Increasing TLB Reach Using Superpages Backed by Shadow Memory

Mark Swanson, Leigh Stoller, and John Carter
Department of Computer Science
University of Utah
Salt Lake City, UT 84112

Abstract

The amount of memory that can be accessed without causing a TLB fault, the reach *of a TLB, is failing to keep pace with the increasingly large working sets of applications. We propose to extend TLB reach via a novel Memory Controller TLB (MTLB) that lets us aggressively create* superpages *from non-contiguous, unaligned regions of physical memory. This flexibility increases the OS's ability to use superpages on arbitrary application data. The MTLB supports* shadow pages, *regions of physical address space for which the MTLB remaps accesses to "real" physical pages. The MTLB preserves per-base-page* referenced *and* dirty *bits, which enables the OS to swap shadow-backed superpages a page at a time, unlike conventional superpages. Simulation of five applications, including two SPECint95 benchmarks, demonstrated that a modest-sized MTLB improves performance of applications with moderate-to-high TLB miss rates by 5-20%. Simulation also showed that this mechanism can more than double the effective reach of a processor TLB with no modification to the processor MMU.*

1. Introduction

The size of application program virtual address spaces has increased dramatically over the years, driven by the availability of large inexpensive DRAMs and the increasing complexity of applications. On most modern machines, virtual memory is implemented by memory management units (MMU) that map virtual to physical addresses, usually on a page granularity. Typical page sizes are fixed at 4-8 kilobytes, which is quite small compared to the total amount of physical memory in most machines. This organization eases management of the physical space and limits the amount of physical storage wasted due to internal fragmentation.

To complete a load or store operation, a processor must first convert the requested virtual address to a physical address. Given the size of typical process virtual address spaces, the total number of translations is huge, and thus is stored in main memory. To reduce the overhead of performing these translations, processors maintain a cache of recent virtual-to-physical address translations called a *translation lookaside buffer* or TLB. TLB accesses are commonly on the critical timing path for load and store instructions, and thus single-cycle TLB access is a requirement in most processors.

The amount of physical memory that modern processes can access without suffering an expensive TLB miss is determined by its TLB's *reach*, which is a product of the number of entries contained in the TLB and the size of memory mapped by each such entry. Unfortunately, the size of TLBs has grown at a much slower rate than the size of process virtual address spaces, limited by the chip area required to implement large fast associative caches and the need to multi-port or replicate TLB data to support multiple load/store units. This factor combined with the small size of typical pages severely limits the reach of typical systems. For example, the HP PA8000[12] supports a 96-entry unified instruction/data TLB. When used with a fixed page size of 4 kilobytes, the resulting TLB reach is only 384 kilobytes. Considering that the PA8000's first level data cache is one megabyte, there is a clear disparity between the expected working set sizes and the ability of the TLB to support immediate access to that data. This disparity leads to high TLB miss rates for many programs, which limits performance by perturbing the pipeline, interrupting the issue of application instructions while a new translation is found and inserted into the TLB. The small reach of the PA8000

Mark Swanson is now at Intel Corporation. Current email addresses: Mark_R_Swanson@ccm.dp.intel.com, {stoller,retrac}@cs.utah.edu
This effort was sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-98-1-0101; and under SPAWAR contract #N00039-94-C-0018 and ARPA order #B990. The views and conclusions contained herein are those of the authors and should not be interpreted as necessariy representing the official policies or endorsements, either expressed or implied, of DARPA, the Air Force Research Laboratory, or the US Government.

TLB was a sufficiently serious performance bottleneck for many commercial applications that HP increased the TLB size to 120 entries for its next generation PA 8200[6]. Even so, the PA8200 TLB reach is still only 480 kilobytes when 4-kilobyte pages are used, less than half the size of the L1 cache.

Starting in the early 1990's, processor architectures began to support TLBs that allow each entry to be independently configured to map variable-sized *superpages*[14, 12, 5]. Superpages are constrained to be a power of 2 multiple of some base page size aligned on a multiple of the superpage size. In the SGI R10000[14] and HP PA-RISC 2.0[12], the base page size is 4 kilobytes, and superpages sizes can be power of 4 kilobytes from 16KB to a maximum of 16MB or 64MB. Unfortunately, a number of complexities related to using superpages has resulted in their use being restricted primarily to mapping a small number of large non-paged segments, such as the frame buffer and the non-pageable portions of the OS kernel. To date, these problems have precluded general exploitation of superpages within production operating systems[1].

