Appears in Proceedings of SOSP'95

A New Page Table for 64-bit Address Spaces

Madhusudhan Talluri[*†], *Mark D. Hill*[*], *Yousef A. Khalidi*[†]

[*]Computer Sciences Department
University of Wisconsin
Madison, WI 53706, USA
{talluri, markhill}@cs.wisc.edu

[†]Sun Microsystems Laboratories
2550 Garcia Ave., MTV-29
Mountain View, CA 94043, USA
{madhu,yak}@eng.sun.com

Abstract

Most computer architectures are moving to 64-bit virtual address spaces. We first discuss how this change impacts conventional linear, forward-mapped, and hashed page tables. We then introduce a new page table data structure—*clustered page table*—that can be viewed as a hashed page table augmented with subblocking. Specifically, it associates mapping information for several pages (*e.g.*, sixteen) with a single virtual tag and **next** pointer. Simulation results with several workloads show that clustered page tables use less memory than alternatives without adversely affecting page table access time.

Since physical address space use is also increasing, computer architects are using new techniques—such as *superpages, complete-subblocking,* and *partial-subblocking*—to increase the memory mapped by a translation lookaside buffer (TLB). Since these techniques are completely ineffective without page table support, we next look at extending conventional and clustered page tables to support them. Simulation results show clustered page tables support medium-sized superpage and subblock TLBs especially well.

1 Introduction

One long-standing computer trend is that programs' memory usage doubles each year or two [Henn90]. Theoretically, therefore, systems that support paged virtual memory [Denn70] should increase their virtual and physical address sizes linearly each year. In practice, however, compatibility issues force address spaces to grow discontinuously, especially for virtual addresses. The industry is currently undergoing such a discontinuity with most processor architectures moving from 32- to 64-bit virtual addresses (while more-modestly increasing the size of physical addresses). All things being equal, increasing address space size adversely affects page table and translation lookaside buffer (TLB) cost or performance. This paper explores the effect of 64-bit addresses on page tables. An *address* is a virtual address unless we explicitly identify it as a physical address.

A *page table* stores translation, protection, attribute and status information for (virtual) addresses [Huck93, Chan88, Levy82, Silh93, Lee89]. The information for each page is called a *page table entry* (PTE). The TLB miss handler accesses the page table on a TLB miss to load the appropriate PTE into the TLB. An ideal page table would facilitate a fast TLB miss handler, use little virtual or physical memory, and flexibly support operating systems in page table modifications. Section 2 reviews conventional page tables—linear, forward-mapped, and hashed—and discusses the challenges of extending conventional page tables to support 64-bit address spaces. It explains why both linear and hashed page tables are viable for 64-bit addresses, and why forward-mapped page tables are probably impractical as each TLB miss requires about seven memory references. Many processors now support TLB miss handling in software, *e.g.*, MIPS [Kane92], Alpha [Site93], UltraSPARC [Yung95]. This makes page table design an operating system issue and in this paper we explore alternate operating system data structures for storing page tables and servicing TLB misses.

Section 3 introduces the central contribution of this paper: the *clustered page table*. It is a new page table data structure that can be viewed as a hashed page table augmented with *subblocking*, a simple but effective technique used in hardware caches and TLBs [Lipt68, Good83, Hill84, Tall94]. Hashed page tables associate a tag with every PTE. Clustered page

This work was primarily supported by a National Science Foundation Presidential Young Investigator Award (MIPS-8957278) and a Sun Microsystems External Research Grant. The experiments were performed on equipment donated by Sun Microsystems.

tables associate a single tag for an aligned group of consecutive pages (*e.g.,* sixteen 4KB pages), called a *page block*. Clustered page tables are effective when spatial locality makes it likely that consecutive pages are in contemporaneous use. For the assumptions given in Section 3, for example, a clustered page table with 16 pages per page block uses less memory than a hashed page table if six or more pages are populated. Experimental results (Figure 9) show that clustered page tables use less memory than the best conventional page tables—linear page tables for dense address spaces and hashed page tables for sparse address spaces.

Hardware designers are increasing the effectiveness of TLBs for 64-bit systems using techniques such as *superpages* [Tall92] and *subblocking* [Kane92, Tall94]. These techniques are very effective at improving TLB performance, reducing the number of TLB misses by 50% to 99%, and providing an average execution time speedup of upto 20% for the workloads we use [Tall95]. However, without support in the page table to store such PTEs or in the TLB handler to traverse such page tables, these TLB techniques are completely ineffective. Page tables that support conventional single-page-size TLBs also can use superpage or subblock techniques to reduce page table size by an order of magnitude (Figure 10) and get better cache performance.

Sections 4 and 5 present the second contribution of this paper: extending page tables to support superpage and subblock PTEs. We suggest replicating PTEs at each base site as a way to extend any conventional page table to support the new PTE formats without affecting TLB miss penalty. We also present alternate solutions that have some drawbacks but are usable in specific situations. Section 5 then shows how clustered page tables are ideal for supporting medium superpages or subblocks, as they result in smaller page tables, while retaining fast TLB miss handling time and flexibility.

Section 6 gives preliminary performance numbers simulating ten 32-bit workloads. We show that clustered page tables use less memory than any other page table and are faster to access when using superpage or subblock PTEs. Clustered page tables, for example, use 50% of the memory required by hashed page tables for our workloads (Figure 10). We simulate several TLB and page table organizations to calculate the estimated page table access time. The appendix includes formulae for estimating the number of cache misses during TLB miss handling and page table size for different page tables.

Section 7 includes a discussion on some tradeoffs, extensions, and optimizations to the page table organizations described in the paper. Section 8 reiterates our contributions.

2 Extensions to Conventional Page Tables for 64-bit Address Spaces

This section reviews commonly-used page tables—linear, forward-mapped, and hashed—and discusses extending them to support 64-bit (virtual) addresses. A detailed description can be found in Huck and Hays [Huck93]. For all page table designs, 64-bit address mapping information will require eight bytes (unless physical addresses are restricted to less than about 36 bits). Figure 1 illustrates example mapping information that contains one valid bit, a 28-bit PPN (40-bit physical address with 4KB pages), 12 bits of software and hardware attributes, and reserves PAD bits for future use. We use little-endian notation to number the bits, so the least significant bit is bit 0.

V	PAD	PPN	ATTR
63	40	12	0

Figure 1: Example PTE format for 64-bit address spaces

A *linear page table* conceptually stores all PTEs for a process in a single array. The array is indexed by the virtual page number(VPN), as depicted by Figure 2. Complete linear page tables are very large and are only partially populated. Consequently, they reside in virtual address space, using page faults to dynamically populate the table (*e.g.,* VAX-11 [Levy82], MIPS R4000 [Kane92]). Consequently, PTEs are allocated a page at a time and space overhead is high if an address space is used sparsely. A separate data structure stores mappings to the page table itself. Hardware or software searches this data structure on a nested TLB miss when attempting to access the linear page table with a virtual address. A multi-level tree of linear page tables is commonly used, *e.g.,* Ultrix uses a two-level tree and OSF/1 uses a 3-level tree on the MIPS R3000 [Nagl94]. The straightforward extension of linear page tables to 64-bit addresses uses a virtual array with four thousand trillion entries and a 6-level tree. This design is practical, as a portion of the TLB is reserved for mappings to the page tables [Nagl94] and the tree is rarely traversed. Alternatively, some other data structure can be used to store the mappings to the linear page table itself (*e.g.,* a hashed page table or a forward-mapped page table, described below).

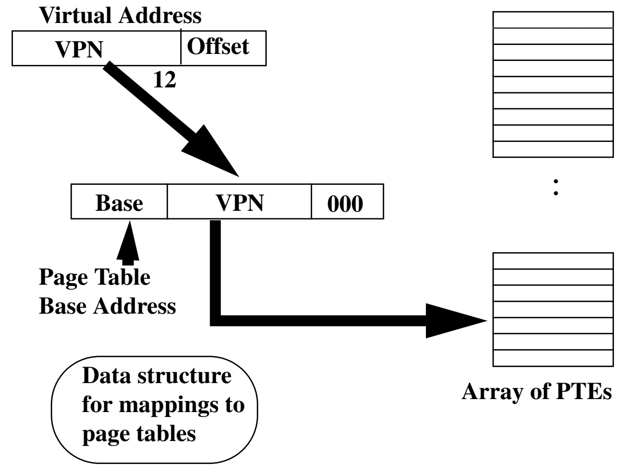

Figure 2: Linear Page Table (4KB page size)

Forward-mapped page tables store PTEs in n-ary trees, with each level of the tree indexed using fixed address fields in the VPN (Figure 3). The leaf nodes store PTEs while intermediate nodes store pointers (PTPs) to the next level (*e.g.,* SPARC Reference MMU [SPAR91]). Extending to a 64-bit address space extends the number of levels to seven. Forward-mapped page tables are impractical for 64-bit address spaces, because the overhead of seven memory accesses on every TLB miss is not acceptable. Techniques to short-circuit some levels, *e.g.,* guarded page tables [Lied95] or Region Lookaside Buffers [Chan95], are partially effective but still require many levels.

