Page Tables for 64-Bit Computer Systems

Kevin Elphinstone, Gernot Heiser
School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia
{kevine,gernot}@cse.unsw.edu.au http://www.cse.unsw.edu.au/~disy

Jochen Liedtke
IBM T. J. Watson Research Center
30 Saw Mill River Road, Hawthorne, NY 10532, USA
jochen@us.ibm.com

UNSW-CSE-TR-9804
August 1998

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Most modern wide-address computer architecture do not prescribe a page table format, but instead feature a software-loaded TLB, which gives the operating system complete flexibility in the implementation of page tables. Such flexibility is necessary, as to date no single page table format has been established to perform best under all loads. With the recent trend to kernelised operating systems, which rely heavily on mapping operations for fast data movement across address-spaces, demands on page tables become more varied, and hence less easy to satisfy with a single structure.

This paper examines the issue of page tables suitable for 64-bit systems, particularly systems based on microkernels. We have implemented a number of candidate page table structures in a fast microkernel and have instrumented the kernel's TLB miss handlers. We have then measured the kernel's performance under a variety of benchmarks, simulating loads imposed by traditional compact address spaces (typical for UNIX systems) as well as the sparse address spaces (typical for microkernel-based systems). The results show that *guarded page tables*, together with a software TLB cache, do not perform significantly worse than any of the other structures, and clearly outperform the other structures where the address space is used very sparsely.

1 Introduction

Most modern 64-bit microprocessors feature a software-loaded TLB, and thus leave the choice of the page table format to the operating system (OS) designers. This is a reflection of the fact that to date no single page table structure has been shown to provide the best performance in all relevant circumstances.

At the same time, address-space usage by operating systems is changing. In particular, *kernelised systems*, such as those based on Mach [1], make heavy use of virtual memory mappings in order to transfer data efficiently between different address spaces. This leads to sparsely populated address spaces, in contrast to the traditional two-segment model typical for UNIX systems. Sparse address space use is also typical for some other client-server based systems, such as object-oriented databases [2], or for single-address-space systems [3–5].

In order to get good performance out of such systems, it is important to use page tables which support efficiently the operations these kernels require. In this paper we examine these issues in detail. We use a representative architecture, the MIPS R4x00 family [6], and our own implementation of the *L4 microkernel* [7,8] as a testbed. L4 is presently the fastest kernel available [9]; its low intrinsic overhead makes it particularly sensitive to page-table performance, and therefore an ideal target for such an investigation.

The goal of this study is to examine how various page table structures perform under different conditions, as they are likely to exist in the next generation of computer systems. We will specifically attempt to determine whether there exist page tables which perform well under all anticipated loads. Such a structure would then be ideal for microkernels, which are supposed to present a platform on top of which a variety of different systems, traditional as well as novel, can be implemented.

2 Page Tables for 64-Bit Architectures

In this section we give a brief description of various page table structures in use, for more details see [10].

2.1 Linear and multi-level page tables

In most 32-bit systems *forward-mapped page tables* are used, which consist of page table entries (PTEs) containing physical frame numbers and which are sorted by virtual page number. A single *root page table* is enough to map all the pages of a page table covering a 32-bit address space. This root page is kept in unmapped memory.

In a *multi-level page table* (MPT) the most significant bits of the virtual page number are used as an index into the root page table. This contains a pointer to a secondary page table, which is then indexed with the remaining bits of the page number to find a PTE containing the physical address. If the secondary page is not mapped, a secondary fault occurs, which can be avoided by allocating the page table in physical memory. Secondary page tables are allocated as needed. This scheme is used on many 32-bit architectures, such as the SPARC and the Motorola 680x0. For larger address spaces more levels are required (six for a 64-bit address space with a 4kb page size and 4kb page tables).

An elegant simplification is the *linear page table*, LPT, which removes the need for the tree structure by allocating the page table for the full virtual address space as an array in virtual memory. Any PTE can be accessed by a single indexing operation into the page table, reducing the number of operations required, compared to the MPT. However, if the appropriate page table page is not mapped, a secondary fault is generated. In 32-bit systems the mapping for any page of the page table can be

obtained from the root page which is held in unmapped memory. This approach has been used in the VAX [11].

For larger address spaces, multiple misses can occur, up to six cascaded faults for 64-bits. This is unavoidable, as it is infeasible to hold the complete LPT in physical memory. Hence, the LPT is faster than the MPT only as long as all required portions of the page table are mapped. As the cost of a nested TLB refill is much higher than that of indexing the 5 higher-levels of an MPT, an LPT can outperform an MPT only if cascaded TLB misses are very infrequent. Furthermore, the necessity to handle up to 5 nested TLB misses on the page table requires TLB entries for the relevant pages. Hence a significant number of TLB entries are used up by the page table.

While both data structures work well in 32-bit address spaces, good performance in wide address spaces can only be expected as long as address space usage is compact.

2.2 Hashed page tables

Large-address-space architectures generally use page tables based on the idea of an *inverted page table* (IPT) [12, 13], which is a table sorted by frame numbers containing virtual page numbers. The index is obtained from a hash table. The more popular version combines the two tables, resulting in a table containing both, the virtual page number and the frame number [14]. It is indexed by some easily computable hash of the page number. Some data structure, usually a linear list, is used for resolving collisions. This structure is called a *hashed page table* (HPT).

Clustered page tables (CPTs) are a variant of HPTs designed to reduce space needs [15]. They store mapping information for several consecutive pages with a single tag.