Three of the most serious problems associated with general utilization of superpages are (i) the requirement that they be used only to map regions of physical memory that are appropriately sized, aligned, and contiguous[15]; (ii) the difficulty associated with determining for which regions they are suitable and economical[18], and (iii) the need for the OS to swap entire superpages on and off disk if paging is required.

In this paper, we present a mechanism that addresses problems (i) and (iii) directly, and by changing the economics of using superpages, reduces the importance of problem (ii). The proposed mechanism eliminates much of the complexity associated with using superpages by allowing them to be created from discontiguous base physical pages. We observe that the range of physical addresses addressable by many modern processors greatly exceeds the amount of actual DRAM typically present. Many modern processors export 32 to 40 physical address bits, which can address four gigabytes to one terabyte of actual memory. In the common situation where the amount of addressable physical memory exceeds the installed physical memory, we propose to use a portion of this unused physical address range to *virtualize* physical memory a second time in the main memory controller (MMC). This extra level of indirection between application virtual addresses and physical DRAM addresses makes it possible to create arbitrary-sized contiguous superpages from discontiguous fixed-sized real pages.

We introduce the notion of *shadow pages*. Shadow pages are "shadows" of real physical pages accessed at otherwise unused physical addresses that are remapped by the MMC to real physical addresses. For example, consider a system in which 32 bits of physical address are exported from the processor to the memory system and in which only one gigabyte of DRAM is installed. In such a system, only one-fourth of the physical addresses that can be generated by the processor are legal. Accesses to other addresses will usually cause serious system faults – for ease of explanation, we are ignoring memory-mapped I/O addresses in this discussion. In our proposed system, portions of the physical address space that do not correspond to real memory, but which are properly sized and aligned, can be used as superpages. The MMC translates shadow physical addresses into real physical addresses whenever a memory operation for such a region is performed by the processor. With this mechanism in place, superpages can be created from discontiguous physical pages through the use of contiguous shadow pages, accesses to which are retranslated to the corresponding discontiguous physical pages.

From the point of view of the processor and OS memory management system, shadow addresses are used in place of real physical addresses when needed. Shadow addresses will be inserted into the TLB as mappings for virtual addresses, they will appear as physical tags on cache lines, and they will appear on the memory bus when cache misses occur. The existence of shadow memory can be made completely transparent to user programs, and can be isolated to small portions of the OS. The management difficulties associated with previous superpage approaches are avoided, and only modest changes to the VM software are required. Finally, the proposed mechanism is widely applicable because it exploits the existing superpage capabilities of processor-resident TLBs *without requiring modification of the CPU or MMU/TLB*.

When we compared the simulated performance of five TLB-constrained programs on a system with a conventional TLB and on one that exploited the mechanism described above, we found that our proposed system improved application performance by from 5-20%. In effect, we found that a system with a 64-entry TLB combined with an MMC that supported shadow superpages achieved the same performance as a system with a 128-entry TLB and a conventional MMC. These results demonstrate the potential of the proposed mechanism. It is likely to be even more effective on applications with significantly larger working sets and worse spatial locality, such as is often found in large databases and other commercially important applications[17].

The rest of the paper is organized as follows. In Section 2 we present in more detail the proposed functionality, including detailed descriptions of the extra MMC translation hardware and the modest support required from the operating system's VM software. Section 3 reports the simulated

[1] SGI's IRIX6.4 appears to be the first commercially available OS with such support. Even on the SGI machines, support for superpages for user memory appears to be limited to their NUMA systems. These machines are primarily used as batch-like compute servers where applications are carefully sized to fit in memory and paging is an infrequent event.

performance of our proposed system. In Section 4 we discuss some of the limitations of our proposed solution on current systems, and possible ways to overcome these limitations. Related work is discussed in Section 5. Finally, in Section 6 we summarize the work, draw conclusions, and describe possible future uses of shadow memory.

2. Implementing Superpages Based on Discontiguous Physical Pages

This section describes in detail the mechanism we have developed for using shadow memory to allow superpages to be created from discontiguous physical pages, thereby increasing the TLB's effective reach. This mechanism is implemented through a combination of a secondary TLB implemented on the MMC and straightforward changes to OS memory management routines.

2.1. Overview

Figure 1 illustrates in detail our proposed mechanism in action. The OS has mapped a contiguous 16-kilobyte virtual address range to a single shadow superpage at "physical" page frame 0x80240. When an address in the shadow physical range is placed on the system memory bus, the MMC detects that this "physical" address needs to be retranslated using its local shadow-to-physical translation tables. For sake of discussion, assume that the processor exports 32 bits of physical address and that physical addresses above 0x8000000 are not backed by real physical memory. In this case, the MMC can easily determine which addresses are shadow addresses and which are real physical addresses, and proceed accordingly. In the example in Figure 1, an access to virtual address 0x00004080 is translated by the processor MMU/TLB to shadow physical address 0x80240080, which in turn is translated by the MMC's MMU/TLB to real physical address 0x40138080. Memory-mapped I/O devices add a small amount of complexity, since these devices are often mapped in special "high" address regions. In this case, the operating system and memory controller would need to avoid treating such addresses as shadow addresses.