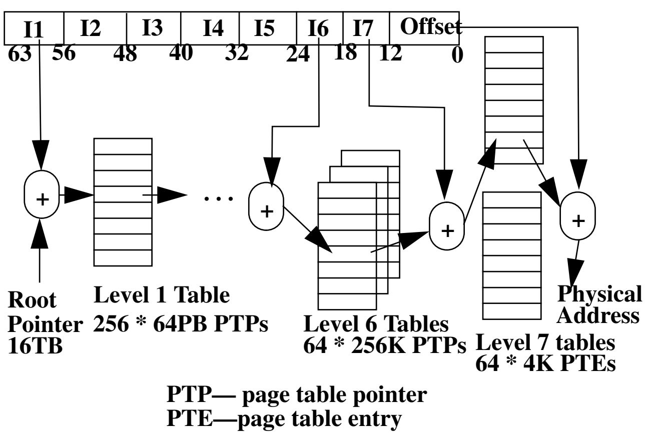

Figure 3: Forward-mapped page table

Large address space systems often use *hashed (inverted) page tables* [Lee89, Chan88, Rose85, Huck93]. The simplest implementation uses an open hash table with a hash function that maps a VPN to a bucket. Each PTE in the hash table, stores the mapping information for one page, a tag identifying the VPN, and a **next** pointer. The hash tables use chaining to handle overflows (Figure 4). During page table look-up, the hash function indexes into an array of hash nodes—the first elements of the hash buckets, traverses the hash bucket until a PTE is found with a tag matching the faulting address.

```
for (ptr = &hash_table[h(VPN)]; ptr != NULL; ptr =
  ptr->next)
      if (tag_match(ptr, faulting_tag))
            return(ptr->mapping);
pagefault();
```

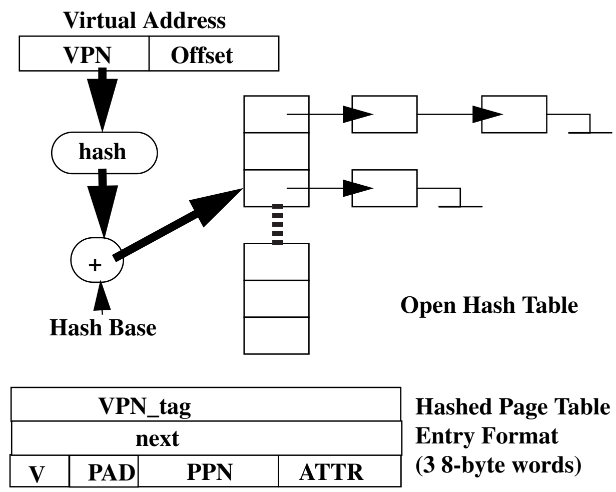

Figure 4: Hashed Page Table

Extending hashed page tables to 64-bit addresses is straightforward. A drawback is that the tag and **next** pointers are now eight bytes each, resulting in sixteen bytes of overhead for each eight bytes of mapping information.

Variations on hashed tables include inverted page tables and software TLBs. *Inverted page tables*, *e.g.,* in IBM System/38 [IBM78], hash to an array of pointers that when dereferenced obtain the first element of the hash bucket. *Software TLBs* (*e.g.,* swTLB[Huck93], TSB [Yung95], STLB[Bala94], PowerPC's page table [Silh93]) eliminate a hashed page table's **next** pointers by pre-allocating a fixed number of PTEs per bucket. They are so-named, because they can be viewed as memory-resident level-two TLBs with overflow handled in many ways [Agar88, Silh93, Thak86]. The innovations we develop for hashed page tables are applicable to inverted page tables or software TLBs also. Due to space constraints, in this paper we only describe hashed page tables. Inverted page tables and software TLB variations are described in [Tall95].

Multi-level linear page tables and forward-mapped page tables are both n-ary trees with some important differences. Multi-level linear page tables are accessed in a bottom-up fashion, require each intermediate node to be a page, and accessed with virtual addresses. Forward-mapped page tables are accessed in a top-down fashion, can have different branching factors at intermediate nodes, and can be accessed with physical addresses.

Which page table should 64-bit systems use? Linear page tables work well when most PTEs in each page of the page table are used, but perform poorly for sparse address spaces. Hashed page tables have fixed overhead—regardless of whether address space use is dense or sparse—but this overhead is 200% (sixteen bytes for eight bytes). What we would like is the low-overhead of linear page tables in the common case of dense address space use, while retaining the more graceful degradation of hashed page tables for sparse use. We next introduce clustered page tables to achieve this goal.

3 Clustered Page Table

The central contribution of this paper is the introduction of *clustered page tables.* They are a variant of hashed page tables that store mapping information for several consecutive pages (*e.g.*, sixteen) with a single tag and **next** pointer. Thus, for dense address space use, spatial overheads are much less than with hashed page tables. For sparse address space use, overheads are much less than with linear page tables, because few (*e.g.*, sixteen) not many (*e.g,* 512 = 4KB/8B) mappings need be allocated. In addition, clustered page tables perform ideally in cases where several consecutive pages are used together (*e.g.*, medium-sized objects and buffers). The section introduces clustered page tables for pages of a single page size (4KB), *base pages*. Section 5 extends them to work with superpage and subblock TLBs.

A clustered page table uses subblocking [Good83] to extend a hashed page table. Each node in the hash table stores one tag but stores mappings for multiple base pages that belong to the same page block—an aligned group of consecutive pages. The number of base pages in a page block is the *subblock factor*. Figure 5 shows the format of a clustered PTE with subblock factor of four and an open hash table constructed using them.

Subblocking for page tables is effective when programs store mappings to groups of contiguous virtual pages. Many programs map objects into their address space that are few to many pages long. These objects may be scattered anywhere in the address space. Thus, the address space of many programs is "bursty" and not arbitrarily sparse with mappings to isolated base pages. Clustered page tables exploit this property by storing mappings to a set of contiguous virtual pages in a single PTE and using a hash table to support a sparse distribution of these objects.

VPBN_tag			
next			
V **PAD**	**PPN0**	**ATTR0**	
V **PAD**	**PPN1**	**ATTR1**	
V **PAD**	**PPN2**	**ATTR2**	
V **PAD**	**PPN3**	**ATTR3**	

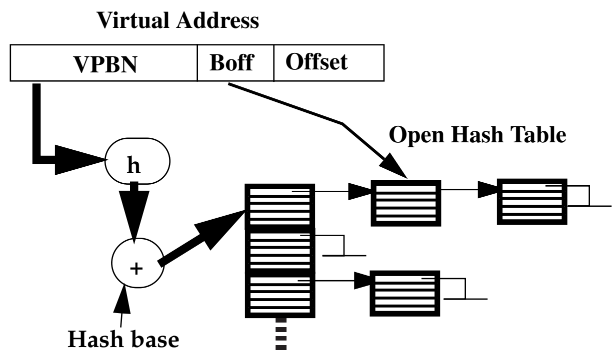

Figure 5: Format of Clustered PTE (subblock factor 4) and a Clustered Page Table

Many page table operations in a clustered page table are similar to those in a hashed page table. During page table lookup, for example, the virtual page number is split into a virtual page block number (VPBN) and a block offset (Boff). The VPBN participates in the hash function and the block offset indexes into the array of mappings in the PTE with a matching tag. TLB miss handling is identical to that for a hashed page table when traversing the hash list but differs *after* finding a PTE with matching tag:

```
for (ptr = &hash_table[h(VPBN)]; ptr != NULL; ptr =
  ptr->next)
    if (tag_match(ptr, faulting_tag))
          return(ptr->mapping[Boff]);
pagefault();
```

Figure 5 uses a subblock factor of four to simplify the illustration. Real implementations might use a larger subblock factor (*e.g.*, sixteen) determined by two issues. First, larger subblock factors reduce overhead when most entries are used, but increase overhead when mappings are sparse. Second, larger subblock factors pack mappings for consecutive pages close together, improving their spatial locality and potentially reducing cache misses while servicing TLB misses. If the size of the block of mappings is larger than a cache line, however, it may place the VPBN tag and mapping information in two different cache lines, potentially causing an additional cache miss on a TLB miss.

We next compare clustered and hashed page tables. Clustered page tables amortize the per-PTE overhead over many potential mappings. Page table size is smaller if enough mappings are used. For subblock factor sixteen, for example, a clustered page table uses the same memory as a hashed page table when six mappings are used, and about one-third the memory if all are used. Clustered page tables can require more memory than hashed page tables, however, if address space use is very sparse. The choice of the subblock factor can take into account the degree of sparseness. Further, to support address spaces with varying degree of sparseness, clustered page tables generalize to include PTEs with varying subblock factors with only a small increase in page table access time (a few extra instructions in the TLB miss handler) but with better memory utilization [Tall95].

Clustered page tables store mappings for multiple base pages in a single PTE. This reduces the number of PTEs in the page table and results in shorter hash table lists, a hash table with fewer buckets, or both. Shorter hash table lists reduce hash table search time on TLB misses [Knut68, Morr68]. Clustered page tables can have a worse page table access time if cache performance on TLB misses is worse when tag and mapping information reside in separate cache lines.

3.1 Clustered Page Table support for Page Table Manipulations

Page tables must support several operations. Much of this paper emphasizes supporting address translation on TLB misses. This section qualitatively discusses some advantages and disadvantages of clustered page tables relative to hashed page tables for other operations important to operating systems, including adding a mapping, changing mappings for a virtual address range, and synchronization in multi-threaded operating systems. A quantitative evaluation is not possible with our simulation methodology (Section 6.1), and must await a full-fledged operating system implementation.

