Variable Radix Page Table: A Page Table for Modern Architectures

Cristan Szmajda[1] and Gernot Heiser[1,2]

[1] School of Computer Science and Engineering
University of New South Wales
Sydney 2052, Australia
[2] National ICT Australia, Sydney, Australia
{cls,gernot}@cse.unsw.edu.au

Abstract. This paper presents a new page table structure, the *variable radix page table*, which overcomes many of the disadvantages of other page table structures. Unlike a hashed page table, the variable radix page table naturally accommodates shared segments and mixed page sizes. But unlike a multi-level page table, the radix page table is space-efficient and requires few memory references to look up, even in large and sparse address spaces. Our measurements show that the variable radix page table outperforms other page table structures, and is even competitive with a memory-based TLB cache.

Recent research has shown that thrashing of the TLB is an increasing bottleneck in modern processors: measurements of the TLB's contribution to execution time often exceed 40%. Such results sometimes even understate the full impact of TLB thrashing due to the presence of indirect overheads such as cache pollution and the effect of exceptions on the processor pipeline. By reducing the cost of TLB misses, the variable radix page table can achieve a significant overall speedup. The variable radix page table's mixed page size support also facilitates the reduction of TLB miss frequency, addressing the architectural imbalance that causes TLB thrashing. Our conclusions are also significant in the debate on the different hardware organizations in use for virtual memory.

1 Motivation

1.1 Performance

Virtual memory (VM) is almost universally supported in modern architectures and operating systems. VM has many benefits and one main cost: the overhead in time and space for maintaining and looking up page tables. In most processors, page table look-ups are cached in a hardware translation lookaside buffer (TLB). Historically, TLBs have been very effective. Clark and Emer [1] measure the contribution of TLB misses to execution time of a variety of workloads on the VAX-11/780 in the range 5–8%.

However, more recent measurements show that the TLB overhead in modern processors is surprisingly high. Huck and Hays [2], Romer et al. [3], Subramanian et al. [4], and Navarro et al. [5] all report TLB overheads which often exceed

40%. Using simulations, Kandiraju and Sivasubramaniam [6] report data TLB miss *rates* in excess of 20% for some benchmarks, which translates to an extremely high overhead given that the typical cost of a TLB miss is 30 cycles or more.

What is the cause of such poor TLB performance results? The TLB is another casualty of the widening gap between processor and memory speed. Processor speed and memory sizes have been increasing steadily, but the coverage of the TLB much more slowly. Navarro et al. observe that the coverage of the TLB ten years ago was in the order of 1% of the main memory size: today it is in the order of 0.01%. There are architectural reasons why the TLB has been growing slower than the rest of the memory hierarchy. The TLB is a virtual cache, so it usually requires wider content-addressable memory (CAM) tags. The TLB must be frequently invalidated, so larger sizes bring diminishing returns. To avoid context switch invalidations, each tag is often widened further by an address space identifier. Because of their relatively small size and potential for pathological misses, the TLB often has high associativity already: as a result, TLBs are difficult to build simultaneously large, fast, and cool. Shared pages also effectively increase the amount of physical memory without reducing the number of TLB entries required to cover it. The cost of each individual TLB miss is also increasing due to deeper pipelining, the overhead of handling precise exceptions, and page tables for 64-bit address spaces.

Inertia must also be blamed. Once specified in the architecture, TLB parameters (such as the page size) are often difficult to change without introducing incompatibilities with existing system software. There is also some neglect. Much attention is paid to enhancing caches with prefetching, multiple levels of hierarchy, and better integration with the processor pipeline. The TLB does not receive nearly as much attention as the rest of the memory hierarchy.

1.2 Superpages

An easy way to improve TLB coverage is to increase the page size. However, an architecture's page size often cannot be changed in an upwards-compatible manner. Larger pages also increase fragmentation and I/O latency. Therefore, many architectures instead provide multiple page sizes: a base page and one or more *superpages*, which are power-of-two multiples of the base page size. A *superpage TLB* allows pages of different sizes to be used simultaneously, even in the same address space (provided, of course, that they do not overlap). Most current processors provide superpage TLBs to extend TLB coverage rather than attempting to build larger conventional TLBs.

The first applications of superpages in operating systems were special-purpose: kernel virtual memory, memory-mapped I/O devices, and mappings which bypass address translation. Superpages are ideal for these applications as their mappings are usually large, contiguous, and long-lived.