Figure 1 illustrates a number of important issues in our design. First, the pages backing a superpage *need not be contiguous* in physical memory, only in shadow addresses. This lets the OS create superpages from pages that have been dispersed throughout main memory, such as occurs naturally when paging is done at the 4-kilobyte page granularity. Second, the processor MMU/TLB design is unaffected by the presence of an extra level of address translation performed at the MMC. Full support is given for protection (e.g., *read-only* and *supervisor-only* pages) and for the fine-grained accounting used by the paging mechanism to select good candidate pages for swapping (*accessed*) and determine when they need to be written back (*dirty*). Third, the physical pages backing a superpage need not even be present in physical memory as long as the MMC can generate a precise fault to the CPU whenever an access to such a page occurs. The availability of such precise exceptions is discussed in Section 4.

The protection bits (*read-only* and *supervisor-only*) are maintained only by the processor MMU/TLB, and as such must be identical for all base pages mapped by a superpage. Programs often have large segments of densely populated virtual address space that have similar access protection, which renders this restriction moot. Good examples of such regions include the text segment, data segment, and heap of a program. Such regions, or pieces of such regions, can be mapped from large contiguous virtual segments to large contiguous shadow segments, and thus are amenable to the use of superpages. In this way, virtual regions of up to 16MB or 64MB in size can be mapped with a single CPU TLB entry, which would extend the reach of a typical processor-resident TLB from well under one megabyte to several tens of megabytes.

2.2. Design of the Memory-Controller TLB

Obviously, real physical addresses are eventually required to satisfy load and store operations. We propose placing a simple secondary MMU and TLB in the main memory controller (MMC). This memory-controller TLB (MTLB) translates from shadow physical addresses to real physical addresses in one cycle, if the appropriate mapping is present in the MTLB. We believe it is possible to make this secondary TLB significantly larger than the one found in the processor, and thus increase the effective reach of the processor TLB, for four reasons:

1. The speed requirements within the MMC are much less aggressive than those within the CPU, so the slowdown that comes with increased size is easier to tolerate.

2. Unlike TLBs on modern processors with multiple load/store units, the MTLB need only be single ported, as long as the interconnect between processor(s) and the MMC is only capable of delivering one memory request to the MMC on each cycle.

3. The MTLB is simple: it supports only a single base page size

4. A larger size allows the MTLB to adopt a less aggressive structure than the full associativity that is standard in many processor TLBs[12, 14, 5].

A typical MTLB-enabled system might provide on the order of 512 megabytes of shadow virtual address space,