Page tables must support adding a mapping. For hashed page tables, adding a mapping incurs a fixed overhead of memory allocation, list insertion and tag initialization for each PTE added to the page table. Clustered page tables amortize the overhead of allocating memory for a PTE and inserting in the hash list over multiple PTE insertions for the same page block. This is an important benefit as programs show spatial locality in their access patterns.

A second operation that operating systems often use is modifying the PTEs for a virtual address space range. In hashed page tables this requires searching the hash table once per base page. Clustered page tables require searching the hash table only once per page block and is more efficient.

Page tables must support multi-threaded operating systems with a synchronization protocol to coordinate concurrent page table operations. The synchronization protocol can significantly affect performance [Khal94]. Hashed and clustered page tables may associate a lock with each hash bucket. When executing a range operation, clustered page tables require the operating system to acquire a single lock for an entire page block instead of one per base page as in hashed page tables. While this allows for efficient range operations, it can restrict concurrent page table lookups on neighboring base virtual pages, *e.g.,* during TLB miss handling in a multiprocessor system. However, this is not critical as TLB miss handlers typically access page tables and update reference and modified bits *without* acquiring any locks. PowerPC, for example, defines a synchronization algorithm for page table updates that accounts for this unorthodox behavior of TLB miss handlers [May94]. Systems that require TLB miss handlers to acquire locks can use readers-writer locks that allow multiple TLB lookups in parallel.

We next review recent proposals for new TLB techniques that require page table support to be effective. We discuss extending page tables to support such TLBs. This will demonstrate additional advantages of clustered page tables.

4 Adapting Conventional Page Tables for Superpage and Subblock PTEs

This section and Section 5 present the second contribution of this paper: discussing the page table changes needed to make superpage and subblock TLBs (described below) effective. There are two *potential* advantages of adding such support. First, using the new TLBs reduces the number of TLB misses by 50% to 99% and provides an average execution time speedup of upto 20% for the workloads we use [Tall95]. Second, superpage and partial-subblock PTEs (described below) store mapping information more compactly than conventional PTEs, thus decreasing page table memory usage. We next review the new TLB techniques and then examine adapting conventional page tables to support superpages, partial-subblocking, and prefetching into complete-subblock TLBs.

4.1 Superpage and Subblock TLBs

A TLB is a cache whose entries hold recently-used PTEs [Mile90]. A conventional TLB entry has VPN (and process ID, etc.) for a tag and physical page number (PPN) (and protection information, page-modified bit, etc.) as data. With 64-bit addresses, hardware designers are increasing the effectiveness of TLBs with support for superpages [Tall92] and subblocking [Tall94].

Superpages use the same linear address space as conventional paging, have sizes that must be power-of-two multiples of the *base page size*, and must be aligned in both virtual and physical memory [Tall94]. Many processors now support superpages, *e.g.,* MIPS [Kane92], UltraSPARC [Yung95], Alpha [Site93], PowerPC [Silh93] etc. Supporting superpages is easier than supporting segments, which use a two-dimensional address space, may be arbitrarily long, and may start at arbitrary physical addresses [Orga72]. The MIPS R4000 [Kane92], for example, supports a 4KB base page size and superpages of 16KB, 64KB, 256KB, 1MB, 4MB, and 16MB. Large superpages, *e.g.,* 256KB and larger, are useful for kernel data, frame buffer, database buffer pools, etc. Since there are usually few large superpages in use, their mappings may be setup with limited changes to existing operating systems. Medium superpages, *e.g.,* 16KB or 64KB, require more substantial operating system changes to provide the mechanisms to support them and the policies for choosing appropriate page sizes [Tall94, Khal93, Rome95].

Subblocking associates multiple PPNs with each TLB tag [Tall94]. With a *subblock factor* of sixteen and 4KB pages, for example, each tag covers an aligned 64KB block of (virtual) addresses. The MIPS R4X00 [Kane92] processors support a subblock factor of two in the TLB. Subblocking is effective when spatial locality makes it likely that consecutive pages are in contemporaneous use. A disadvantage of subblocking is that the TLB data area is much larger than in conventional TLBs, because the data contains multiple PPNs. One way to reduce the area is to store a single PPN and require physical pages mapped by a single TLB entry be placed in a single, aligned block of physical memory, *i.e., properly placed*. Pages not properly placed use multiple TLB entries. We call this TLB design *partial-subblocking*[1], and use *complete-subblocking* to refer to the first subblock design [Tall94]. A partial-subblock TLB entry is like a superpage TLB entry but allows a subset of the base page mappings to be valid—specified by a valid bit vector (bottom of Figure 6). While a superpage can be used only when all base pages are valid and properly placed, a partial-subblock PTE can be used even when some base pages are not in memory, *e.g.,* when only fifteen of sixteen pages are memory resident.

Superpages and partial-subblocking are effective only when operating systems often properly place virtual pages in physical memory. We have proposed one algorithm, page reservation, that is described in [Tall94, Tall95]. Superpages and partial-subblocking also require support from the page table to store such mappings and in the TLB miss handler to traverse such page tables. Page tables must support finding a PTE on a TLB miss using the faulting address (without knowing the page size when starting the access) and without significantly increasing the TLB miss penalty. To the best of our knowledge, current commercial operating systems do not include either such memory allocation or page table support, rendering the hardware TLB extensions useless.

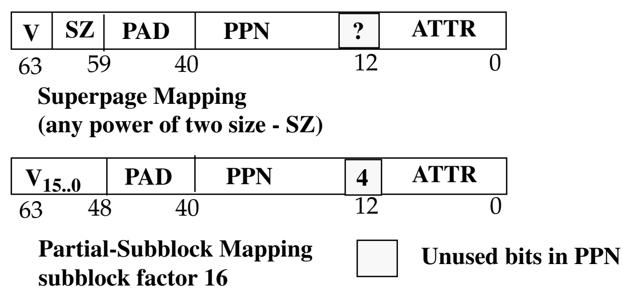

Figure 6: Superpage and Partial-subblock mapping format

A naive way of supporting superpage and subblock TLBs with a single-page-size page table would be to construct superpage and subblock PTEs in the TLB miss handler if PTEs are compatible to use the new TLB formats. This requires the TLB miss handler to search the page table for the PTEs for base virtual pages that are in the same virtual page block as the faulting virtual address. For hashed page tables this is very expensive as PTEs for neighboring base virtual pages will be in different hash buckets. Even in linear, forward-mapped, and clustered page tables where the PTEs are stored contiguously the compatibility check is expensive [Tall95]. A more efficient way would be to modify the page tables to store superpage and partial-subblock PTEs (Figure 6)

1. We only recently proposed partial-subblock TLBs and we are not aware of commercial processors that implement them today.

preconstructed by the operating system. In the next few subsections, we show some ways to modify conventional page tables. In Section 5 we show the additional advantages of clustered page tables for storing superpage and partial-subblock PTEs.

4.2 Supporting Superpages

Here we discuss adapting conventional page tables to support superpage PTEs (the top of Figure 6). There are, at least, two solutions for supporting superpages that work for any page table: *Replicate PTEs* and *Multiple Page Tables*.

Replicate PTEs. This solution stores a superpage PTE at the page table site of every base page PTE covered by the superpage. Thus, the information for a 64KB superpage gets repeated at sixteen base PTE sites. On a TLB miss, the handler finds the mapping as if the address was contained in a base page, but ends up loading a mapping for the whole superpage.

This solution is satisfactory. It is simple. It facilitates better TLB performance than for a conventional TLB by permitting superpage PTEs to reduce the frequency of TLB misses, while having *no* affect on the TLB miss penalty. It has two drawbacks. First, it does not allow the use of superpages to make page tables smaller. Second, replicated PTEs make adding or atomic update of superpage PTEs more complex in multi-threaded operating systems [Eykh92].

Multiple Page Tables. This solution creates a page table for each page size (or a set of page sizes) in use. On a TLB miss, the handler first accesses one of these page tables. If a mapping is not present, it examines the next page table and so on. The page tables probably should be sequenced from the page size most- to least-likely to cause a TLB miss, *e.g.*, the smallest to the largest page size.

This solution appears less good than the first solution. Its principal disadvantage is that it will make TLB miss handling slower, unless most TLB misses go to one page size. Furthermore, the spatial overhead of supporting many page tables mitigates its potential to improve page table size. With linear page tables, PTEs for different page sizes cannot share page table pages. With hashed page tables, hash buckets must be set up for each page size.

A simple variation of the multiple page table approach uses the same page table to store PTEs of different page sizes and examines it multiple times with different page sizes. This is feasible with hashed and clustered page tables, where all logical page tables could share the same buckets at a cost of longer hash chains. All PTEs in a linear page table, by definition, are of the same page size and multiple page tables cannot be combined.

There are also some superpage strategies that only work for specific page tables.

Linear Intermediate Nodes. Linear page tables that use a multi-level tree structure can store superpage PTEs at intermediate tree nodes. With 4KB base pages and eight byte PTEs, for example, each entry in the last level of intermediate nodes points to a page of 512 PTEs. This solution allows the intermediate node entry to point to a superpage covering the same virtual space (2MB = 512*4KB).