The more general use of superpages is inhibited by assumptions pervading the operating system. The whole VM subsystem, in its page fault handling, page replacement algorithms, and free frame management generally assumes throughout that all pages are the same size. Several approaches for the general use of superpages for ordinary applications have recently been presented, with varying extents of modification to existing operating system software. Romer et al. count TLB misses to small pages, promoting them when the miss count exceeds a threshold. Ganapathy and Schimmel [7] use a background daemon which promotes pages based on memory pressure and hints specified by the user in the executable. However, because both these schemes still allocate small pages to noncontiguous frames, the pages must first be copied into a contiguous region before promotion. Copying is such a performance penalty that Swanson, Stoller, and Carter [8] have proposed adding another TLB at the DRAM interface to optimize it. Subramanian et al. avoid copying altogether: they reserve contiguous memory for superpages by using a free frame manager based on the buddy system. Navarro et al. also use the page reservation technique, and have solved many practical issues using this scheme.

However, while several approaches exist for making use of superpages in higher layers of the VM subsystem, very little advantage of superpages is taken in the page table. Most page tables support only one or two page sizes directly. Other page sizes must typically be expanded into multiple page table entries (PTEs), each with the coverage of the base page size. For example, if the base page size is 8 kBytes, a 4 MByte superpage would require 512 PTEs. Not only does this duplication waste space, but it can make superpage promotion and demotion operations expensive. It is also a poor match for page size assignment policies which use a single page size for each process or for each segment. With most page tables, these simple policies would suffer much from PTE duplication.

1.3 Sharing

Many systems allow physical frames to be shared between address spaces. This is used for shared code segments, shared libraries, memory-mapped I/O, and interprocess communication. However, most page tables also have poor support for shared segments. The PTEs for the shared segment are usually duplicated in every page table. A page table which could represent shared segments without duplication would save space, be easier and faster to update, and friendlier to caches.

A shared page shares the same physical memory and physically-indexed cache lines, but not TLB entries. With typical address space identifier (ASID) tags, a separate TLB entry is still required for each address space which shares a page. Sharing therefore increases the coverage of memory and physically-indexed caches, but not the TLB. This makes TLB performance worse relative to the rest of the memory hierarchy. Some TLBs have more sophisticated tags which allow TLB entries to be shared between address spaces [9, 10]. The tag identifies not an individual address space but a whole *protection domain*, of which more than one address space may be a member. Whether or not the TLB supports domains directly, the motivation for improving the performance of shared segments in the page table is clear.

Another effect of shared segments is that the page tables occupy more space relative to the size of physical memory. According to Khalidi and Talluri [11], unshared page tables for shared pages can increase page table size by an order of magnitude. Since many operating systems cannot page out page tables, page table space can be a significant problem.

2 Previous Page Tables

The most common page table structures in current use fall into two broad categories: those based on radix trees and those using hashing.

2.1 Radix-Based Page Tables

Probably the most common page table structure is the *multi-level page table* (MLPT), which is essentially a shallow radix tree. To look up an MLPT, the page number is split into m fields which are used to index m arrays in turn (see Fig. 1). The choice of m and the size of the arrays is a time–space tradeoff. Choosing fields of about 10 bits wide yields a page table of manageable size. With 32-bit addresses, only two or three levels are required.

But with full 64-bit addresses, a manageable MLPT would require at least five levels. Looking up a 5LPT would require five sequential memory references: an unacceptably large overhead. Moreover, a 5LPT is not particularly space efficient either. Segments of memory scattered sparsely throughout the address space would require many arrays to be allocated, which are mostly empty space. These problems make MLPT impractical to use with 64-bit addresses. Nevertheless, the new AMD x86-64 architecture defines a 4LPT, but restricts virtual addresses to 48 bits.

MLPT enables some simple optimizations. Since an aligned superpage of appropriate size would fill a whole leaf page table level, a common technique is to factorise it into a single leaf entry in the upper level page table. Fig. 2 shows the effect of this representation: indexing is truncated when enough bits of the page number have been translated to uniquely identify the superpage. However, the number of different page sizes that can be supported with this technique is limited. Typically the minimum supported superpage size is several MBytes, and all other page sizes must be represented by duplicating PTEs. Nevertheless, superpage factorization is common in systems using MLPT.

Another optimization to MLPT is to share whole segments by cross-linking page tables. However, similar limitations apply to this technique: each shared segment must be isolated in its own aligned region of several MBytes. If many small segments must be shared, the amount of virtual address space wasted by putting each into its own aligned region is considerable. Few systems using MLPT cross-link page tables.

