_1
  else *page_fault*
fi .

so that in the case of successful termination, $s1$ determines the size and p the physical address of the page.

B.1.2 R4600 Implementation

Before presenting a concrete implementation of GPT parsing, a brief R4600 introduction is necessary. The R4600 is a member of the MIPS R4000 family of

processors which feature 64-bit integer and floating point operations. They have thirty-two general purpose 64-bit registers of which two are special. Register *r0* ignores writes and always returns zero when read. Register *r31* is used to store the return address of Jump And Link (JAL) instructions.

The R4600 has a primary 16KB instruction cache and a 16KB data cache on chip. Both caches are two-way set associative, use a 32 byte line size, and FIFO replacement within a set. Secondary cache is external and optional.

A four (64-bit) word write buffer is used to buffer writes to external memory arising from cache write-back, cache write-through, and uncached stores. This enables the processor to proceed in parallel while external memory is updated.

The R4600 has a five stage pipeline which has a one cycle latency for computational instructions. Computational instructions perform arithmetic, logical, and shifting operations using register operands or a register operand and a 16-bit signed immediate.

Load instructions don't allow the instruction immediately following, termed the *load delay slot*, to use the result of the load, thus giving a load latency of two cycles. Scheduling of instructions in the delay slot is desirable for increased throughput, though not strictly required, as the pipeline will slip one cycle in the case of a dependent instruction in the delay slot.

All jump and branch instructions have a latency of 2 cycles. The instruction in the *delay slot* following the jump is executed while the target of the jump is being fetched. The exception being if a conditional branch likely instruction is not taken, in which case the delay slot instruction is nullified.

B.1.3 From 11 To 8 Instructions

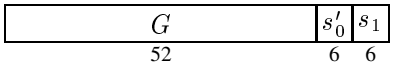
For the R4600 implementation, four 64-bit registers are needed. We name them r1, r2, v and P. A first compilation of the algorithm leads to 11 instructions per translation step:

```
do:
        shr     r2,v,r2         u := v ≫ s0
        shl     r2,5            32u
        add     P,r2            p + 32u
        ld      r1,[P].ext_guard
        ld      r2,[P].s0
        xor     v,r1            v := v XOR G
        ld      r1,[P].s1
        ld      P,[P].table
        shr     r1,v,r1         v ≫ s1
        bz      r1,do

shl     r1,r2                   (v ≫ s1) ≪ s0'
bnz     r1,page_fault
```

Note that all load delay slots in this (and the following) versions are filled with useful operations, i.e. do not cost additional cycles. By using appropriate coding[1], the same holds for the branch delay slot.

Further optimising, we use the fact that the R4600's minimal page size is 4K and the range of s_0' and s_1 is always $0 \ldots 63$. Therefore $2 \times 6 = 12$ bits are sufficient for s_0' and s_1 and since the 12 lower-most bits of G are never used, we combine these three fields in one 64-bit word:



The second 64-bit word is used for pointing to the next level table (or data page). By this, we avoid load instructions and reduce the page table entry size to 16 bytes. The resulting code

[1] Use the `bzl` instruction which nullifies the immediately following instruction if the branch is *not* taken:

```
        shr     r2,v,r2
do:     shl     r2,5
        ...
        bzl     r1,do
        shr     r2,v,r2
```

```
do:     shr     r2,v,r2         u := v ≫ s0
        shl     r2,4            16u
        add     P,r2            p + 16u
        ld      r1,[P]          r1 := (G, s0', s1)
        ld      p,[P].table
        xor[2]  v,r1            v := v XOR G
        shr     r2,r1,6         r2 := s0'
        shr     r1,v,r1         v ≫ s1
        bz      r1,do
```

requires only 9 instructions per translation step.

The instruction '`shl r2,4`' is somehow annoying, because it is only used for setting the 4 lowest bits to zero. Without this requirement, we could have stored $s_0' - 4$ instead of s_0' in the s_0'-fields so that the previous `shr` instruction already includes the multiplication with 16. Indeed, it is not necessary that the 4 lowest bits must be zero. It is sufficient that the 4 lowest bits of p after the addition have a *fixed value* which does not depend on the value of the actual v. This can be achieved by

```
xor    p,r2
```

instead of adding, provided that the 4 lowest bits of p are always 1111. Therefore, we store $p + 15$ instead of p in the table-fields and always use `P-15` instead of `P` for addressing a table or table entry.

```
do:     shr     r2,v,r2         r2 := v ≫ (s0 − 4)
        or      P,r2            p + 16u + 15
        ld      r1,[P-15]       r1 := (G, s0' − 4, s1)
        ld      P,[P-15].table
        xor     v,r1            v := v XOR G
        shr     r2,r1,6         r2 := s0'
        shr     r1,v,r1         v ≫ s1
        bz      r1,do
```

The final code requires only 8 instructions per translation step.

[2]Note that although '`xor v,r1`' destroys the 12 lowest bits of v (the 12 lowest bits of `r1` contain s_0' and s_1), it does not affect the algorithm, since these bits certainly belong to the offset part of the virtual address and are not required for translation.