This solution supports superpages with a modest increase in TLB miss handling time (to decide whether an intermediate node is a superpage PTE or points to the next level). The TLB miss handler would, however, still access the base PTE site and incur a nested TLB miss. Its key disadvantage is the lack of flexibility. It only supports page sizes that correspond to intermediate nodes—in our example these are 2MB, 1GB, 512GB, 256TB and 64PB. In particular, it supports no medium-size superpages.

Forward-Mapped Intermediate Nodes. Forward-mapped page tables are a multi-level tree structure and also can store superpage PTEs at intermediate tree nodes, *e.g.*, SPARC Reference MMU [SPAR91] and HaL [Chan95] support a few fixed superpage sizes in this fashion. It is possible to extend forward-mapped page tables to support any arbitrary superpage size by varying the tree's branching factor dynamically. A software-traversed forward-mapped page table is flexible and can support this (unlike linear page tables where the number of PTEs per page fixes the branching factor or hardware traversed page tables that have fixed branching factors). This solution may increase the levels in the tree and will likely make the already too-long TLB miss handling time of forward-mapped page tables even longer.

Superpage-Index Hashed. One way to support superpages in a conventional hashed page table is to always assume a specific superpage size in the hash function and to associate with a bucket all appropriate superpage and base page PTEs. Talluri *et al.* [Tall92] describe a similar scheme for hardware TLBs, where set-associative TLBs support two page sizes using the superpage index. If we hash on 64KB superpages, for example, we could find that a particular 64KB region mapped by (a) one 64KB superpage, (b) sixteen 4KB base pages, (c) two 16KB superpages and eight base pages, etc. This would result in one, sixteen, or ten PTEs chained to the same bucket (besides any other PTEs mapping to the same bucket). This solution is not so good, because the longer hash chains will increase TLB miss handling time. In addition, superpages larger than the size selected for hashing must be handled another way.

In summary, the replicate PTE method is probably the best method so far for supporting medium-sized superpages in conventional page tables—it decreases frequency of TLB misses *without* increasing the TLB miss penalty. Large superpages (*e.g.*, 1MB), on the other hand, may be handled on an *ad hoc* basis, since there are few such mappings and they miss less often in a TLB.

4.3 Supporting Partial-subblocking

This subsection applies superpage page tables to supporting partial-subblock PTEs (bottom of Figure 6). The advantages of supporting partial-subblock PTEs over superpage PTEs are four-fold. First, partial-subblock TLBs are more effective than superpage TLBs [Tall95]. Second, partial-subblock PTEs reduce page table size more effectively than superpages (Figure 10). Third, partial-subblocking requires simpler operating system support than superpages [Tall94]. Fourth, a partial-subblock PTE is a natural intermediate format for page tables that construct superpage PTEs in an incremental fashion. The disadvantage is that large subblock factors, *e.g.*, 32 or larger, are not practical due to the limited number of valid bits in a PTE.

Page table support for partial-subblock PTEs is similar to supporting a base page size and one medium superpage size equal to the base page size times the subblock factor. A partial-subblock PTE resides in a page table exactly where a corresponding superpage PTE would have resided. Page blocks that cannot use partial-subblock PTEs use base page PTEs. More complex optimizations are possible [Tall95].

The extensions described in Section 4.2 for superpage PTEs are mostly applicable to storing partial-subblock PTEs also. The differences include: When using replicated PTEs, adding or deleting a mapping that is part of a partial-subblock PTE always requires modification of multiple PTEs, whereas superpage PTEs tend to use explicit operating system directed page promotion or demotion. When using multiple page tables, the order of searching the page tables should favor the partial-subblock PTEs over the base page table if partial-subblock PTEs will be accessed more often than base page PTEs. In superpage-index hashed page tables, partial-subblock PTEs reduce the length of the hash lists. When superpages could not be used, multiple base page PTEs are added to the same hash bucket—typically, one or two partial-subblock PTEs can replace the base page PTEs.

4.4 Prefetch support for complete-subblock TLBs

Another hardware technique for increasing the memory mapped by a TLB is complete-subblocking. A complete-subblock TLB entry has one tag but has a subblock-factor-number of PPNs and attribute fields, similar to a clustered PTE. A complete-subblock TLB requires no special operating system or page table support. On a TLB miss, the handler merely searches any page table for the base page PTE and loads it into the TLB—exactly as in a single-page-size system.

A closer look at complete-subblock TLBs reveals, however, that there are *block* misses and *subblock* misses. Block misses allocate a new TLB entry, often replacing an old entry (and its associated mappings). Subblock misses add a new PPN and attribute information to an existing TLB entry, without causing a replacement. Subblock misses can be eliminated, however, if each block miss loads (prefetches) all mappings associated with its tag, as the MIPS R4000 does for two PTEs [Kane92]. For example, on a TLB miss to virtual address 0x41034 the TLB miss handler for MIPS R4000 would prefetch mappings to virtual pages 0x40000 and 0x41000. Subblock prefetching never pollutes the TLB by replacing more useful mappings, because it never causes extra replacements [Hill87], but reduces the number of TLB misses significantly—50% or more [Tall95].

A drawback of subblock prefetching is the increased time to service TLB block misses. This penalty is large for hashed page tables, because multiple hash probes are needed. This penalty is reasonable for linear, forward-mapped, and clustered page tables, because the additional mappings reside in adjacent page table memory. In addition, the penalty can be reduced even further by modifying the clustered PTE format to match the format of the corresponding complete-subblock TLB entry exactly.

5 Partial-subblock and Superpage PTEs in Clustered Page Tables

The section first examines incorporating partial-subblocking into clustered page tables. This step is natural, since a node in a clustered page table (for base pages only) resembles a complete-subblock TLB entry. This section then incorporates superpages into clustered page tables.

VPBN_tag				
next				
V	PAD	S	PPN0	ATTR0
V	PAD	S	PPN1	ATTR1
V	PAD	S	PPN2	ATTR2
V	PAD	S	PPN3	ATTR3

Clustered PTE (or Complete-subblock PTE)

VPBN_tag				
next				
V3..0	PAD	S	PPN	ATTR

Partial-subblock PTE

VPBN_tag					
next					
V	SZ	PAD	S	PPN	ATTR

Superpage PTE

Figure 7: Clustered PTE variations

The match between partial-subblock TLBs and clustered page tables is best when both use the same subblock factor. Figure 7 (top) illustrates a clustered PTE with subblock factor four, and, therefore, has an array of four base page mappings. A clustered page table is essentially a "complete-subblock hashed page table" and a clustered PTE can also be called a complete-subblock PTE. Figure 7 (center) illustrates a partial-subblock PTE. Figure 8 shows how a partial-subblock PTE fits in a clustered page table. On a TLB miss, the handler hashes on the VPBN and walks the hash chain as usual. If a match is found, the handler consults the new S field and then reads the appropriate mapping. The S field—for *S*ubblock/*S*uperpage—distinguishes a partial-subblock (and superpage, discussed next) PTE from base page PTEs, since both reside in the same page table. The key here is that the TLB miss handler sees no difference from a regular clustered page table while traversing the hash list matching tags and only differs when reading the mapping. Thus we are able to service TLB misses to both partial-subblock and base page PTEs without increasing the TLB miss penalty while using less memory for partial-subblock PTEs.

```
for (ptr = &hash_table[h(VPBN)]; ptr != NULL; ptr =
   ptr->next)
        if (tag_match(ptr, faulting_tag))
                return(ptr->mapping[0].S ? ptr-
   >mapping[0] : ptr->mapping[Boff]);
pagefault();
```

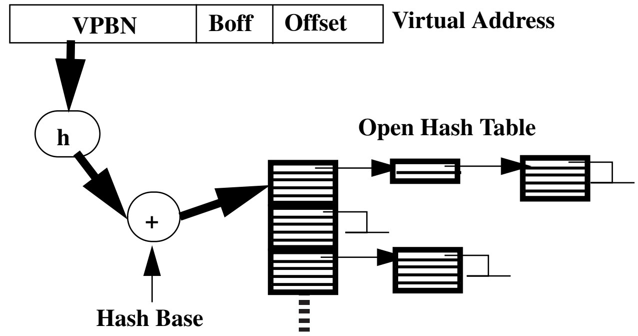

Figure 8: Storing partial-subblock and superpage PTEs in clustered page tables

Superpage support is also straightforward. Figure 7 (bottom) illustrates support for a medium-sized superpage, whose size is the same as the virtual page block. The superpage PTE is similar to a partial-subblock PTE, except it only has one valid bit. Smaller superpages can be supported using the SZ field to identify them. The above example could allow a node with two 8KB superpages. We also can support one 8KB superpage and two 4KB base pages in one 16KB page block by using two nodes on the same hash chain—one with one of two 8KB superpages valid and the other with two of four base pages valid. This support also requires that the TLB miss handler continue searching the hash chain after a tag match that fails to find a valid mapping [Tall95].

Larger superpages can be supported in at least two ways. First, one can use the "Replicate PTEs" solution, but replicate once per clustered PTE instead of once per base page PTE. For subblock factor sixteen, for example, a clustered page table supports large superpages with a factor of sixteen less overhead than conventional page tables. Second, one can continue to use any *ad hoc* method previously developed for conventional page tables—in particular, the multiple page table approach is reasonable if TLB misses to large superpages are not frequent.