The virtual linear array (VLA) page table is a variation of MLPT which mirrors the multi-level structure, but allocates the page tables in virtual memory. The page table thus appears to be a single large array indexed by virtual page

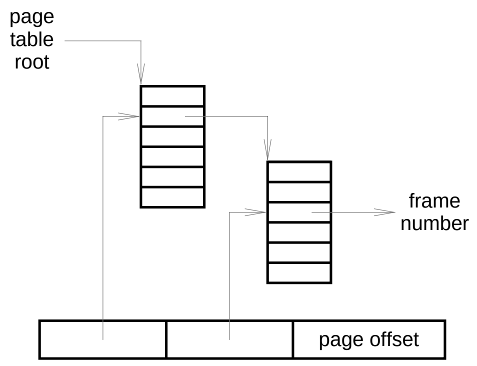


Fig. 1. A two-level page table

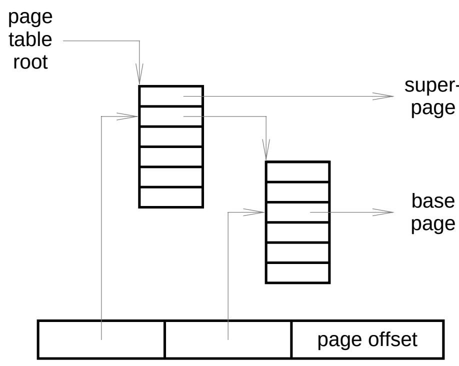


Fig. 2. Superpage representation in a two-level page table

number. Storing page tables in virtual memory effectively allows interior nodes to be cached in the TLB, short-circuiting page table look-up in the common case to a single (virtual) memory reference. Only if all levels miss in the TLB does the full m-level look-up take place. However, unless the necessary control is included in hardware, each miss requires an expensive nested exception to handle. TLB entries for the virtual linear array also compete with TLB entries for applications. Finally, the virtual linear array consumes a large swath of virtual address space: a finite resource. Nevertheless, VLA page tables are popular, and some architectures provide support for them either directly or by assistance with nested exceptions or multiple exception vectors.

2.2 Hashing-Based Page Tables

An alternative to ordinary page tables is the *inverted page table* (IPT), which is indexed by physical, not virtual, address. An IPT is particularly attractive with 64-bit addresses because its size is proportional to the size of physical memory, not the size of the virtual address space. Moreover, only one IPT is required for all address spaces in the system, and the reverse function (physical to virtual) is provided automatically. But unless hardware is available to search it in parallel, the IPT requires an additional *hash anchor table* to look up the IPT by virtual address. Another disadvantage of IPT is that sharing a frame between multiple address spaces is hard.

Huck and Hays [2] show that a hash table on its own performs slightly better than IPT, and removes some of its disadvantages. Their *hashed page table* (HPT) is essentially a hash table with collision chaining (Fig. 3). A HPT is somewhat larger than other page tables because the virtual address must be present in each PTE to check for hits and collisions.

The *clustered page table* (CPT) proposed by Talluri, Hill, and Khalidi [12] is a variation of HPT which stores multiple adjacent PTEs per hash bucket. This combines both hashing and radix-based approaches, applying hashing on the sparse high-order bits and array indexing on the dense low-order bits. A CPT may also use less space than a pure HPT, as only one virtual address needs to be stored per hash bucket, instead of one per PTE.

Hashing-based page tables are attractive for 64-bit address spaces as a hash function is just as efficient for large virtual addresses as small ones. Sparse address space distributions are also no problem. However, hashing has some undesirable properties. Collisions cannot be eliminated, even with a perfect hash function. Hash tables are also difficult to traverse: consider the problem of deleting an address space whose PTEs are scattered throughout an HPT. Finally, hashing does not support superpages well: common workarounds are to try multiple hash functions or to expand superpages into many base page size PTEs.

Nevertheless, the advantages of HPT are such that many recent architectures, wishing to include page table support in hardware for performance but not wishing to set an inflexible page table in stone, have included hardware support for HPT, either as a complete page table or as a cache to accelerate another page table implemented in software.

2.3 Path Compression

Path compression is an established technique for reducing the depth of radix trees [13]. Path compression works on chains of non-branching nodes, whose entries are all invalid except for one. Every such chain is abbreviated into a single entry containing just the tail of the chain and an indication that several bits of the key were skipped on the path. The skipped bits may also be stored in the entry and checked during the look-up; alternatively keys may be stored in the leaves and checking deferred till the end.

The *guarded page table* (GPT) is a multi-level page table which applies path compression (Fig. 4). In the terminology of Liedtke [14], each node is 'guarded' by skipped bits, which must compare equal with the page number key before descending further in the page table.