2.3 Guarded page tables

The guarded page table (GPT) is a forward-mapped structure recently proposed by Liedtke [16, 17], specifically to deal with sparse memory usage in large address spaces. Sparsity is problematic in MPTs as it leads to page tables with very few (maybe only a single) valid mapping in intermediate nodes. This can at worst lead to excessive space overhead for page tables in extreme cases, and at best wastes TLB entries. GPTs avoid this problem by storing a *guard* with each pointer in the page table. If the virtual address to be translated is valid, this guard must be a prefix of the remaining address bits (after removing bits already used at higher levels of the page table). In the case of a match, the prefix can be removed as if it were used for indexing into a page table.

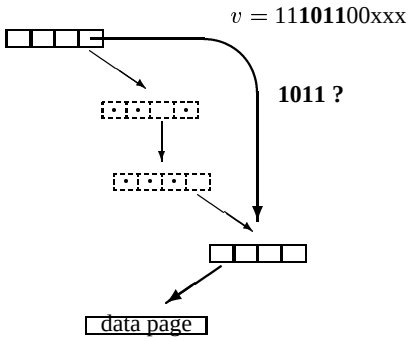

Figure 1: Guarded Page Table: Storing the guard **1011** together with the child pointer at entry **11** of the top-level table removes the need to store and traverse two intermediate page tables.

This is shown in Figure 1: Two intermediate page tables each contain a single valid entry, corresponding to address bits **10** and **11**. The GPT short-circuits these intermediate pages by storing the guard **1011** with the higher-level entry. The higher level is indexed with the leading bits (11), the guard bits (**1011**) are stripped, and the remaining bits serve as an index into the next lower page table.

An interesting property of GPTs is that page tables can be of variable size, anything from a two-PTE node to one as big as the hardware page size. A single GPT tree can arbitrarily mix node sizes, even on the same level. Multiple page size can also easily be supported.

2.4 Software TLB cache

Instead of just using a page table for handling TLB misses, a software cache for TLB entries, called *software TLB* (STLB) or *secondary TLB*, can be used [18]. The TLB miss handler first attempts to load the missing entry from the STLB and only on a miss consults the proper page table. This can significantly speed up TLB miss handling when using forward-mapped page tables.

Tagging the entries of the STLB with address space IDs allows sharing it between all processes. This not only reduces the amount of memory required for the STLB, it also reduces address-space setup and tear-down cost and context switching overhead.

3 Methodology

In order to investigate the performance implications of these page table structures, we have performed a number of experiments, running a set of benchmarks on an instrumented μ-kernel.

3.1 Test bed

As mentioned earlier, we use for our experiments the L4 μ-kernel [7, 8] running on a 100MHz MIPS R4700 processor [6]. The R4700 has a tagged, software-loaded TLB with 48 entries, each entry mapping a pair of contiguous virtual pages of 4kb each. The processor supports a 40-bit address space. However, we designed our page tables to work with full 64-bit addresses.

L4 is known for low system-call overhead and efficient IPC. The MIPS version takes <60 cycles to perform a null system call and <100 cycles for a null IPC [9].

We created several versions of the kernel, each using a different page table structure. For the study of absolute TLB miss handling costs, each TLB refill hander was carefully instrumented. The instrumentation code counts all cycles spent in TLB refill, except the cost of the initial fault, the return from the trap, and two instructions of the refill routine. The uncounted part of TLB refill, as well as the instrumentation overhead, were exactly the same for all page table versions. For the measurement of elapsed times (and thus the study of relative TLB miss handling cost) we used uninstrumented versions of the kernel.

3.2 Page table implementations

We implemented the following page tables:

MPT A multi-level page table. Its internal nodes contain 32-bit pointers, an optimisation possible by allocating the page table in high memory and using sign extension on 32-bit pointers. This allows internal nodes to hold 1024 pointers to lower-level page tables. Leaf nodes contain pairs of 4-byte PTEs as well as a pair of 4-byte mapping tree pointers. The latter are required

by the μ-kernel to manage mappings between (recursive) address spaces. 32-Bit leaf entries suffice to map 16TB of RAM, which is more than what the R4700 supports. In spite of these optimisations, a six-level page is required to map 64-bit addresses. The levels are indexed by bits 63–61, 60–51, 50–41, 40–31, 30–21 and 20–13.

G2–G256 GPTs with various node sizes, fixed for the whole page table. "Gn" has nodes holding n entries. Each internal entry consists of a 64-bit pointer and a 64-bit guard. The leaf nodes hold pointers to external nodes containing two MIPS format PTEs, plus two 32-bit pointers to the mapping tree. Leaf entries are therefore also 16 bytes in size.

H8k An 8kb hashed page table. It contains 32-byte entries consisting of a 64-bit tag, two 32-bit PTEs, two 32-bit mapping tree pointers, and a 64-bit pointer to a linear overflow chain. The refill routine moves the hit entry to the head of this list.

H128k A 128kb version of H8k.

C128k A 128kb CPT. Each bucket contains a 64-bit tag, eight 32-bit PTEs, eight 32-bit mapping tree pointers, a 64-bit pointer to the overflow list, and 48 bytes padding for proper alignment, for a total of 128 bytes. The overflow buckets contain the same fields, except no padding is necessary.

S8k/G16 An 8kb tagged direct mapped software TLB cache containing 16-byte entries consisting of a 64-bit tag and two 32-bit PTEs. The cache is backed by a G16 page table.

S128k/G16 A 128kb version of S8k/G16.

All page tables were allocated in physical memory. This is, of course, only realistic as long as the page tables remain relatively small. If that is no longer the case, page tables must be allocated in virtual memory, implying the occurrence of nested TLB misses, and thus reduced performance. Our μ-kernel presently cannot handle secondary TLB misses at all, which is why we could not implement an LPT.

3.3 Benchmarks

In order to examine the effect of page table structures on system performance under various application loads, we used a variety of standard and non-standard benchmarks. All benchmarks were run in memory, i.e., there was no I/O activity during the runs.