Supporting superpages and partial-subblocks in clustered page tables offers several advantages over extending hashed page tables. First, its hash chain remains short, whereas hashed page tables require longer hash chains when using base pages. Second, partial-subblock and superpage PTEs reduce both hashed and clustered page table size but clustered page tables do not increase TLB miss penalty whereas hashed page tables do. In adding superpage or partial-subblock PTEs, clustered page tables do not change the hash table structure but only modify the size of some PTEs. Hashed page tables, on the other hand, either require multiple page tables/probes or have suboptimal hash functions when storing superpage and partial-subblock PTEs. Third, clustered page tables simplify incremental creation of partial-subblock and superpage PTEs by storing mappings for consecutive base pages together. If the operating system, notices that all base page mappings in a node are valid, it could decide to promote them to a superpage. Gathering this information in other page tables is less efficient.

In summary, clustered page tables for base pages use less memory than hashed page tables by combining mappings for neighboring base virtual pages that have nearly identical *tags* into a single PTE with a single tag. In this section, we took clustered page table one step further to support superpages and partial-subblocking by combining mappings for neighboring base virtual pages that also have nearly identical *PPNs* into a single PTE. It is also straightforward to support both superpages and partial-subblocking in the same clustered page table [Tall95].

6 Performance Evaluation

This section presents simulated performance results for the page tables discussed so far. We use estimates of page table size and page table access time (the TLB miss penalty) as metrics. Our numbers are approximate, for example, because we do not compute the cache misses saved by a smaller page table. Nevertheless, we find that clustered page tables use less space and can be accessed faster than conventional page tables. Further, their performance relative to conventional page tables improves further when supporting superpages or partial-subblocking.

6.1 Simulation Methodology and Metrics

The ultimate measure of performance is program execution time. We are, however, unable to measure execution time on a real system for two reasons. First, it requires implementing all the different page table organizations in a commercial operating system. Implementing even a single page table organization is a multi-man-year project and is beyond the scope of this paper. Second, we are evaluating TLBs and page tables that support medium-sized superpages, complete- and partial-subblock TLBs. We did not have access to a SPARC machine that supports medium-sized superpages and no current processor includes the subblock-TLBs we study here. We instead include a TLB simulator in the kernel to simulate these TLBs and count the number of TLB misses.

We perform our study on a SparcServer10 running Solaris 2.1. We modified the operating system in three ways. First, we implemented policies and mechanisms in the operating system to support medium-sized superpages and partial-subblock TLBs. We use a dynamic page-size assignment policy that chooses between a base page size of 4KB and a superpage size of 64KB (detailed description is available in [Tall94] and [Tall95]). We also use a physical memory allocation algorithm, page reservation, that allocates aligned physical pages for virtual pages (detailed description is available in [Tall94] and [Tall95]). This memory allocation makes partial-subblock TLBs effective and makes superpage creation efficient. Second, we added a mechanism to trap on TLB misses (trap-driven simulation). We use this mechanism to drive a TLB simulator that counts the number of TLB misses for different target TLBs—TLB configurations that are different from the hardware TLB. Third, we include a page table simulator that builds hashed and clustered page tables. It counts the number of cache lines that would have been accessed if the hardware TLB was identical to the target TLB and the simulated page table was fully implemented in the kernel. Note that we do not replace Solaris' native page table implementation with the new page tables.

Our base cases assume 64-entry fully-associative TLBs, 4KB base pages, and—as appropriate—subblock factor sixteen and superpage size 64KB. We assume a 256-byte (level-two) cache line size for accessing page tables. We assume 4096 hash buckets in hashed and clustered page tables. We also discuss sensitivity analysis for these assumptions, but space limitations prevent a full presentation [Tall95].

We first study page table size. Page table size has significant affect on cache behavior, even when page tables are much smaller than available physical memory (*i.e.,* when their affect on page faults is negligible). Smaller page tables are expected to result in a higher cache hit rate and lower cache pollution that can significantly improve overall system performance. Yoo and Rogers [Yoo93], for example, observed a 10% improvement in execution time mostly due to cache/TLB effects of reducing page table size for a commercial database workload. Further, when using a private address space model and per-process page tables, a smaller page table size for each process on a large server system with thousands of active processes translates to significant savings.

Our metric for page table size is page table size normalized by hashed page table size. We estimate page table size in a two step process. First, we take a snapshot of each workload's mappings at a point near the program's maximum memory use. Second, we use this information to generate alternate page tables using the following additional assumptions. Mapping information takes eight bytes. Linear page tables use the minimum possible six-level tree. We also show "1-level" numbers that assume intermediate nodes are stored in a data structure that takes zero space. Forward-mapped page tables use a seven-level tree as in Figure 3. Hashed and clustered page tables have an overhead of sixteen bytes per PTE to store tag and next pointers. We compute page table size for a multiprogrammed workload as the sum of page table sizes for the constituent programs.

We next study page table access time. Regrettably, our simulation environment does not allow us to measure it directly. Instead we use the average number of cache lines accessed to handle one TLB miss as an indirect metric. This metric would be proportional to page table access time if the (level two) cache rarely contains page table data and other overheads are minimal. There are at least three drawbacks to this metric. First, and most important, it ignores that some page table data may still be in cache, particularly for page tables that are smaller and store PTEs to exploit spatial locality. Thus, we would expect the access times for clustered page tables, which use less page table memory, to be better than the results we report. Second, the metric ignores the initial overhead of a TLB miss, but this penalty is independent of page table type. Third, it neglects the time to execute TLB miss handler instructions. This allows the metric to account for hardware TLB miss handlers that typically take time proportional to the number of memory accesses. Even with software TLB miss handling, the instruction overhead for hand-coded TLB miss handlers is expected to be small on next generation superscalar processors that can execute three, four, or more instructions per cycle, compared to a main memory access of about a hundred cycles.

We estimate the average number of cache lines accessed on a TLB miss as follows. We modified the operating system to simulate hashed and clustered page tables. On each TLB miss, our evaluation system traps to the operating system, providing us the faulting address. We do a page table traversal to calculate the number of cache lines accessed. We estimate the cache lines by further assuming each PTE starts on a cache line boundary. Linear page tables always access one cache line and occasionally access higher tree levels. We approximate this by reserving eight of 64 TLB entries for higher tree level mappings and assuming each TLB miss to the remaining 56 entries only accesses one cache line. For our 32-bit workloads, the eight reserved TLB entries are sufficient and we never take a nested trap[2]. We assume forward-mapped page tables access one cache line for each tree level. When storing superpage and partial-subblock PTEs, we assume that linear and forward-mapped page tables use the replicate PTE approach and hashed page tables use separate page tables for 4KB and 64KB page blocks, with the 4KB page table searched first. We normalize the number of cache lines accessed by the number of TLB misses incurred by a 64-entry TLB, which is independent of the page table type. For linear page tables we estimate the number of cache lines accessed when using 56 TLB entries and normalize with the number of TLB misses for a 64-entry TLB. Our metric thus includes the opportunity cost for reserved TLB entries.

6.2 Workloads

We selected ten 32-bit workloads that spend significant time in TLB miss handling. **Nasa7, compress, wave5, spice,** and **gcc** are from the SPEC92 suite [SPEC91]; **fftpde** is a NAS benchmark [Bail91] operating on a 64X64X64 matrix; **mp3d** and **pthor** are uniprocessor versions from the SPLASH benchmark suite [Sing92]; **coral** [Rama93] is a deductive database executing a nested loop join; **ML** [Appe91] is executing a stress test on the garbage collector [Repp94]. **Compress** and **gcc** are multiprogrammed workloads. Many programs have negligible TLB miss ratios and would not benefit from page table enhancements. By emphasizing workloads for which TLB miss handling is important our results overestimate the potential benefit for workloads with small processes. We expect future 64-bit workloads and object-oriented programs to have larger and sparser address spaces. Such workloads would make TLB and page table effects more important and both hashed and clustered page tables more attractive.

Workload	total time (user time) in seconds	#user TLB misses (000s)	% user time in TLB miss handling	Memory for Hashed page table
coral	177 (172)	85974	50%	119KB
nasa7	387 (385)	152357	40%	21KB
compress	104 (82)	21347	26%	8KB
fftpde	55 (53)	11280	21%	88KB
wave5	110 (107)	14511	14%	86KB
mp3d	36 (36)	4050	11%	29KB
spice	620 (617)	41922	7%	22KB
pthor	48 (35)	2580	7%	92KB
ML	950 (919)	38423	4%	194KB
gcc	159 (133)	2440	2%	34KB
kernel space	N/A	N/A	N/A	186KB

Table 1: Workload characteristics

2. We save the reserved TLB entries across context switches.

Table 1 displays workload data, with the workloads sorted from most to least percent of user time spent on TLB miss handling. Column two gives total execution time, with user time in parenthesis showing that these workloads spend most of their time in user mode. Columns three and four give the number of user TLB misses (for a 64-entry fully-associative single-page-size TLB) and the percent of user time spent servicing these misses (assuming a 40 cycle TLB miss penalty), showing that user TLB miss handling time is significant. Finally, column five shows the amount of memory used by a hashed page table to map the workload.

6.3 Results

We first discuss page table size. Figure 9 displays relative page table sizes—normalized to hashed page table size—for various workloads in a single-page-size system with base page size of 4KB. Figure 9 truncates values above 5.0. The important observation is that clustered page tables (asterisk marked) use less memory than the best conventional page tables for all the workloads. For dense address spaces, *e.g.,* **coral, ML, kernel**, clustered page tables are comparable or better than linear and forward-mapped page tables. For sparse address spaces, *e.g.,* **gcc** and **compress**[3], clustered page tables use less memory than hashed page tables.