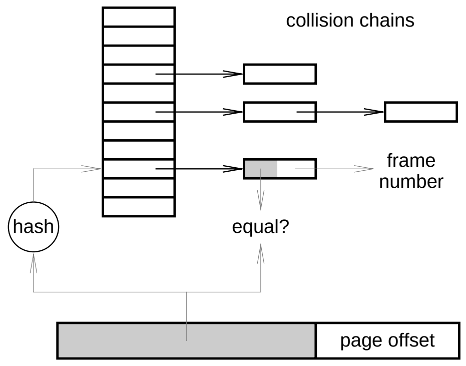


Fig. 3. A hashed page table

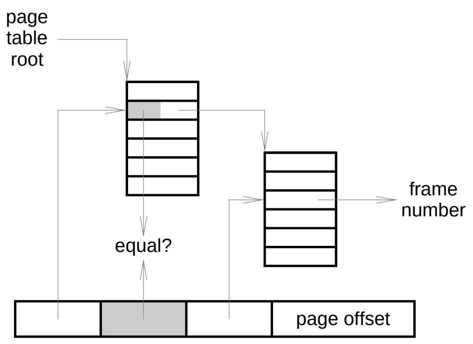


Fig. 4. A guarded page table

The GPT works well in sparse address spaces [15]. Since very few applications use more than a tiny fraction of a 64-bit address space, path compression is effective at reducing radix tree depth, especially in higher levels of the radix tree.

However, implementations of GPT have not performed well in large and dense address space regions. While Liedtke originally contemplated a GPT with variable radices, all implementations have to date used a fixed radix. Moreover, to take advantage of sparsity, and to reduce the cost of GPT updates, this radix is typically small. Elphinstone [15] determined that the optimal GPT radix over a range of look-up, creation, and deletion benchmarks is 16. As a result, GPT depth can sometimes blow out, especially for large contiguous segments.

3 Radix Page Table

3.1 Level Compression

Level compression [16] is another technique for reducing the depth of radix trees. It works by reducing complete subtrees, whose nodes are all valid, reducing them to a single, flat super-node.

The *variable radix page table* (VRPT) is a radix tree which applies both path compression and level compression (Fig. 5). Each node may skip any number of bits which are insignificant, and point to an array of any power of two in size. Each level may be a different size, and the depth of the tree may be different at different regions of the address space.

VRPT also dispenses with the need for actual guards. Instead, checking the validity of a look-up is deferred until a leaf is reached. This optimistic strategy is reminiscent of hashing, where key comparison only occurs after the hash algorithm selects a bucket. Omitting guards also greatly simplifies VRPT updates, allowing the easy restructuring of VRPT levels to arbitrary powers of two.

The combination of path compression and level compression is more effective at reducing page table depth than either technique alone. Flexible radices allow VRPT to take advantage of any regularity or structure present in the address space layout to reduce the number of required levels. There is no need to choose a compromise radix that suits both sparse and dense address spaces. If an address space contains a mixture of sparse and dense regions, the level size and depth may even be different in the different regions.

Sparse address space distributions contain many compressible paths, especially near the root of the page table. Path compression is therefore most effective near the page table root. Level compression is effective at the leaves, though is sometimes also effectively applied near the root for certain address space layouts. Large dense segments benefit the most from level compression. Regularly laid out address spaces typically require only two or three levels, even in 64-bit address spaces.

Indexing an VRPT may seem to be a complex operation, but in fact the data structure and look-up algorithm can be made simple and reasonably efficient using bit arithmetic (Fig. 6). Each page table index is extracted with two

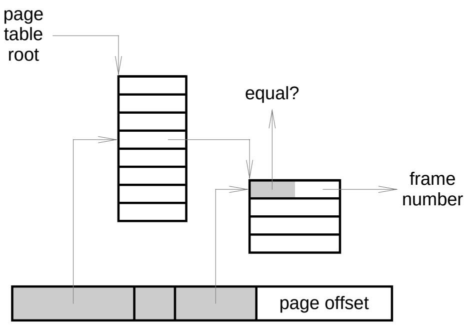


Fig. 5. A variable radix page table

```
$p$ = root of page table
$v$ = virtual address

**repeat** {
    $p = p \rightarrow ptr + (v \ll p \rightarrow skip \gg p \rightarrow size)$
} **until** $p$ points to a leaf

**if** $p \rightarrow virt \neq v$ {
    page fault
}
```

Fig. 6. VRPT indexing algorithm

instructions: a variable left shift followed by a variable right shift. An internal node consists only of a pointer and two shift amounts. The left shift amount, which is related to the number of bits skipped before indexing, is called *skip*. The right shift amount, which is related to the size of the following level, is called *size*. A bias is applied to *size* so that the index is 'pre-scaled' to the size of page table entries. Bits skipped in the page number are simply ignored during indexing, and are checked at the end. With a typical RISC instruction set, the inner loop can be implemented in only 7 or 8 instructions.