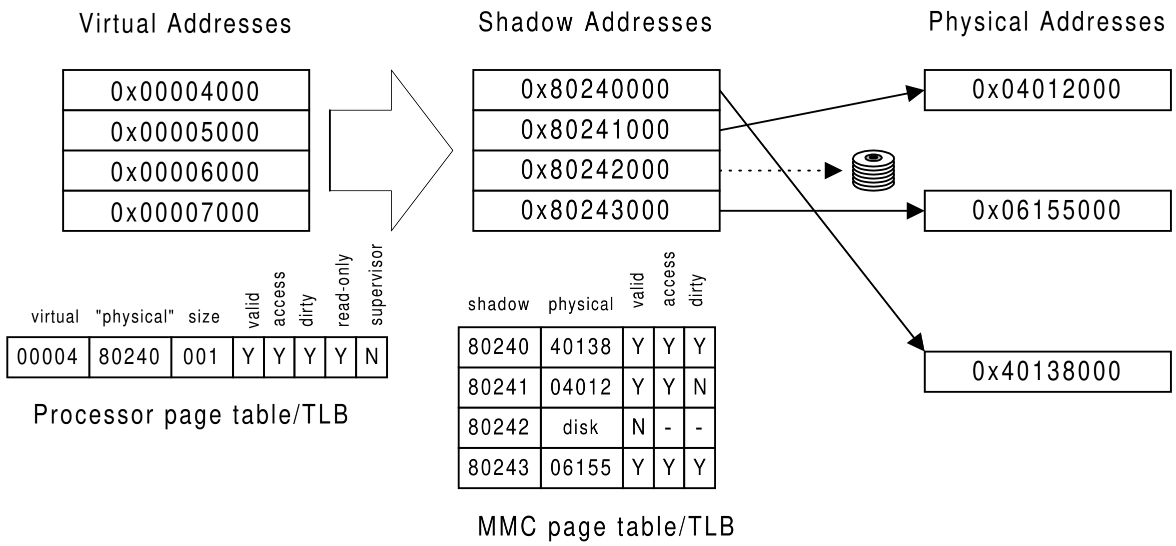

Figure 1. Detailed Example of Using Shadow Physical Regions

enough to map three orders of magnitude more memory than a typical processor TLB. In a system with 4-kilobyte base pages, the shadow-to-physical translation table would consist of 128K 4-byte entries, for a total cost of 512 kilobytes of memory. A 4 byte entry could hold a 24 bit page frame number (sufficient to map 64 GB of real memory), as well as validity, page fault, reference, and modification bits, with room left over for future expansion. For this base page size, the translation table represents an added space overhead of only 0.1% for a shadow address space equal in size to the real physical memory. Given the relatively small size of such a maximal MMC page table, we propose to use a dense, flat array indexed by shadow page offsets to store shadow-to-physical translations. We further propose to use a hardware TLB fill mechanism, which would simply need to extract the appropriate bits from the shadow physical address and use them as an offset into the main memory translation table.

The following example is designed to illustrate the normal operation of the MTLB – loading and unloading translations and managing the shadow physical address space is discussed in the following subsection. Consider an access to virtual address 0x00050040 in the example illustrated in Figure 1. If this access misses in the processor cache, a cache fill request containing the shadow address 0x80241040 is sent to the MMC. The MTLB examines the address of every cache fill request and write back, to determine if the address is real or shadow. To simplify this discussion, assume that the MTLB can determine whether an address is real or shadow using a simple comparison such as "physical address greater than installed memory," or, to handle I/O addresses, AND'ing the address with a mask of legal shadow region addresses initialized by the OS. This determination must be made as fast as possible, since it occurs on every cache fill request. In our simulations, we assume that, together with a possible MTLB lookup, this operation adds one 120 MHz MMC cycle to *every* MMC operation. This is likely an overly conservative estimate - our most recent design work indicates that we can perform this extra comparison in parallel with other bus interface related operations. Thus, there should be no impact on the performance of applications that do not employ shadow memory.

In this example, the address presented (0x00050040) is a shadow address, so the MTLB attempts to translate it to a physical address. If this lookup hits in the MTLB cache, the entry's valid bit is examined to determine if this physical page is present in main memory. If it is not, the MTLB issues an exception - this operation is described in more detail in Section 4. If the entry is valid, as it is in this case, the real page frame number from the TLB entry (0x04012) is merged with page offset portion of the shadow address (0x040) to form a real address (0x04012040) and the access can proceed as normal. As with a processor TLB, the *accessed* and *dirty* bits are updated appropriately.

If the shadow-to-physical translation lookup fails in the MTLB, an MTLB fill sequence is initiated, and proceeds as follows. Assume that shadow memory is mapped in the 512 megabytes between physical address 0x80000000 and physical address 0xa0000000. Further assume that the base of the MMC page table has been configured by the OS to be at physical address 0x00000000. The MMC fill hardware would left shift the shadow page index two bits, because the entries are 4 bytes, and add the resulting value to the base physical address of the MMC page table. In this case, the fill hardware would perform a 4-byte load to address 0x0x00009000 (0x0240 left shifted two bits and added to 0x00000000). Once the entry has been loaded in to the MTLB, the original memory access can continue as described above.

2.3. Setting Up Shadow Superpages

We envision that the normal mode of superpage creation will entail having the programmer tell the operating system to use superpages for specific portions of the address space, e.g., the data segment, at program initialization time. In our simulation experiments, however, we have used an explicit request from the user application via a `remap()` system call to cause the OS to map regions using superpages. In our experiments, we also modified the `sbrk()` library routine to map dynamically allocated data to superpages. Most of the programs that we studied performed many small allocations, so our modified `sbrk()` pre-allocates a large region, from which it satisfies subsequent small requests. Static data and text can be allocated to superpages via simple modifications to the OS loader, an operation we simulated via explicit `remap()` operations added to the benchmark code.