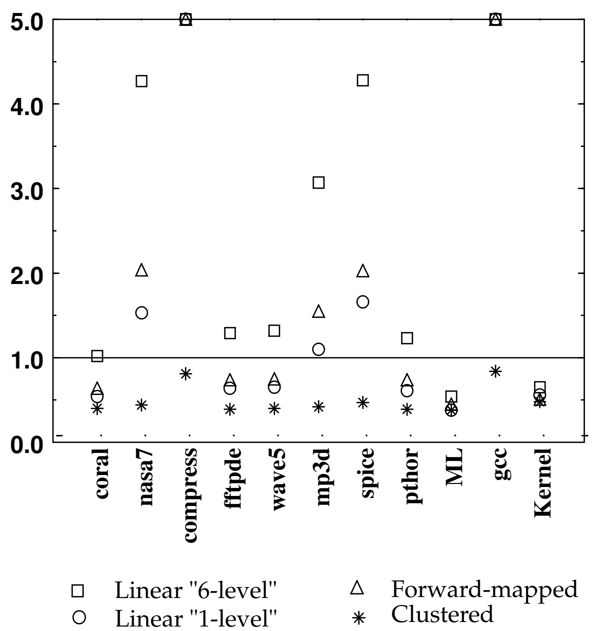

Figure 9: Page table size for single-page-size page tables. Normalized to hashed page table size.

3. **Gcc** and **compress** had multiple processes, *e.g.,* **make**, **sh**, **script**, many of which were small and had sparse address spaces. For other workloads we measure page table usage of only the main program.

Figure 10 zeroes in on page tables that use less memory than hashed page tables, *i.e.,* normalized page table size less than 1.0. It also adds variations that use superpages and partial-subblocking to store mappings to multiple base pages in a single PTE. Use of superpage PTEs in the clustered page tables reduces memory usage upto 75% (comparing circles and asterisks in Figure 10) and with partial-subblock PTEs by upto 80% (comparing triangles and asterisks in Figure 10). Use of superpage mappings similarly improves hashed page table size also (squares in Figure 10). Corresponding improvements are not possible in linear or forward-mapped page tables as we assume replication of PTEs.

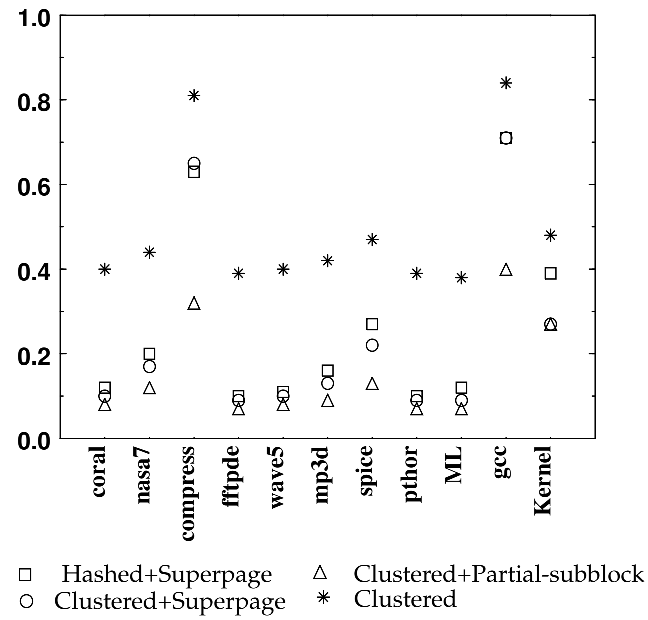

Figure 10: Hashed and clustered page table sizes for 4KB base pages and 64KB superpages/ partial-subblocking with subblock factor 16. Normalized to hashed page table size.

We next discuss page table access time (on TLB misses) for various workloads and page tables using the coarse metric: *average number of cache lines accessed on a TLB miss.* Each graph in Figure 11 assumes a different fully-associative 64-entry TLB design.

Figure 11a assumes a single-page-size TLB, *i.e.,* no TLB support for superpages or subblocks. Results show that forward-mapped page tables perform unacceptably but other designs are similar. This is not surprising since our metric does not reward the more-compact clustered page tables. Clustered page tables have shorter hash lists relative to hashed page tables reducing the number of accesses in some cases, *e.g.,* **ML**. Results for linear page tables are optimistic due to assumptions discussed in Section 6.1.

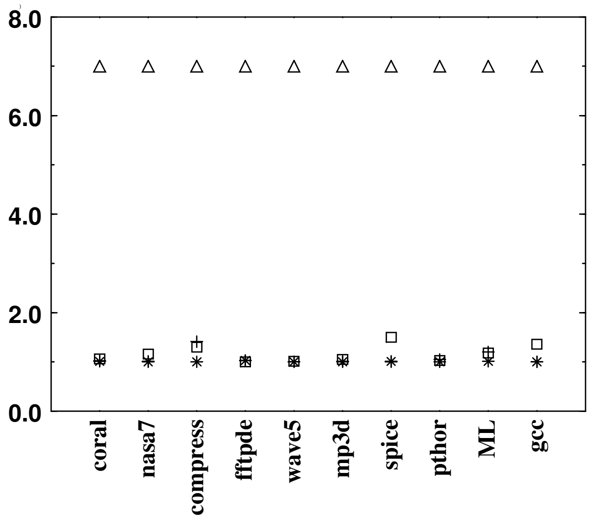

Figure 11a: Single-page-size TLB

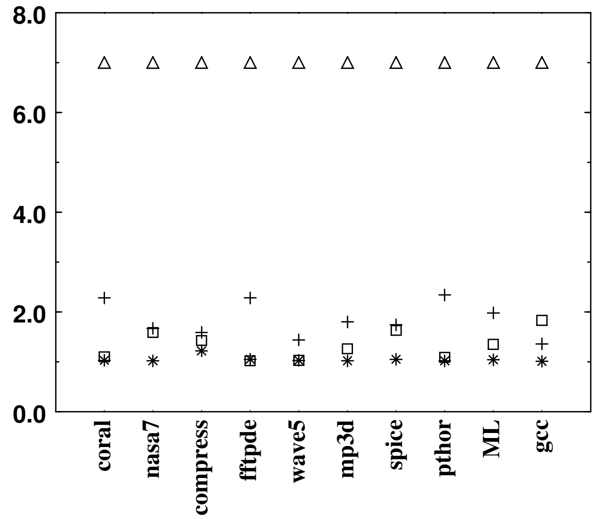

Figure 11b: 4KB/64KB Superpage TLB

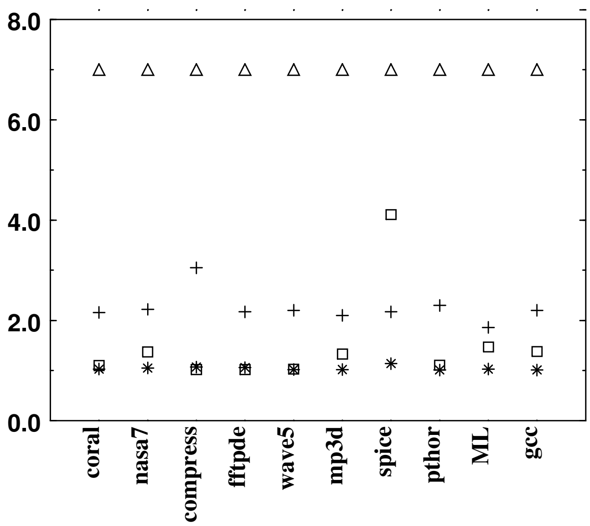

Figure 11c: Partial-subblock TLB (subblock factor 16)

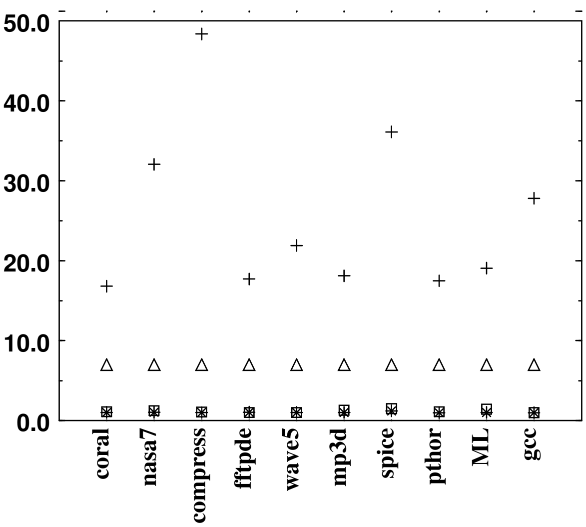

Figure 11d: Complete-subblock TLB (subblock factor 16)

□ Linear △ Forward-mapped + Hashed ⁎ Clustered

Figure 11: Average number of cache lines accessed per TLB miss (64-entry fully-associative TLBs)

Figure 11b present results when the TLB and page tables support superpages. Not shown is that use of superpages reduces TLB miss frequency by 50% to 99% [Tall95], which is the main reason for supporting superpage PTEs in the page table. What is shown is the number of cache lines accessed by the remaining misses. Results are modestly worse for linear page tables as the opportunity cost of fewer TLB entries is higher, unchanged for forward-mapped, and much worse for hashed page tables. Hashed page tables take longer to access superpage PTEs as we first search the 4KB page table and then the 64KB page table, *e.g.*, poor performance of hashed page tables for **coral** is due to a higher fraction of TLB misses to superpage PTEs than for **gcc**. Results for clustered page tables continue to be close to 1.0, showing that they handle the remaining TLB misses without increasing TLB miss penalty.