3.2 Radix Policy

The difficult part of implementing a flexible radix tree is choosing the radices. Nilsson and Tikkanen [17] promote or demote a level to the next power of two if it fills up or empties below certain thresholds. Elphinstone [15, p. 36] proposes using knowledge supplied from higher layers about the structure of the application. However, in the absence of such information it is difficult to guess such structure in advance from the lowest layers of the VM subsystem. Moreover, address space structure may evolve during the life of a process. The choice of radix is also a time–space tradeoff, and depends on the resources available to the system. A system with low memory pressure may be able to apply level compression more aggressively, whereas a system which is suffering from paging I/O may decide that aggressive level compression is not worth it.

Instead, our implementation of VRPT uses a radix policy which balances memory pressure against the desire for shallow page tables. The page tables use a smart allocator, based on the buddy system, which 'greedily' allocates the largest levels possible with the available contiguous physical memory. However, it reserves the right to take back some or all of the memory if it is unused and more memory is needed. This policy requires co-operation from the code which updates page tables, but the amount of extra complexity is small. The benefit is that room is provided for growth wherever possible, avoiding expensive tree restructuring operations as an address space is populated.

This policy also solves a subtle garbage collection problem. If higher layers unmap pages for a short time, the memory allocator may incur significant expense if the page tables are eagerly freed and quickly reallocated again. The VRPT allocator frees such page tables lazily, only when required. The VRPT allocator also integrates easily with resource management mechanisms designed to limit the amount of kernel memory which can be consumed by each user.

4 Benefits

In addition to look-up performance, the VRPT structure has some other beneficial properties. Mixed page sizes, shared segments, and many common virtual memory update operations are simple and easy to implement relative to other page table structures.

4.1 Superpages

Section 2 described how MLPT can incorporate limited page size mixtures in an efficient way. VRPT also naturally accommodates page size mixtures, but is much more flexible, allowing the operating system to achieve the maximum benefit of superpages.

Many architectures provide a large number of page sizes, in multiples of 4 or 8. VRPT supports arbitrary page size mixtures by *superpage factorization*: superpages may be treated as variable-length keys in the radix tree, and compressed into small leaves which truncate look-up at any point in the tree. Factorization saves space and allows superpage PTEs to be quickly updated.

Factorization in VRPT is always optional. Our implementation avoids factorizing superpages if the factorized version requires as much space as the expanded version (or more). This heuristic is not motivated by a compulsion to save memory but by the observation that a smaller page table is usually also simpler and shallower. In practice this superpage expansion generally only occurs if a superpage is promoted in-place from a population consisting mostly of smaller pages. In this case, full factorization only occurs when enough pages have been promoted to justify it. This policy also avoids the pathological situations which can occur with certain page size mixtures.

The one drawback of superpage factorization is that it can increase page table depth slightly. However, for this effect to become significant, a very large number of page sizes must be present, in which case the performance benefit of the large pages far outweighs the slight increase in page table depth.

4.2 Cross-linking

VRPT supports cross-linked page tables in a similar manner to the MLPT. But with VRPT, cross-linking is more flexible. Shared segments may be aligned on any power-of-two boundary, not just at a fixed boundary such as 4 MBytes.

Cross-linked page tables can eliminate *minor page faults*, which occur when there is a page fault on a page which is already resident in memory. A common reason for minor page faults is when shared pages are brought into the page cache by another application but not entered into every application's page table. If the shared segment is mapped by a cross-linked page table, this problem goes away.

Compared with the TLB miss handler, the page fault handler is usually a 'slow path' through the system, and is not as aggressively optimized. For this reason, minor page faults have a significant performance cost in many systems, even though they involve no actual I/O. Linux has recently addressed this problem by optimizing its page cache with a radix tree [18]. FreeBSD has instead added code to preload page tables from its page cache on address space creation [19]. Cross-linked page tables achieve the same effect as page table preloading without the added start-up cost.

4.3 Page Table Updates

When higher layers update page tables, they often do so in particular ways. The VRPT structure allows some simple optimizations for these common update patterns.

One common operation is to map, unmap, or change the protection attributes of an entire segment of memory at once. The VRPT structure allows the internal nodes to contain protection bits which qualify all underneath PTEs. These qualifying protection attributes may be accumulated during look-up by adding one extra AND instruction to the VRPT indexing loop: an instruction which may be free on a multiple-issue processor. Protection operations on whole memory segments may be implemented by modifying the internal node rather than every leaf PTE. When combined with sharing, address spaces may share segments with different protection attributes.

Another common operation is to update or invalidate all the PTEs which refer to a particular physical frame. For example, the page daemon may write-protect or pageout a frame from all address spaces. In some systems the number of PTEs referring to one frame may be large due to features such as shared libraries and the indiscriminate use of copy-on-write. Cross-linked page tables can help, but can not always be used, for example if segments are shared at different addresses or are inappropriately aligned. Searching for all the PTEs may be expensive. In some systems it is not even known at pageout time which address spaces contain a particular frame, and a brute force search is required.