3.3.1 Conventional benchmarks

In order to examine the performance of traditional UNIX-like applications, we ran a subset of the SPEC95 benchmark [19]. The selected programs are characterised by high TLB miss handling overhead. In addition we used a number of other popular benchmarks from a collection maintained by Al Aburto [20]. Table 1 lists the benchmarks used. For simplicity we ran the `gcc` benchmark on a single input file only (`1amptjp.i`). Furthermore, we ran `mm` only using the "normal" algorithm, as the other algorithms do not exercise the TLB to any significant degree. All other benchmarks were run unmodified. In the following, this suite will be referred to as the "conventional" benchmarks.

name	*size [Mb]*	*type*	*remarks*
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
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 1: "Conventional" benchmarks used, the top six are from SPEC95. Type "I", "F" stands for integer or floating point, respectively. † indicates modification as explained in the text.

3.3.2 Sparse benchmarks

In order to evaluate the handling of sparse address use by the different page tables, we defined two sets of synthetic sparse memory benchmarks. The first one, called "*uniform*", allocates between 64 and 8192 single pages at uniformly distributed random addresses. This is obviously a "tough", and somewhat pathological benchmark, as the uniform distribution implies essentially no clustering of pages.

The second benchmark, called "*file*", allocates multi-page objects at uniformly distributed random addresses. The sizes of these objects are taken from a measured file system size distribution [21]. We expect this to be a more realistic model of sparse address-space usage in future 64-bit systems.

3.3.3 Tasking benchmarks

Task creation and deletion costs were measured by executing the loop

```
create task
wait for IPC from child
delete child
```

100 times. The child does nothing but send a null IPC to the parent; it requires only a single page to run.

3.3.4 Mapping benchmarks

Appel and Li [22] point out the importance of efficient virtual memory primitives. The primitive most relevant to this study is decreasing accessibility (unmapping or write protecting) of large regions of the address space (their PROTN operation).

We implemented a version of Appel & Li's benchmark, using processes, client and server. The server initially maps 128 consecutive pages into the client's address space. The benchmark times the cost of write-protecting all 128 pages and then unprotecting them all in random order. In the PROT1 version the pages are write-protected one-by-one, while the PROTN version protects them in a single system call.

We also implemented a sparse scenario: Again we protect and unprotect a total of 128 pages, but this time the pages are randomly (and uniformly) distributed in an otherwise unmapped region of varying size (64kb–16Mb). Two versions of this benchmark are implemented: In the first one the client only has a minimum set of pages mapped, only a code and a stack segment besides the pages being protected and unprotected. This is not a particularly realistic scenario, as a process is not likely to write-protect almost its whole address space (except for uses like checkpointing). The second version therefore has, in addition, a large (64Mb) data segment, which is not affected by the mapping operations.

4 Results

4.1 Optimal GPT size

As pointed out in Section 2.3, GPTs are a fairly general data structure; the design leaves a significant amount of flexibility to the implementation. So far, practical experience with GPTs is very limited. Liedtke [17] published a number of theoretical results mostly concerned with the behaviour of GPTs under worst-case conditions. No results on GPT sizing under practical conditions have been published to date.

For the purpose of comparing GPTs with other data structures we decided to use only GPTs of a fixed size. This means that there is a potential for improving our results for GPTs by implementing sophisticated sizing schemes; our results on GPTs therefore need to be understood as a scenario which might be optimised further.

We used the conventional and the sparse benchmarks to determine a single GPT size which would constitute a good compromise, suitable for comparison with the other data structures.

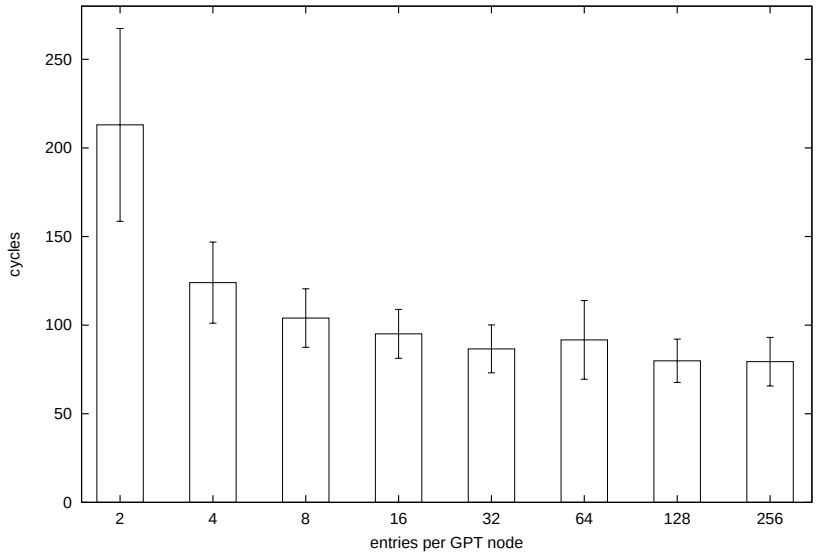

Figure 2: TLB refill cost over GPT size, averaged over all conventional benchmarks. Error bars indicate standard deviation.

Figure 2 shows the number of cycles used for TLB refills as a function of GPT size. These (as all refill costs further on) are what is measured by the instrumentation code, and therefore do not include

the time taken for the TLB miss trap and the return from the exception. Refill costs were averaged over all 11 conventional benchmarks. As expected, increasing node size reduces the refill cost as a result of reducing the tree depth. The improvement becomes small for node sizes exceeding about four.

Figure 3 shows the total size of the page tables, normalised to the number of PTEs stored, as a function of GPT node size. While the actual data depend on the particular program run, the general picture is the same: Very small GPT nodes result in deep trees, leading to high overhead. Very large nodes lead to high overhead, resulting from poor utilisation of the GPT nodes. In all benchmarks a 16-entry GPT has close to minimum space overhead.