For superpages created after process startup, including those created as a side effect of our modified `sbrk()`, the VM system must take care to ensure that consistency is maintained in the cache hierarchy when a page's mappings change from real to shadow addresses (or back). The safest approach is simply to flush the entire (virtual) region from the cache before the mapping is changed and purge existing TLB mappings for any page being remapped. Some TLB designs automatically discard pre-existing mappings for the same virtual range and would not require explicit flushing. These costs must be considered when deciding, on a *dynamic* basis, whether to create shadow-backed superpages.

2.4. Allocating Shadow Address Ranges

Given the large physical address spaces of current processors, the shadow space can be quite large relative to physical memory. Thus, a good deal of fragmentation of the shadow region can be tolerated, allowing the use of simple, computationally inexpensive algorithms for its management.

For simplicity, we preallocate regions of shadow physical address into *buckets* of regions of legal superpage sizes, in much the same way that `malloc()` manages regions of heap memory. At runtime, we pick any available region from the appropriate sized bucket for each superpage that we desire to create. Figure 2 illustrates one possible static distribution of shadow superpages when the machine is configured to have 512 megabytes of shadow memory. Obviously, it is possible to run out of a particular sized region, which could limit the OS's ability to create an optimally-sized superpage. For the sake of our experiments, this scheme was adequate, but experience may suggest that more complex schemes, such as a buddy-system that splits and recombines superpages, as is used in most efficient `malloc()` implementations, should also be used. Such approaches would become more appealing if the mapped ranges are allowed to be sparsely populated or paged, which might require splitting regions.

Superpage Size	Count	Address Space Extent
16KB	1024	16MB
64KB	256	16MB
256KB	128	32MB
1024KB	64	64MB
4096KB	32	128MB
16384KB	16	256MB

Figure 2. Example Partitioning of a 512 MB Pseudo-Physical Address Space

Given this partitioning scheme, creating one or more superpages for a given region of virtual addresses is trivial. The mapping algorithm starts by determining the smallest superpage-aligned address larger than the specified start address of the region in question. Any small (sub-16KB) region skipped over is not remapped. The mapping algorithm then walks through the rest of the specified virtual region, creating maximally sized superpages by (i) allocating a shadow region from the the shadow region pool, (ii) initializing the MMC's shadow-to-physical page mappings for this shadow region, and (iii) setting up a real TLB superpage mapping from the relevant virtual address range to the allocated shadow region (e.g., a 16-kilobyte superpage mapping from virtual address 0x00004000 to shadow address 0x8024000). MMC shadow-to-physical mappings are initialized via uncached writes by the kernel to a special MMC control register, specifying the shadow physical address, the real physical address, and the various state bits described above. Initial configuration information, such as the range of legal shadow memory addresses and the base physical address of the MMC's page table, and purges of MTLB mappings can be handled via similar uncached writes to an MMC control register.

2.5 Maintaining Access Information

Paging activities in most operating systems require the maintenance of *reference* and *dirty* bits for each page of memory. Reference bits are usually maintained by setting a bit in the page table entry (PTE) on the first TLB miss for a given page. Accesses that should set the dirty bit can be ascertained at TLB miss time by detecting the kind of reference (read or write); or by inserting a mapping that allows only read access. In the latter case, if a write occurs, the resulting exception can then be caught, the dirty bit can be set in the PTE, and the TLB entry can be upgraded to allow further writes without faults.

With conventional implementations of superpages, the large granularity of the pages causes a significant loss of information, as only a single TLB miss and a single protection failure are generated for a whole range of base pages. The OS is forced to swap entire, potentially very large, superpages on and off disk if paging is required, which increases the effective working set size of applications by up to 60%[16]. This increases memory fragmentation and leads to less efficient use of main memory.

Our proposed MTLB alleviates this problem by maintaining reference and dirty bits on a per-base-page basis in the MMC's page table. For the MTLB, *read* and *write* accesses are *cache-fill* requests, the former for a *shared* cache line, the latter, an *exclusive* one. Unfortunately, the per-base-page reference information (*accessed*) is only approximate, because the MMC only sees *cache fill* requests – if all of a given base page's referenced lines remain in the cache after its reference bit is reset by the OS, the page will appear to be unreferenced even though it might be quite active. This could reduce the effectiveness of CLOCK and similar page replacement strategies. Evaluation of the efficacy of this detailed reference information is beyond the scope of this paper.

Unlike the reference information, the MTLB can maintain completely accurate per-base-page dirty bits, as the OS only resets this information after it has cleaned a page by deallocating it or paging it out, just as in standard systems. The cleaning process necessarily results in all the lines belonging to the page being flushed from the cache. Any subsequent writes to locations within a cleaned page will result in exclusive cache line fill requests to the MMC, which the MTLB will note. Thus, when the OS chooses to swap a shadow-based superpage on to disk, it only needs to flush back the specific base pages within the superpage that have been written, and not the entire superpage - a potentially large performance benefit for very large applications.