A simple enhancement allows such operations to be optimized. A single pointer is added to each PTE which links together all the PTEs referring to the same physical frame. An operation on all virtual pages for a single physical frame may simply follow the linked list to find all affected PTEs. While this technique is not specific to VRPT, the inclusion of the virtual address in the VRPT leaf allows common operations such as pageout to proceed by following this list without reference to any external data structure. Other systems have a similar data external structure, such as the `pv_entry` structures in FreeBSD, which are actually used as machine-independent templates for generating the hardware page tables [19]. Merging this data structure into the page table itself saves considerable complexity at the cost of only one word in each PTE.

5 Performance

VRPT was benchmarked on two 64-bit architectures: MIPS64 and IA-64.

MIPS64 is a typical RISC architecture: all TLB misses are handled by trapping to software. Its TLB has been extensively studied, for example by Chen, Borg, and Jouppi [20]. Seven page sizes from 4 kBytes to 16 MBytes are supported. The hardware and experimental methodology is comparable with that of Elphinstone [15]: a 100 MHz MIPS R4700 processor running benchmarks selected from the SPEC CPU95 suite.

IA-64 is an architecture for high-performance servers, and includes many architectural features supporting fast virtual memory. The TLB is tagged by a domain (called a *protection key*). Eleven page sizes from 4 kByte to 4 GByte are supported. TLB refill hardware (called the virtual hash page table, VHPT) can be configured in one of three ways: as a hardware-walked VLA page table (short-format), as a hardware walked TLB cache (long-format), or disabled, with all TLB misses handled by software exception handlers. Our test platform used a 733 MHz Intel Itanium processor, using a custom kernel created for the purpose.

5.1 Methodology

A common methodology used in the literature is to count TLB misses and instrument the TLB miss handler, either in hardware or software. Instrumentation usually does not include effects such as the disruption of the processor pipeline and the cache misses due to displacement of cache lines by the TLB miss handler. More complex instrumentation could conceivably measure these indirect overheads, but it would be difficult to verify both the correctness of the measurements and the absence of other introduced overheads.

A better methodology which includes all the direct and indirect costs of TLB misses is to perform a second run of each benchmark in physical mode, with the TLB disabled. (Where the machine does not allow physical addressing in user mode, the effect can be simulated with very large superpages.) The relative difference between the two times is the TLB overhead.

Indirect costs are the probable explanation of the anomalous observation of Subramanian et al. [4] that the speedup due to superpages exceeded the total apparent TLB overhead.

5.2 Comparison between Page Tables

VRPT was implemented as a kernel module in the L4/MIPS microkernel [21]. For comparison, several other page table modules are also available in L4/MIPS. Fig. 7 shows the results. HPT, CPT, and GPT are implementations of the page tables described in Section 2. The implementation of GPT is also described in detail by Liedtke and Elphinstone [22]. An MLPT implementation was also available, but performed so poorly that it was excluded to avoid distorting the graph. CACHE is the same as GPT, but page table look-ups are accelerated by a 128 kByte software-maintained TLB cache in main memory [23].

On the MIPS, VRPT outperforms other page tables in terms of TLB miss overhead by a factor of two. VRPT is competitive with CACHE, even though the TLB cache in this benchmark achieves close to 100% hit rate. The number of memory references made by each page table is the main determiner of performance. Because these benchmarks are compact in their use of the address space, both CACHE and VRPT touch only one cache line per TLB reload. HPT and CPT must touch at least two to compute the hash and resolve collisions. GPT typically requires three or more levels.

Our results are generally consistent with the conclusion of Huck and Hays [2] that page table look-up is dominated by cache misses. In their simulations, 20–35% of the cycles spent in handling TLB misses are cache miss penalty cycles. They also compute cache miss rates of 2–3% for a 2LPT root, 12–15% for a 2LPT leaf, and 20–50% for an HPT. The advantages of hashing must be balanced against its reduced locality.

Not surprisingly, TLB performance is also highly sensitive to the locality of the application. *Swim*'s data segment is large, but its working set is relatively small. *Gcc* is smaller, but touches a large number of pages, perhaps due to the preponderance of algorithms operating on linked data structures.

An application's locality behaviour may benefit different levels of the memory hierarchy differently. We find little correlation between TLB miss rate and L1 cache miss rate, L2 cache miss rate, or page fault rate. This makes it difficult to compare TLB coverage with cache sizes, or to recommend a TLB size.