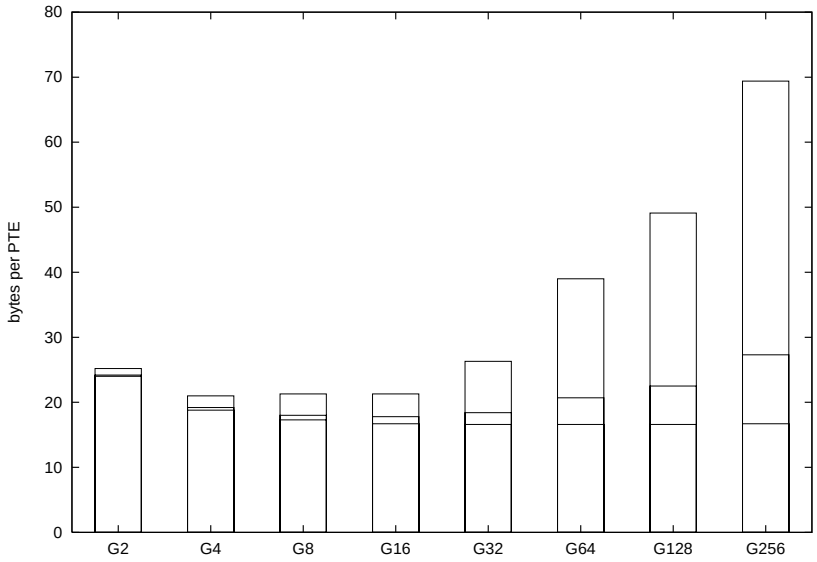

Figure 3: GPT space overhead over GPT size for the conventional benchmarks. The three values are the biggest overhead observed in our benchmarks (`go`), the average overhead observed, and the smallest overhead observed in our benchmarks (`wave5`).

If address space usage is very sparse, space overhead is minimised by small GPT nodes. This is shown in Figure 4, which shows the page table size overheads obtained from running the synthetic sparse benchmarks. Results fluctuated significantly (as indicated by the error bars) with the particular random sequence used. Interestingly, we found essentially no dependence of the size overhead (per PTE) on the number of pages mapped.

As expected, the overhead increases with increasing GPT node size. However, even in the rather pathological "uniform" case, the increase is much less dramatic as predicted by the theoretical worst-case scenario. In the more realistic "file" benchmark space overhead remains reasonable except for the largest GPTs. All in all the space overhead remains reasonable for node sizes $\leq$16: G16 space usage is about 2.5 times that of G2 for "uniform" and 1.7 times for "file".

Based on these results we selected a GPT node size of 16 for the comparisons with other page tables.

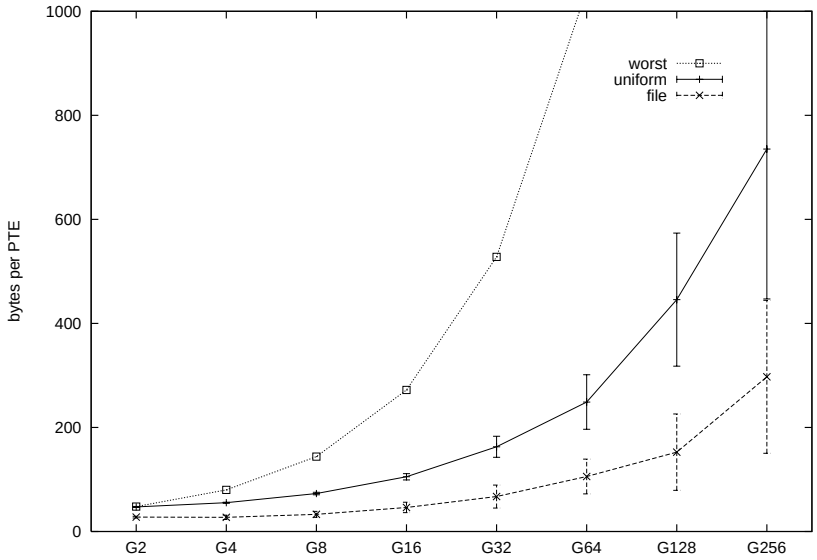

Figure 4: GPT space overhead over GPT size for sparse benchmarks, compared to the theoretical worst case. Error bars indicate standard deviation.

4.2 Comparison of page table structures

4.2.1 Conventional benchmarks

benchmark	MPT	G16	H8k	H128k	C128k	S8k/G16	S128k/G16
go	1.0	1.00	0.98	0.98	0.99	0.97	0.97
swim	1.0	1.00	1.00	1.00	1.00	1.00	1.00
gcc	1.0	0.97	0.97	0.86	0.86	0.91	0.84
compress	1.0	0.99	0.95	0.91	0.92	0.91	0.91
apsi	1.0	0.99	0.96	0.96	0.97	0.96	0.96
wave5	1.0	0.97	0.97	0.91	0.91	0.91	0.91
c4	1.0	0.95	0.82	0.71	0.75	0.76	0.69
nsieve	1.0	0.97	0.94	0.93	0.93	0.93	0.93
heapsort	1.0	1.00	0.99	0.99	0.99	0.99	0.99
mm	1.0	0.96	0.67	0.67	0.63	0.64	0.63
tfftdp	1.0	0.90	0.84	0.80	0.80	0.80	0.79
geom. mean	1.0	0.97	0.91	0.88	0.88	0.88	0.87

Table 2: Elapsed time of traditional benchmark runs for different page table structures, normalised to the time used with an MPT. The last line gives the *geometric mean* of all normalised execution times for a particular page table.