3. Performance Results

This section begins with an examination of some of the overhead introduced when using an MTLB, followed by a comparison of systems with and without an MTLB. It closes with an exploration of the sensitivity of the results to the MTLB size and organization.

3.1. Benchmarks

Obviously, only programs with working sets that are large relative to the CPU TLB's reach and which display poor locality will benefit from use of an MTLB. Based on preliminary measurements of the SPECint95 benchmark suite and two other applications, we selected 5 programs as likely beneficiaries of our approach: *compress95*, *vortex*, *radix*, *em3d*, and *cc1*.

Compress95 is from the Spec95 benchmark suite. The working set is dominated by the hash table and code table, which have a combined size of approximately 440KB and which are accessed in a relatively random manner. We changed compress95 to map four regions to pseudo-physical addresses: one region containing the hash table, the code table, and the intervening data structures (557056 bytes, 10 superpages), and the initial portion of the 3 buffers containing the original, the compressed and the uncompressed versions of the "file" (each was the same length, 999424 bytes, but due to differing alignments, they result in 13, 7, and 13 superpages, respectively). Compress95 is run with an initial 1,000,000 characters and is run through 2 compress/decompress cycles.

Vortex, another Spec95 benchmark, is an object-oriented database that builds several in-core databases and performs transactions against them. As the databases and transaction results are continually being allocated from the heap, the modified `sbrk()` described earlier performed all superpage creation. The region size preallocated by `sbrk()` is initially set quite large (8MB) so that the basic datasets are all mapped in one group (approximately 9MB). The increment is then reduced to 2 MB after the basic datasets are created. Another 10 MB is later mapped in five separate mappings as the result of dynamic allocation during transaction processing. We measure *vortex* with a slightly modified Spec95 training run, which dynamically allocates approximately 18 MB over the course of the run.

Radix is a Splash2 benchmark[19] sorting program. Its primary data structures are all dynamically allocated at the beginning of the program. We map the entire dynamically allocated space after the allocations are complete and before the larger structures are initialized. Radix is run with the default arguments, except that the number of keys is set to 1048576. The amount of space mapped is 8437760 bytes in length, and requires 14 superpages.

Em3d performs three dimensional modeling of electromagnetic wave propagation. The particular version used here[3] is a message passing version run on a single processor. The runs reported here model 6000 nodes and use 4.5MB of dynamically allocated space, which is remapped using 16 superpages.

Gcc is actually the *cc1* pass of the version 2.5.3 gcc compiler for the Sparc architecture, also from Spec95. Reported here is the compilation of the file "1insn-recog.c". Again, all superpage creation was performed by `sbrk()`.

3.2 Simulation Environment

The simulation results were all obtained using an execu--tion-driven simulator that models a single-issue 240 MHZ processor with cycle-accurate models of the cache, bus, and memory controller. Given that the cache is non-blocking and the processor does implement stall-on-use, we believe the results of our experiments will not change dramatically for multi-issue processors. The bus modeled is HP's Runway bus[2] clocked at 120 MHZ. The main memory controller (MMC) is similar to HP's memory controller[7] used in its HP 9000 J high end workstations. The instruction cache is assumed to be perfect. The data cache model employs a single level, direct mapped, 512-kilobyte, virtually indexed, physically tagged cache, similar to that used with the HP PA8000[12]. Cache lines are 32 bytes, hits are handled in a single cycle, and the cache is non-blocking and writeback. The CPU TLBs modeled are all unified I/D, single-cycle, fully associative, and employ a not-recently-used replacement policy. Misses are handled by a trap routine that employs a 16K entry virtual-to-physical hash table. Each entry is 16 bytes in length. The table structure used is the hashed page table model commonly used on HP PA-RISC architectures[10]. In addition to the main TLB, a single-entry micro-ITLB holding the most recent instruction translation is also modeled. Kernel code and data structures are mapped using a single *block TLB* entry that is not subject to replacement.

The times reported are complete simulation times from initialization of the BSD-based (micro)kernel, starting an *init* process, and then running the program of interest through completion of process `exit()` code in the kernel. The kernel supports the required process control (`fork()` and `exec()`), scheduling, virtual memory management, timer interrupts, and TLB miss handling. The execution time and memory accesses of these kernel operations are included in the simulation results.