Figure 11c presents results for a partial-subblock TLB. To the first order, they are similar to results using a superpage TLB. As these workloads use partial-subblock entries more often than superpages, the hashed page tables have worse performance. For these workloads and with proper operating system support, doing the page traversals in the reverse order—the 64KB page table followed by the 4KB page table—would be a better option.

Finally, Figure 11d gives complete-subblock TLB results, assuming the prefetching described in Section 4.4. As expected, hashed page tables perform terribly due to the high cost of multiple probes (sixteen). Note that Figure 11d uses a different scale from the other graphs. Linear and clustered page tables continue to be close to 1.0 as they place the mappings for consecutive base pages nearby.

In summary, clustered page tables improve significantly on hashed page tables by supporting superpage and subblock TLB architectures without increasing the TLB miss penalty. Clustered page tables are, however, sensitive to the cache line size. A superpage or partial-subblock clustered PTE occupies 24 bytes but a clustered PTE occupies 144 bytes (subblock factor sixteen) and can span multiple cache lines. This would increase the average number of cache lines accessed when using clustered PTEs — *e.g.*, by 0.125 for 128 byte cache lines or 0.625 for 64 byte cache lines. However, the good news is that using superpage or partial-subblock PTEs in a clustered page table, irrespective of the hardware TLB, eliminates most of this penalty [Tall95]. Another solution is to use a smaller subblock factor, *e.g.*, 4 or 8, which makes the space/time tradeoff of increasing memory usage to reduce TLB miss penalty.

7 Discussion

With the increasing popularity of software TLB miss handling, page table design is now completely an operating system issue. The transition to 64-bit address spaces and need for supporting superpages and subblocking gives us reasons to reevaluate traditional hardware page table designs.

Linear page tables are generally an acceptable solution with fast and simple TLB miss handling. However, hashed page tables are better suited for sparse address spaces, as 64-bit address spaces are expected to be. Clustered page tables improve the memory overhead of hashed page tables while efficiently supporting medium-size superpages and subblocking. We emphasize again that it is important to support such TLBs as they reduce the number of TLB misses dramatically for large working set sizes.

An important limitation of our workloads is that we do not stress the TLB with multiprogrammed workloads[4]. Multiprogramming can increase the number of TLB misses and make TLB miss handling more significant [Agar88]. Multiprogramming can also affect physical memory allocation in superpage and partial-subblock systems. When physical memory demand is high, the operating system may not be able to use superpages or partial-subblocking as effectively as our simulations show.

Multi-level linear page tables do not scale to 64-bit address spaces due to a high overhead in the upper levels of the six-level tree that are sparse (the "6-level" numbers in Figure 9). Linear page tables are, however, still attractive as they have low overhead for dense address spaces and fast page table access if the mappings hit in the reserved TLB entries. The "1-level" numbers in Figure 9 and the average number of cache lines accessed estimates in Section 6.3 assume this optimistic assumption. In practice, it is possible to efficiently store the data structure for the mappings to the linear page tables in a hashed page table. This would result in a performance only slightly worse that reported in Section 6.3.

Many processors support more than one superpage size, *e.g.*, the MIPS R4000 processor supports pages sizes of 4KB, 16KB, 64KB, 256KB, 1MB, 4MB, and 16MB [Kane92]. While our quantitative measurements include the effect of only two page sizes—4KB and 64KB—clustered page tables support multiple page sizes more effectively then other page tables. Superpage PTEs for page sizes smaller than the page block size (64KB) coreside in a clustered page table without replication and are accessed without any increase in TLB miss penalty (Section 5). Superpage PTEs for page sizes larger than the page block size involve a space/time tradeoff as in conventional page tables but are more efficient. With the replicate PTE approach, clustered page tables are better because the large superpage size PTEs are replicated into fewer 64KB superpage PTEs instead of sixteen times as many 4KB PTEs. With the multiple page table approach, clustered page tables require fewer tables. Two clustered page tables suffice for all page sizes between 4KB and 1MB, for example, one clustered page table stores mappings for page sizes from 4KB to 64KB and another for larger page sizes upto 1MB. Conventional page tables may require as many page tables as the number of page sizes supported, *e.g.*, five in the MIPS R4000.

The performance of hashed and clustered page tables can be improved further in two ways. First, the load factor of the hash table can be reduced by increasing the number of hash buckets. Reducing the load factor reduces the average number of hash nodes searched during a traversal but slightly increases the amount of memory used if some buckets are empty. Second, constructing hashed or clustered page tables as a software TLB can reduce the number of cache lines accessed. We describe how clustered page table techniques can be applied to software TLBs in [Tall95]. A disadvantage of hashed and clustered page tables is the unpredictability of the hash table distribution that depends on the state of the current set of active processes. One solution is to use a per-process or per-process group page table instead of a single shared page table.

4. Except **compress** with runs two processes in parallel. **Gcc** is multiprogrammed also but runs multiple processes sequentially.

In Section 2, we described set-associative software TLBs as an alternate native page table structure, *e.g.,* page tables for the PowerPC [Silh93] are of this type. Software TLBs, which can have thousands of entries, are also effective as a cache of recently used translations and may reside between the TLB and a native page table. They reduce the TLB miss penalty to a single memory access on a hit but increase the TLB miss penalty on a miss. The use of software TLBs reduces the frequency of page table accesses and the importance of page table access time in determining overall performance. A software TLB allows the choice of a larger subblock factor in clustered page tables than the cache line size dictates or makes it practical to use a slower forward-mapped page table [Huck93]. The flexibility and efficiency of implementing operating system page table algorithms (Section 3.1) then becomes the overriding factor in choosing a page table structure.

A typical multiprogramming operating system, *e.g.,* UNIX [Thom74], maintains one page table per process or associates a process id with each PTE in a shared page table. The page table techniques described in this paper are equally applicable to single address space systems, *e.g.,* Opal [Chas94] or MONADS [Rose85], and segmented systems that use global effective virtual addresses, *e.g,* HP [Lee89] and a single shared page table. Hashed and clustered page tables are especially suited to single address space and segmented systems as they tend to have a very sparse but "bursty" address space.

Any set of experiments is finite and many variations and hybrid page table implementations are possible that we do not study or discuss in detail in this paper. For example, hashed page tables can be optimized by packing both the tag and **next** pointer into eight bytes by using a shorter **next** pointer and not storing tag bits that can be inferred from indexing the table [Huck93]. This reduces hashed page table size by 33%. This, however, does not change our results as clustered page tables are more effective with page table size reductions of 50% or more over the unoptimized hashed page table (Figure 10). The average number of cache lines per TLB miss accessed by the optimized version remains unchanged. A shorter **next** pointer also restricts page table placement in memory and operating system flexibility.

8 Conclusion

As the computer industry makes the transition from 32-bit to 64-bit systems, TLBs and page tables are affected. While linear and hashed page tables are still practical, forward-mapped page tables are not, because accessing them is too slow. Linear page tables have significant memory overhead and TLB pollution for sparse address spaces. Hashed page tables seem to be the logical choice for sparse 64-bit address spaces, but have a large per-PTE memory overhead. This paper makes two key contributions in the area of page table design.

The central contribution of this paper is a new page table organization, the *clustered page table*, which augments hashed page tables with subblocking to address their disadvantages. Specifically, clustered page tables are hashed page tables that store mapping information for several consecutive pages (*e.g.*, sixteen) with a single tag and **next** pointer. Clustered page tables use less memory than other page table organizations, are often faster to access during TLB miss handling and are flexible to support custom operating system needs.

The second contribution of this paper is a study of how to store superpage and partial-subblock PTEs in different page tables. Hardware architects are using (or considering) superpages and subblocking in TLBs to increase the memory size that can be mapped by a TLB entry. These TLB enhancements are largely useless if the page tables and the operating system does not support them with proper memory allocation and TLB miss handling. We showed that there exists a straightforward way to store such mappings in a page table—replicate the mappings—that uses the new TLB architectures to reduce the number of TLB misses and does not increase the TLB miss penalty. We also show that clustered page tables support medium superpage and partial-subblock TLBs without increasing the TLB miss penalty and—at the same time—reduce page table size.

It remains to be seen if commercial operating systems will incorporate the memory allocation and page-size assignment support needed for these new TLBs. Nevertheless, we suggest the use of superpage and partial-subblock PTEs in a page table even if the TLB does not require such support. The advantage being that using these mappings can result in smaller page tables that are faster to access. Clustered page tables provide natural support to store such PTEs and get the memory savings without increasing TLB miss penalty.

Finally, clustered page tables are the native page tables in Solaris 2.5, a commercial operating system, on UltraSPARC-based computers.

9 Acknowledgments

We thank Dock Williams and Vikram Joshi who are co-inventors on a patent covering clustered page tables [Tall93]; A. Caceras and V. Joshi who designed and implemented clustered page tables in Solaris 2.5; J. Johnson, G. Limes, S. Chessin, S. Kong, R. Yung, and others at Sun Microsystems Inc. for their role in design of the page table and TLB for UltraSPARC; P. Seshadri and J. Reppy for providing us with coral and ML workloads; J. Larus, M. Callaghan, S. Chandrashekaran, and the reviewers for their comments.