Table 2 shows the elapsed times of the conventional benchmarks using the various page table structures. For most programs the page table structure affects the run time by a few percent, however, in some cases (`mm` and `c4`) the page table can make a difference of more than 30 %. It is interesting to note that these are by no means large benchmarks (7.7MB and 5.1MB respectively), but are obviously the ones with the most “random” memory access patterns. Essentially the R4700’s TLB is too small

to support these applications well, and an efficient page table structure is hence important even for these traditional 32-bit applications.

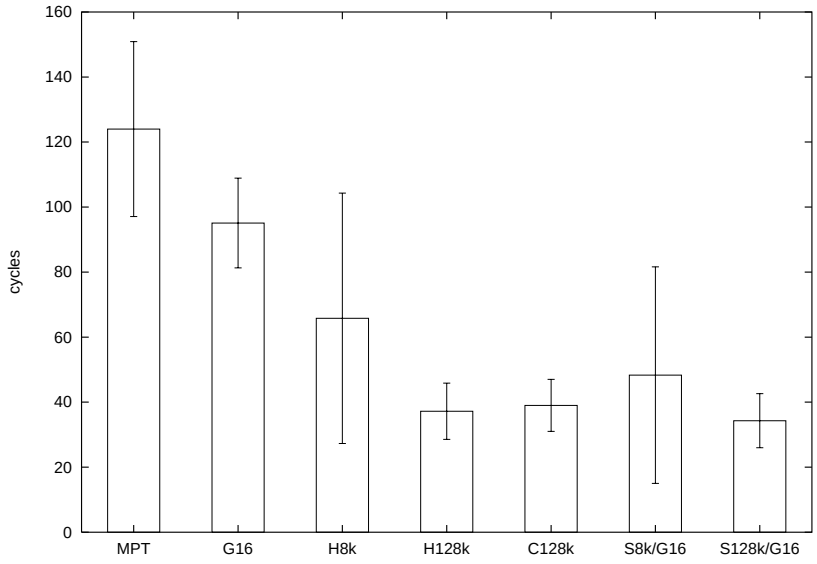

Figure 5: Number of cycles per refill, averaged over all traditional benchmarks, vs. page table structure.

The good news is that a software TLB can remedy the shortage of hardware TLB entries. This becomes clearer in Figure 5, which shows the average number of cycles spent on TLB refills for the different page tables. (The absolute minimum number, assuming no cache misses, would be nine cycles.) The figure shows the reload cost of GPTs being in between that of MPTs and HPTs. However, combining the GPT with a TLB cache performs even better than the HPT. The reason the STLB/G16 combination has a slight edge over the HPT is that the GPT can handle STLB misses faster than the linear overflow chain used with the HPT. We did not implement a STLB/MPT combination, but there is no reason to assume that its performance would be significantly different from STLB/G16.

The standard deviations (shown as error bars in Figure 5) indicate that, while even a small 8kb cache in average reduces TLB miss handling costs significantly, there are some programs where it has little effect. The `swim` benchmark actually performs better with a plain GPT than the 8kb HPT.

Figure 6 compares the space overhead of page tables. The GPT is by far the most compact structure (except H8k, which is only slightly bigger). The space required for a STLB was not included in the comparison, as the (tagged) TLB cache is a system-wide data structure shared by all processes, while all others are per-process. The important point to make here is that HPTs are fairly expensive space-wise, as they need to be large to ensure good performance, but waste significant amounts of memory for small processes. Clustering improves the size of the hashed page table somewhat, but it is still far higher than the hierarchical data structures.

4.2.2 Task creation/deletion

Memory consumption by itself is not a significant issue nowadays. However, the cost of a large data structure is not just the memory it consumes, but the time required to initialise it. This is reflected in Figure 7, which shows the cost of creating and deleting tasks with the different page tables.

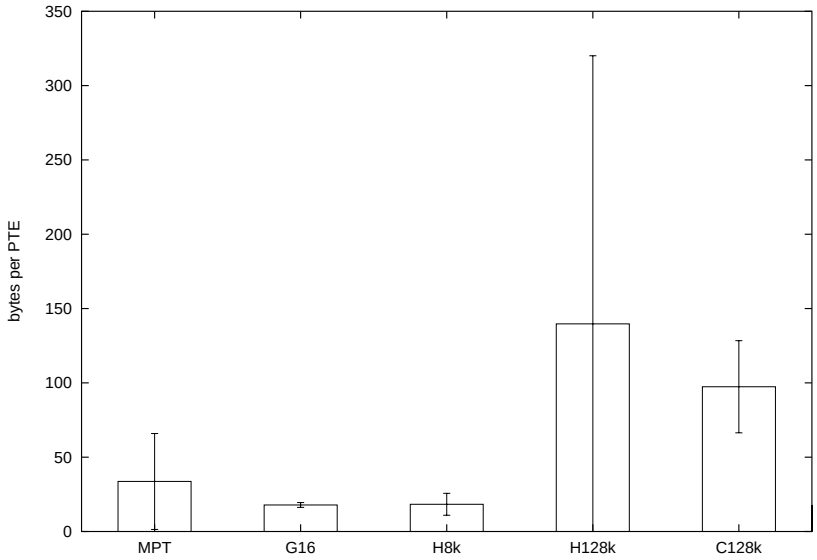

Figure 6: Average space overhead for various page table structures in traditional benchmarks.