3.3 Initialization Costs

All of the results reported in this section were obtained with programs instrumented to remap some or all of their static and dynamic data structures from using base pages to using superpages (and the MTLB). Generally this remapping occurred before any explicit initialization of the data by the user program, but the pages had already been zero-filled in the program's virtual address context. This transition has associated costs: finding appropriate pseudo-physical address ranges, establishing the mappings, shooting down TLB entries, and flushing the affected lines from the cache. The implementation does not try to optimize by determining which pages are dirty, nor does it rely on conflict resolution within the cache/MMC to maintain consistency implicitly. All lines of all remapped pages are explicitly flushed. Still, the cost of cache flushing is quite modest, averaging 1400 CPU cycles per 4KB page. All of these costs are included in our simulation. A comparable cost for copying a 4KB page, when the source page is warm in the cache, is 11,400 CPU cycles. The avoidance of copies in forming superpages is an obvious and significant advantage.

As an example, *em3d* explicitly remaps 1120 pages of initialized dynamic memory before initiating its time step iterations. The total cost of the system call is 1,659,154 cycles. Cache flushing accounts for 1,497,067 cycles, while all of the remaining overhead amounts to just 162,087 cycles.

3.4 Basic Results

In this section, we compare the performance of systems with and without MTLBs for a range of reasonable CPU TLB sizes, selected because they correspond to TLBs in very recent high-end processors[9, 6]. The cache size is fixed at 512KB for all of these runs. The MTLB, when present, is configured with 128 entries, is 2-way set associative, and employs a not-recently-used replacement algorithm. The simulated MTLB does not write back updated reference/modification information into its mapping table. Adding this functionality should have a negligible effect on performance. A base system for normalization purposes is defined as one with a 96 entry CPU TLB and no MTLB.

To reduce simulation time, both the database sizes and the number of transactions in *vortex* were decreased from the Spec *training* run; this serves to dampen the effectiveness of any improvements resulting from the MTLB. Likewise, *compress95* is run for only 2 compress/decompress cycles, rather than the 25 cycles used for reporting Spec results. This, too, will tend to dampen the advantages of the MTLB both by exaggerating the initial system startup time, when no superpages are in use, and by only amortizing the costs of the remapping over a very short run.

Figure 3 presents the runtimes of the programs, normalized to the time for the base system. It also separates out the fraction of total runtime spent in TLB miss handling. We examine this TLB time first. For four of these programs, a CPU TLB size of 64 results in over 20% of runtime being spent in TLB misses. For three of these programs, *em3d*, *radix* and *vortex*, miss time is still significant when the TLB size reaches 128 entries, without an MTLB. *Radix* has particularly poor TLB locality; even at 256 TLB entries, it still spends 13.5% of total runtime in TLB miss handling. In the cases where an MTLB is present, TLB miss times are below 5% in all configurations, and are essentially insignificant in all but *em3d*. These results are not fundamentally different

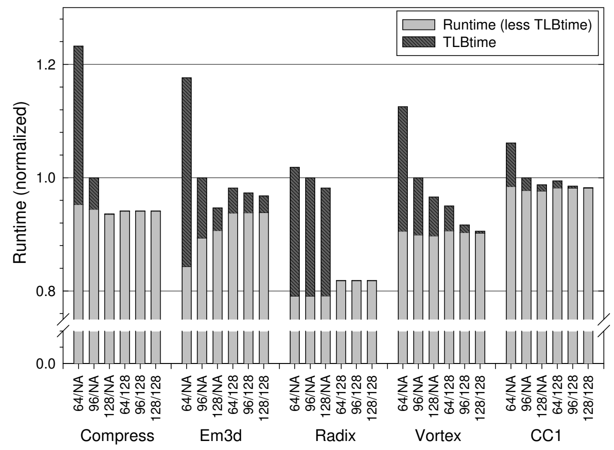

Figure 3. Normalized Runtimes for Three TLB Sizes with and without a 128 Entry MTLB

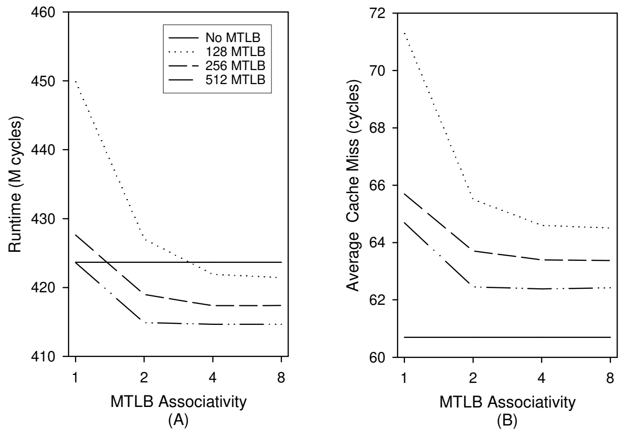

Figure 4. Em3d sensitivity To Various MTLB Sizes and Associativities

from those obtained with other superpage approaches. Of course, in the MTLB approach it is possible that CPU TLB miss time may just have been traded for increased memory time resulting from misses at the MTLB. Looking at total runtimes will at least partially address this possibility.