On the Itanium, (Fig. 8), two other page tables were also measured for comparison: 3LPT is Linux 2.5.43, which uses a three-level page table (but does not supports the full 64-bit address space). Linux was patched to use the HPT (as a TLB cache) rather than the VLA, as the latter does not support the use of super-

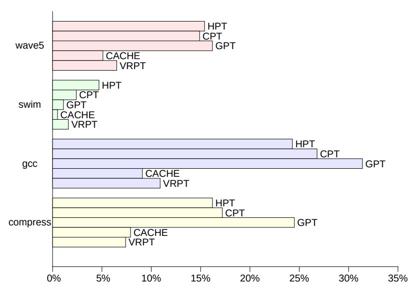


Fig. 7. Page table performance (MIPS64)

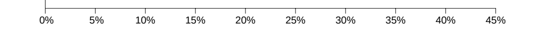
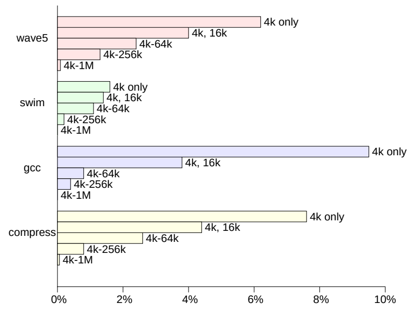


Fig. 8. Page table performance (IA-64)



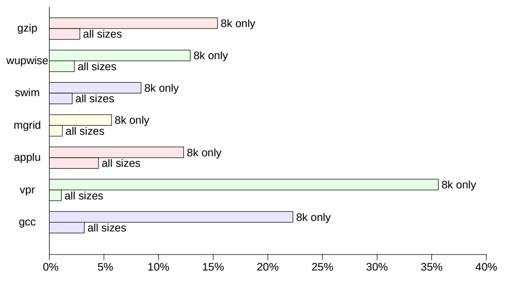


Fig. 9. The effect of superpages (MIPS64)

Fig. 10. The effect of superpages (IA-64)

pages. (The overall performance difference between the two hardware-supported page table formats has been found to be negligible [9].) HPT is FreeBSD 5.0, which uses software TLB reload from a hashed page table.

Every effort was made to minimize the inaccuracy due to the different operating system kernels. All executables were statically linked with the same version of the standard libraries. These libraries emulate most system functionality such as file I/O. The executables were preloaded into the system's page cache by performing a dummy run before benchmarking. The benchmarks themselves are designed to have very little interaction with the system. Nevertheless, some lingering system impact may be present in these results.

Because Itanium features a hardware-loaded TLB, the results in Fig. 8 are somewhat less pronounced. The majority of hardware TLB misses hit in the memory-based VHPT, and do not require page table lookup. Moreover, the cost of taking an exception is a large component of the TLB miss overhead on this processor.

5.3 The Effect of Superpages

One of the benefits of VRPT is its ease in accommodating superpages. In order to establish the benefit of using superpages, VRPT was benchmarked against itself in one of several different configurations. The *4k only* or *8k only* configuration uses only one (base) page size. The other configurations automatically promote smaller pages to superpages where possible.

The use of 64 k and larger superpages all but eliminates the TLB overhead in these benchmarks. This shows that TLB thrashing on the MIPS processor is dominated by TLB capacity misses: conflict misses are all but eliminated by the fully-associative TLB, despite the pseudo-random replacement algorithm.

Results for the Itanium processor are similar. However, a problem with using superpages on the Itanium is that the VHPT cannot cache superpages effectively: only the base page size is cached. Therefore, all TLB misses for superpages become VHPT misses. Despite the high cost of VHPT misses on this processor, superpages reduce the hardware TLB miss rate so dramatically that they are still a win.

A limitation in these benchmark is that it assumes that memory pressure is low, and that there is no paging I/O activity. We may expect different results if the system decides to run the application partly resident in memory.

6 Conclusions

Architectural imbalances have resulted in VM overheads forming a significant bottleneck for many workloads. The contribution of TLB misses to execution time often exceeds 40%. We have attempted to address this problem to the extent possible in software, using commodity hardware.

Page table look-up is the most important algorithm to optimize in software. We have presented the variable radix page table, which attempts to minimize the depth of page table look-ups. Measurements show that page table look-up time is dominated by cache misses.

Superpages present the single biggest opportunity for reducing TLB overhead by extending the coverage of the TLB and thus reducing the frequency of misses. VRPT is ideal for VM subsystems that use superpage optimizations. It provides efficient support for page size mixtures without restricting the way they are used. However, further analysis is required to determine whether sophisticated policies are required to take advantage of superpages, or whether simple policies such as increasing the page size across the board is sufficient.

Another open question is whether the hardware complexity of a superpage TLB is justified given the complexity this adds to operating system software. Clearly a configurable page size is a very desirable feature to allow future expansion or scaling to large memory configurations, but the benefit of supporting arbitrary page size mixtures must be evaluated against simpler TLB configurations. In particular, a TLB with a per-process page size and/or a sub-block TLB [24] may be attractive to compare with a superpage TLB.