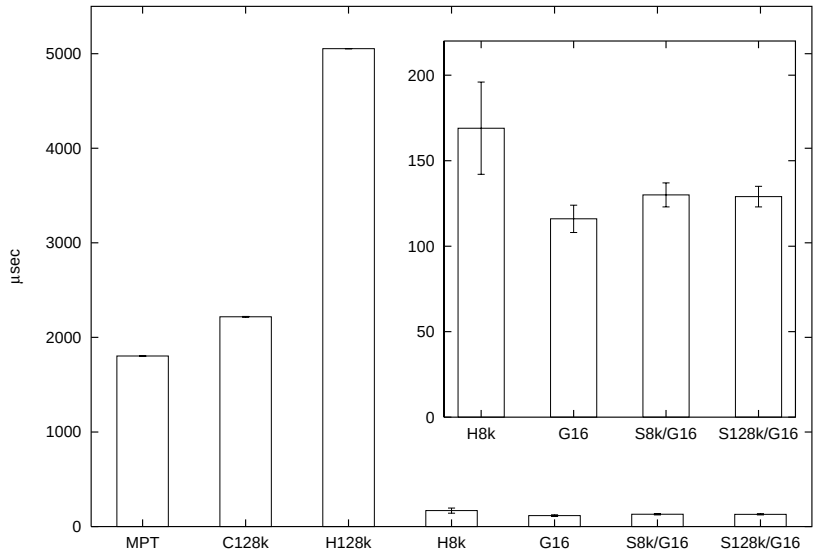

Figure 7: Cost of task creation and deletion for different page tables.

The results clearly show that page table setup/tear-down cost dominates task creation/deletion. GPTs have by far the lowest cost, the task creation/deletion cycle is about 70 % of that of the small HPT, and more than an order of magnitude less than that of the MPT, the CPT or the large HPT. The additional cost incurred through the TLB cache is minimal (about 10 %).

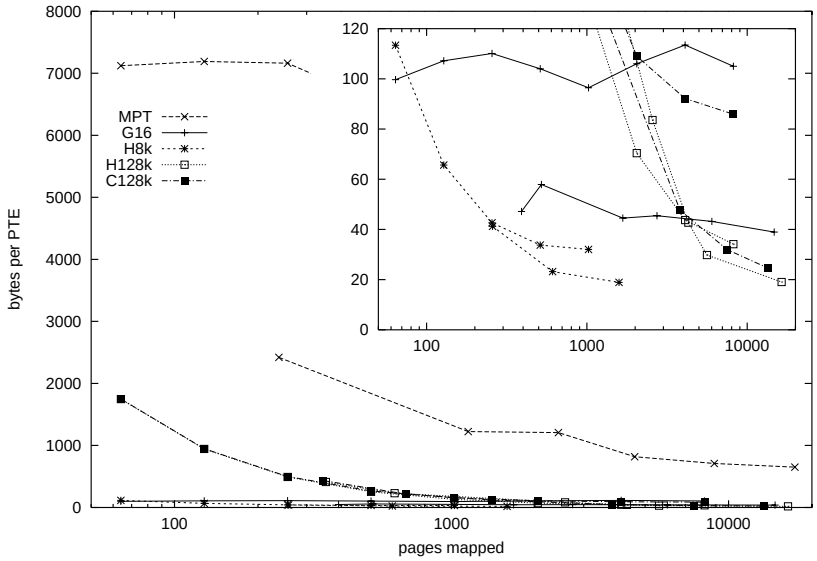

Figure 8: Page table size overhead (bytes/PTE) for the sparse benchmarks. For each page table structure, the upper line is for "uniform", the lower for "file".

4.2.3 Sparse memory use

Figure 8 shows the space overhead of the different page tables under very sparse memory use. MPTs are uniformly bad, consuming about two orders of magnitude more memory than GPTs. The small HPT (which becomes unusable at around 1000 pages because of excessive overflow chaining) uses the least memory. The GPT's space usage is quite stable at about twice the HPTs' best case (for "file"). The large HPT is space efficient only once the number of pages approaches the HPT's capacity. The behaviour of the CPT essentially mirrors the HPT.

4.2.4 Mapping operations

Figure 9 shows the cost per page of protecting and unprotecting all pages in a region of 128 pages. The HPTs perform best, the MPT worst, CPT and GPTs in between. The differences are not very substantial. Protecting all pages at once (PROTN) only reduces the cost by about 10–20 % compared to performing all operations on individual pages (PROT1). This reflects the very low system-call overhead of the μ-kernel.

Figure 10 shows the results of the sparse mapping benchmark, where the pages to be protected are sparsely allocated within a large region, and the task otherwise only has a minimum number of mapped pages. There is little difference between the various page tables as long as the region containing the pages is relatively small (note that a 64kb region is actually completely populated with the 16 pages). However, once the region becomes large, performance with MPTs as well as the large HPT and the CPT drops dramatically. The drop is less dramatic in the case of the small HPT, whereas the performance of the GPT-based schemes is essentially unaffected by sparsity.

Figure 11 shows the result of performing the same mapping operations on a task which has an additional compact data segment of 64Mb size. Here the small HPT performs worst, as it is too small to hold all the relevant mappings. MPT and large HPT performance are further degraded compared

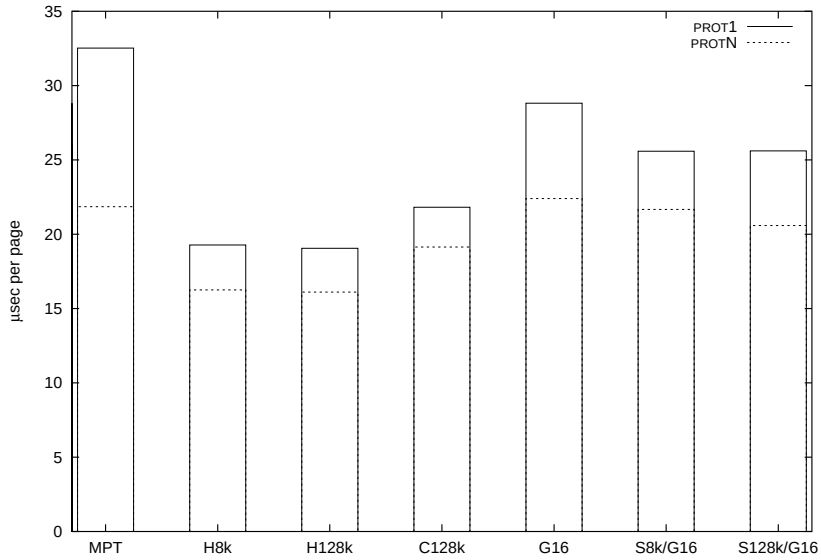

Figure 9: Compact mapping benchmark: Cost per page of write-protecting individual pages (PROT1) or a whole region (PROTN), and successively unprotecting page-wise.