Bibliography

[Agar88] A. Agarwal, M. Horowitz, J. Hennessy. Cache Performance of Operating Systems and Multiprogramming Workloads. *ACM Trans. on Computer Systems*, 6(4):393–431, November 1988.

[Appe91] Andrew W. Appel and David B. McQueen. Standard ML of New Jersey. In *Proc. Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13, August 1991.

[Bail91] David Bailey, John Barton, Thomas Lasinski, Horst Simon. The NAS Parallel Benchmarks. *Intl. Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[Bala94] Kavita Bala, M. Frans Kaashoek, William E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proc. First Symposium on Operating System Design and Implementation (OSDI)*, pages 243–253, November 1994.

[Chan88] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and Programming. *ACM Trans. on Computer Systems*, 6(1):28–50, February 1988.

[Chan95] David Chih-Wei Chang and et al. Microarchitecture of HaL's Memory Management Unit. *Compcon Digest of Papers*, pages 272–279, March 1995.

[Chas94] J. S. Chase, H. M. Levy, M. J. Feeley, E. D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *ACM Transactions on Computer Systems*, 12(4):271–307, November 1994.

[Denn70] Peter J. Denning. Virtual Memory. *Computing Surveys*, 2(3):153–189, September 1970.

[Eykh92] J. R. Eykholt, S. R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, D. Williams. Beyond Multiprocessing: Multithreading the SunOS Kernel. In *Proc. of the Summer USENIX Conference*, pages 11–18, June 1992.

[Good83] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proc. of the Tenth Annual International Symposium on Computer Architecture*, pages 124–131, Stockholm Sweden, June 1983.

[Henn90] John L Hennessy and David A Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[Hill84] Mark D. Hill and Alan Jay Smith. Experimental Evaluation of On-Chip Microprocessor Cache Memories. In *Proc. of the 11th Annual International Symposium on Computer Architecture*, pages 158–166, Ann Arbor MI, June 1984.

[Hill87] Mark D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. Ph.D. thesis, University of California, Berkeley, November 1987.

[Huck93] Jerry Huck and Jim Hays. Architectural Support for Translation Table Management in Large Address Space Machines. In *Proc. of the 20th Annual International Symposium on Computer Architecture*, pages 39–50, May 1993.

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

[Kane92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[Khal93] Yousef Khalidi, Madhusudhan Talluri, Michael N. Nelson, Dock Williams. Virtual Memory Support for Multiple Page Sizes. In *Proc. of the Fourth Workshop on Workstation Operating Systems*, pages 104–109, October 1993.

[Khal94] Yousef A. Khalidi, Vikram P. Joshi, Dock Williams. A Study of the Structure and Performance of MMU Handling Software. Technical Report TR-94-28, Sun Microsystems Laboratories, 1994.

[Knut68] Donald E. Knuth. *The Art of Computer Programming, Volume 3*. Addison Wesley, 1968. Second Printing.

[Lee89] Ruby B. Lee. Precision Architecture. *IEEE Computer*, 22(1):78–91, January 1989.

[Levy82] H. M. Levy and P. H. Lipman. Virtual Memory Management in the VAX/VMS Operating System. *IEEE Computer*, 15(3):35–41, March 1982.

[Lied95] Jochen Liedtke. Address Space Sparsity and Fine Granularity. *Operating Systems Review*, 29(1):87–90, January 1995.

[Lipt68] J. S. Liptay. Structural aspects of the System/360 Model 85, Part II: the cache. *IBM Systems Journal*, 7(1):15–21, 1968.

[May94] Cathay May, Ed Silha, Rick Simpson, Hank Warren. *The PowerPC Architecture*. Morgan Kaufman Publishers, May 1994.

[Mile90] Milan Milenkovic. Microprocessor Memory Management Units. *IEEE Micro*, 10(2):70–85, April 1990.

[Morr68] R. Morris. Scatter Storage Techniques. *Communications of the ACM*, 11(1):38–43, January 1968.

[Nagl94] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge, Richard Brown. Design Tradeoffs for Software-Managed TLBs. *ACM Trans. on Computer Systems*, 12(3):175–205, August 1994.

[Orga72] E.J. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, MA, 1972.

[Rama93] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, Praveen Seshadri. Implementation of the CORAL Deductive Database System. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1993.

[Repp94] John H. Reppy. A High-performance Garbage Collector for Standard ML, 1994. AT&T Bell Labs Technical Memo.

[Rome95] Ted Romer, Wayne Ohlrich, Anna Karlin, Brian Bershad. Reducing TLB and Memory Overhead Using Online Superpage Promotion. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 176–187, June 1995.

[Rose85] J. Rosenberg and D. A. Abramson. MONAD PC: A Capability Based Workstation to Support Software Engineering. In *Proc. of the 18th Hawaii International Conference on System Sciences*, pages 222–231, 1985.

[Silh93] Ed Silha. *The PowerPC Architecture, IBM RISC System/6000 Technology, Volume II*. IBM Corp., 1993.

[Sing92] Jaswinder Pal Singh, Wolf-Dietrich Weber, Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[Site93] Richard L. Sites. Alpha AXP Architecture. *Communications of the ACM*, 36(2):33–44, February 1993.

[SPAR91] SPARC International Inc. The SPARC Architecture

Manual, Version 8, 1991.

[SPEC91] SPEC. *SPEC Newsletter*, 3(4), December 1991.

[Tall92] Madhusudhan Talluri, Shing Kong, Mark D. Hill, David A. Patterson. Tradeoffs in Supporting Two Page Sizes. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, pages 415–424, May 1992.

[Tall93] Madhusudhan Talluri, Yousef A. Khalidi, Dock Williams, Vikram Joshi. Virtual Memory Computer System Address Translation Mechanism that Supports Multiple Page Sizes. Patent application filed, Serial No. 08/139,549, Sun Microsystems, October 1993. (Accepted 1995).

[Tall94] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of Superpages with Less Operating System Support. In *Proc. of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 171–182, October 1994.

[Tall95] Madhusudhan Talluri. *Use of Superpages and Subblocking in the Address Translation Hierarchy*. Ph.D. thesis, University of Wisconsin-Madison Computer Sciences, August 1995. Technical Report #1277.

[Thak86] Shreekant S. Thakkar and Alan E. Knowles. A High-Performance Memory Management Scheme. *IEEE Computer*, pages 8–22, May 1986.

[Thom74] K. Thompson and D. M. Ritchie. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, July 1974.

[Yoo93] Hyuck Yoo and Tom Rogers. UNIX Kernel Support for OLTP Performance. In *1993 Winter USENIX Conference*, pages 241–247, January 1993.

[Yung95] Robert Yung. UltraSPARC-I (Spitfire) Architecture. Technical report, Sun Microsystems, April 1995.

Appendix: Formulae

This appendix includes formulae for approximating the number of cache accesses during TLB miss handling and page table size for different page tables (Table 2). Note, however, that the results presented in Section 6.3 do not use these formulae but instead present results of simulations. We assume 4KB base pages, 8-byte mapping information per PTE, 64-bit virtual addresses, and 64-bit pointers.

The formulae for average number of cache lines accessed when searching hashed and clustered page tables assumes random, uniform distribution of virtual addresses [Knut68]. In practice, spatial locality causes non-random insertion and lookup patterns.

Page Table Type	Term	Definition/Explanation
All	*Nactive(P)*	Number of virtual address regions of size P base pages that have one or more valid mappings in the page table
Linear (all variations)	r	TLB miss ratio for accesses to translations to the first-level linear page table
	m	Average number of cache lines accessed per TLB miss to first-level linear page table
Multi-level Linear/ Forward-Mapped	nlevels	Number of levels in page table tree
	pb_i	Number of base pages mapped by a node at level i of the page table tree
Forward-Mapped	n_i	Number of PTEs or PTPs in a node at level i of the page table tree
Hashed	α	Load factor on hash table = *Nactive*(1)/#buckets
Clustered (all variations)	α	Load factor on hash table = *Nactive*(s)/#buckets
	s	Subblock factor
Clustered with Superpage/ Partial-subblock	*fss*	Fraction of page blocks (*Nactive*(s)) that use superpage or partial-subblock clustered PTEs

Page Table Type	Average number of cache lines accessed per TLB miss	Page Table Size (in bytes)	Notes
Multi-level Linear	1 + r * m	$\sum_{i=1,nlevels} 4KB \times Nactive(pb_i)$	$pb_i = 2^{9i}$
Linear with Hashed	1 + r * m	$(4KB + 24) \times Nactive(512)$	A hash table (24 byte PTEs) stores translations to the first-level linear page table
Forward-mapped	nlevels	$\sum_{i=1,nlevels} n_i \times 8 \times Nactive(pb_i)$	$pb_i = 2^{\sum_{j=1}^{i} n_j}$
Hashed	1 + α/2	$24 \times Nactive(1)$	Each PTE is 24 bytes
Clustered	1 + α/2	$(8s + 16) \times Nactive(s)$	
Clustered with Superpage / Partial-subblock	1 + α/2	$(24 \times Nactive(s) \times fss) + ((8s + 16) \times Nactive(s) \times (1 - fss))$	

Table 2: Formulae