References

1. Clark, D.W., Emer, J.S.: Performance of the VAX-11/780 translation buffer: Simulation and measurement. ACM Trans. Comp. Syst. **3** (1985) 31–62
2. Huck, J., Hays, J.: Architectural support for translation table management in large address space machines. In: Proc. 20th ISCA, ACM (1993) 39–50
3. Romer, T.H., Ohllrich, W.H., Karlin, A.R., Bershad, B.N.: Reducing TLB and memory overhead using online superpage promotion. In: Proc. 22nd ISCA, Santa Margherita Ligure, Itay, ACM (1995) 176–87
4. Subramanian, I., Mather, C., Peterson, K., Raghunath, B.: Implementation of multiple pagesize support in HP-UX. In: Proc. 1998 USENIX Techn. Conf., New Orleans, USA (1998)
5. Navarro, J., Iyer, S., Druschel, P., Cox, A.: Practical, transparent operating system support for superpages. In: Proc. 5th USENIX OSDI, Boston, MA, USA (2002)

6. Kandiraju, G.B., Sivasubramaniam, A.: Going the distance for TLB prefetching: An application-driven study. In: Proc. 29th ISCA, Anchorage, USA (2002)
7. Ganapathy, N., Schimmel, C.: General purpose operating system support for multiple page sizes. In: Proc. 1998 USENIX Techn. Conf., New Orleans, USA (1998)
8. Swanson, M., Stoller, L., Carter, J.: Increasing TLB reach using superpages backed by shadow memory. In: Proc. 25th ISCA, ACM (1998) 204–213
9. Chapman, M., Wienand, I., Heiser, G.: Itanium page tables and TLB. Technical Report UNSW-CSE-TR-0307, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia (2003)
10. Wiggins, A., Tuch, H., Uhlig, V., Heiser, G.: Implementation of fast address-space switching and TLB sharing on the StrongARM processor. In: 8th ACSAC, Aizu-Wakamatsu City, Japan, Springer Verlag (2003)
11. Khalidi, Y.A., Talluri, M.: Improving the address translation performance of widely shared pages. Technical Report TR-95-38, Sun Microsystems Laboratories, Mountain View CA (1995)
12. Talluri, M., Hill, M.D., Khalid, Y.A.: A new page table for 64-bit address spaces. In: Proc. 15th ACM SOSP, Copper Mountain Resort, Co, USA (1995) 184–200
13. Morrison, D.R.: Patricia: Practical algorithm to retrieve information coded in alphanumeric. Journal of the ACM **15** (1968) 514–534
14. Liedtke, J.: Improving IPC by kernel design. In: Proc. 14th ACM SOSP, Asheville, NC, USA (1993) 175–88
15. Elphinstone, K.: Virtual Memory in a 64-bit Microkernel. PhD thesis, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia (1999) `http://www.cse.unsw.edu.au/~disy/papers`.
16. Andersson, A., Nilsson, S.: Improved behavior of tries by adaptive branching. Information Processing Letters **46** (1993) 295–300
17. Nilsson, S., Tikkanen, M.: Implementing a dynamic compressed trie. In Mehlhorn, K., ed.: 2nd WS. Alorithmic Engin. (1998) URL `http://www.nada.kth.se/~snilsson/public/papers/dyntrie`.
18. Corbet, J.: Kernel development. Linux Weekly News (2002) `http://lwn.net/2002/0207/kernel.php3`.
19. Dillon, M.: Design elements of the FreeBSD VM system. Daemon News (2000) `http://www.daemonnews.org/200001/freebsd_vm.html`.
20. Chen, J.B., Borg, A., Jouppi, N.P.: A simulation based study of TLB performance. In: Proc. 19th ISCA, ACM (1992)
21. Elphinstone, K., Heiser, G., Liedtke, J.: L4 Reference Manual: MIPS R4x00. School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia. (1997) UNSW-CSE-TR-9709.
22. Liedtke, J., Elphinstone, K.: Guarded page tables on MIPS R4600 or an exercise in architecture-dependent micro optimization. Technical Report UNSW-CSE-TR-9503, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia (1995)
23. Bala, K., Kaashoek, M.F., Weihl, W.E.: Software prefetching and caching for translation lookaside buffers. In: Proc. 1st USENIX OSDI, Monterey, CA, USA, USENIX/ACM/IEEE (1994) 243–253
24. Talluri, M., Hill, M.D.: Surpassing the TLB performance of superpages with less operating system support. In: Proc. 6th ASPLOS. (1994) 171–182