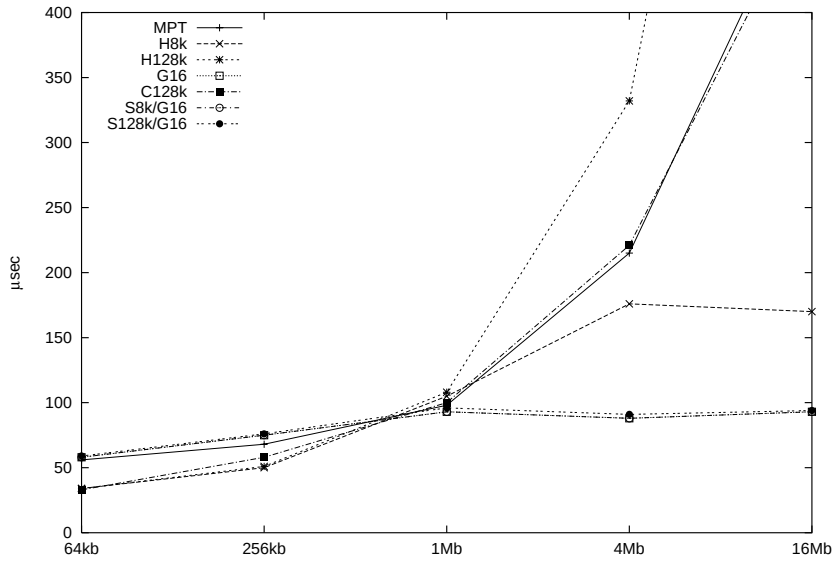

Figure 10: Sparse mapping benchmark: Cost of read-protecting a region of varying size containing 16 allocated pages at random locations. Only a minimum of other pages are mapped.

to the previous benchmark. Similar for the CPT, although the degradation is much weaker. The GPT-based schemes are essentially unaffected and perform consistently well.

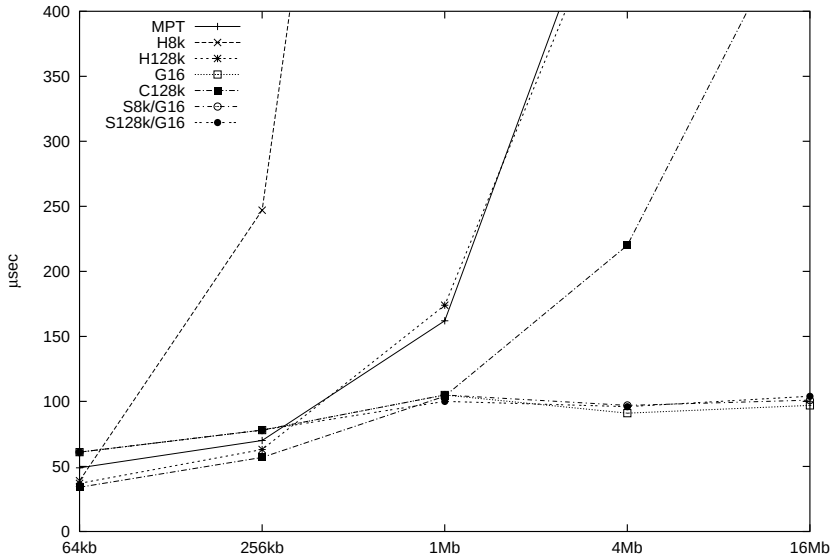

Figure 11: As Figure 10, except the client task also has an additional 64Mb segment mapped which is untouched by the mapping operations measured.

5 Discussion

5.1 Conventional benchmarks

Even with the reasonably small conventional benchmarks (all small enough to fit into physical memory on contemporary workstations) we found that the choice of the page table structure used in the kernel had a significant effect on performance. We found that only hashing schemes are competitive, as they support the fastest TLB refill routines. Furthermore, the hash table needs to be large for best performance.

However, we found that, as far as performance is concerned, it mattered little whether we used a HPT, a CPT, or a STLB reloaded from some hierarchical page table structure.

A large per-process hash table, as for the HPT or CPT, implies significant space overhead for the (mostly relatively small) conventional benchmarks. However, the space effects do not seem big enough to constitute a serious problem for traditional applications.

5.2 Task creation/deletion

These benchmarks showed that the set-up and clean-up costs of 6-level MPTs or large HPTs or CPTs have a disastrous effect on task creation and deletion overhead. Neither type of page tables are usable if inexpensive tasks are desired. As MPTs are also slower for refill, this essentially rules them out for 64-bit systems.

GPTs perform well, resulting from their ability to adapt: For small processes the GPT tree never grows beyond a few levels, but they can grow to support large processes as well.

5.3 Sparse address spaces

The sparse benchmarks confirm the unsuitability of MPTs for wide-address architectures. The large HPT is only competitive for reasonably large processes, the same holds true for the CPT. GPTs, however, perform very well. Their memory usage is at most about 2-3 times that of the (unrealistically small) 8kb HPT, and is clearly less than the small HPT for very small processes. Particularly in the more realistic "file" benchmark GPT space overhead is reasonable.

Recall that in Section 4.1 we opted for a 16-entry GPT as delivering the best size/speed tradeoff. However, this is only the best solution as long as a fixed node size is assumed. A GPT implementation which adapts the node size to the sparsity of virtual memory use would select two-entry GPTs in the sparse benchmarks and have a space overhead less than or equal to the unrealistically small H8k.

5.4 Mapping

The cost of mapping operations is only weakly affected by the page table structure, as long as address spaces are compact. However, as address space usage becomes sparse, performance deteriorates strongly when using MPTs. HPT and CPT performance is only acceptable if the size of the table is well tuned to the size of the mapped address space. Mapping operations on sparse parts of an address space which contains many other pages (unaffected by the mapping operations) is expensive with any size HPT. The CPT performs somewhat better but is still very costly. GPT performance, however, is essentially unaffected by sparsity, independent of whether or not the GPT is combined with an STLB. The GPT clearly meets its design goal of supporting sparse address space use.

5.5 LPTs

As mentioned earlier, we did not implement an LPT, because of the inability of our L4 implementation to handle nested TLB misses. However, we can speculate how LPTs would perform in our benchmarks.

LPTs will be faster than MPTs as long as we have a high TLB hit rate when accessing the page table. In most cases, such a high hit rate will be accompanied by a high hit rate on user pages, in which case TLB miss handling costs are irrelevant. The cost of handling a secondary TLB miss is of the order of 400–500 cycles [23].[1] As it costs, in average, about 110 cycles to traverse the five upper levels of the MPT, this implies that the LPT would outperform the MPT when the TLB hit rate for page table accesses is about 80 % (higher if higher-level TLB misses occur). Hence, it is possible that the LPT would perform somewhat better than the MPT on the conventional benchmarks. However, in the case of sparse address space usage, TLB miss rates on the page table are most likely quite high, and LPT performance is unlikely to be competitive.

Moreover, we already concluded that any forward-mapped page table should be combined with an STLB. Once that is done, there is unlikely to be a significant performance advantage in a LPT. Furthermore, LPTs have essentially the same space overhead as MPTs for sparse address spaces, and the same or worse setup/tear-down overhead. The same arguments which rule out MPTs as a suitable page table structure for 64-bit systems therefore apply to LPTs as well.

[1] Uhlig et al. measured the cost of a secondary miss on a MIPS R2000 CPU. The architecture of the R4700 is sufficiently similar to expect the costs to be roughly the same.

6 Summary

We have examined a number of different page table structures in order to determine their suitability for 64-bit computer systems. Our results show that traditional forward-mapped page tables, which have been used in most 32-bit systems, are no longer competitive on 64-bit architectures, particularly when sparse address space use is an issue, as in μ-kernel-based systems.

Hashing schemes, such as hashed page tables, generally exhibit good TLB refill costs, as long as they are large enough. However, large per-process hash tables have a high space overhead for small processes, and slow down process creation and deletion.

We found a system-wide software TLB cache, backed by per-process guarded page tables, to be the best solution. This scheme outperforms all alternatives in TLB refill costs, task creation/deletion overheads, and VM operations on sparse address spaces. It also exhibits the lowest space overhead for conventional (compact) 32-bit applications, while retaining a reasonably low space overhead, comparable to optimally sized hashed page tables, in cases of extremely sparse address usage. Our GPT implementation still leaves room for improvement, as it does not attempt to adapt GPT node sizes to address space usage. An improved implementation has the potential to further reduce space overhead for very sparse address spaces.

We conclude that GPTs form an excellent base for translation management in operating systems for 64-bit architectures, particularly in microkernels which are meant to support a wide class of operating systems and user environments.

References

[1] Richard Rashid, Avadis Tevanian, 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*, C-37:896–908, 1988.

[2] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):51–63, October 1991.

[3] Kevin Murray, Ashley Saulsbury, Tom Stiemerling, Tim Wilkinson, Paul Kelly, and Peter Osmon. Design and implementation of an object-orientated 64-bit single address space microkernel. In *Proceedings of the 2nd USENIX Workshop on Microkernels and other Kernel Architectures*, pages 31–43, September 1993.

[4] 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.

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

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

[7] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[8] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. *L4 Reference Manual — MIPS R4x00*. School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, December 1997. UNSW-CSE-TR-9709. Latest version available from http://www/cse/unsw.edu.au/~disy/.

[9] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Herrman Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for efficiency). In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 28–31, Cape Cod, MA, USA, May 1997. IEEE.

[10] 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*, pages 39–50. ACM, 1993.

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

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

[13] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and programming. *ACM Transactions on Computer Systems*, 6:28–50, 1988.

[14] John Rosenberg and David Abramson. MONADS-PC—a capability-based workstation to support software engineering. In *Proceedings of the 18th Hawaii International Conference on System Sciences*, volume 1, pages 222–31. IEEE, 1985.

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

[16] Jochen Liedtke. A basis for huge fine-grained address spaces and user level mapping. In *Proceedings of the 7th European Conference on Object Oriented Programming (ECOOP) Workshop on Granularity of Objects in Distributed Systems (GODS'93)*, Kaiserslautern, Germany, July 1993.

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

[18] Kavita Bala, M. Frans Kaashoek, and William E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 243–253, Monterey, CA, USA, 1994. usenix.

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

[20] Al Aburto. Benchmark collection. URL ftp://ftp.nosc.mil/pub/aburto.

[21] Kevin Elphinstone. Address space management issues in the Mungi operating system. Technical Report UNSW-CSE-TR-9312, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 1993.

[22] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, 1991.

[23] Richard Uhlig, David Nagle, Trevor Mudge, Stuart Sechrest, and Joel Emer. Instruction fetching: Coping with code bloat. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 345–56, Santa Margherita Ligure, Itay, June 1995. ACM.