Examining those total runtimes, we see that, in the no-MTLB cases, all programs display monotonic improvement as the TLB size increases, dramatically for *compress*, *em3d*, *vortex*, and *cc1* and more modestly for *radix*. Also, we see that the improvements are more marked between 64 and 96 entries than between 96 and 128, with the exception of *radix*. This indicates that, for the small problem sizes measured, 128 entry TLBs are adequate, or nearly so. On the MTLB side, the decreases in TLB miss times do translate into proportional decreases in total runtime. More interestingly, the results for the cases with the MTLB change very little as the CPU TLB size increases, indicating that even with the very simple mapping approach used here and modest MTLB configuration, 64 CPU TLB entries are sufficient to map the working sets of all the programs. Because of the small footprint of the test programs compared to real commercial applications, these results are likely to underestimate the impact of this work on more realistically-sized applications.

3.5 Sensitivity Tests

Only *em3d* showed measurably better performance without an MTLB, approximately 2%, and then only with the largest CPU TLB size. It is also the program with the worst cache behavior, averaging only an 84% hit rate. Consequently, it performs proportionately many more accesses to main memory compared to the other programs, a factor compounded by its 91% hit rate in the default configuration MTLB. Therefore, we use *em3d* to examine the effect of MTLB configuration. Figure 4(A) compares the runtime of the 128 entry CPU TLB without a MTLB to that of various configurations of MTLBs. From the figure, it can be seen that the total runtime advantage of not using the MTLB can be erased by either doubling the size of the MTLB or increasing its associativity. Not surprisingly, the benefits of increasing both size and associativity is greater than either alone, and a point of diminishing returns is reached quite quickly.

Figure 4(B) reports the average time per cache fill across the same system configurations. The added delay for the MTLB cases compared to the standard system ranges from 10 cycles down to 1.5 cycles. Note that with no MTLB misses whatsoever, our conservative assumptions about the MMC's basic timing dictate a 1-cycle penalty. The additional penalties are all due to the required DRAM accesses to perform MTLB fills. This is one way in which CPU-resident TLBs can sometimes do better, since the page tables needed to service TLB fills can be cached just like other data. On the other hand, the page tables must compete with program data for cache space, which can negatively affect performance when the CPU TLB is thrashing.

4 Limitations

Availability of Free Physical Addresses. To support shadow regions, the proposed mechanism relies on the availability of physical addresses not backed by real physical memory. While this limitation is not a significant problem in typical systems, it will keep us from employing this technique unmodified on extremely high end machines in which all addressable physical memory is installed. This is unfortunate, because such systems will tend most need support for superpages given the tremendous disparity between their TLB's reach and the amount of user data that can be stored in memory. One possible approach is to make all virtual accesses use shadow physical memory, allowing the kernel to disable the MTLB whenever it needed to use real physical addresses. Because of the heavier load placed on an MTLB in such a configuration, it might be necessary to expand its size and/or associativity as indicated in Section 3.5 to maintain performance for programs not using superpages.

Imprecise Exceptions. Figure 1 illustrates the possibility that base pages that back superpages can be temporarily swapped out to disk. If these regions are accessed, the MMC must generate a precise exception to the processor so that the offending load or store instruction will be aborted, the appropriate base page loaded, and the instruction reissued. Since existing processors do not anticipate memory faults once the CPU TLB checks have succeeded, generating this kind of precise exception may require the MMC to signal the processor of the fault indirectly. For example, the memory controller could return data with bad parity as a response to the request, causing the faulting instruction to suffer a memory parity error fault. To allow the OS to distinguish between real parity errors and shadow page faults, this MTLB could mark the invalid entry as having generated a fault and write it back to memory. The OS would determine the physical address that the faulting instruction was referencing and, if it lay in the pseudo-physical range, read its mapping entry to examine the page fault bit. The OS's response to an MTLB mapping failure exception would be to treat it as a page fault. We have not yet tested this operation, as it requires a level of exception precision not available on current processors with superpage capability. We are exploring ways to work around this problem, as the ability to independently swap individual base pages within a superpage would dramatically increase the OS's flexibility to manage superpages.

Note that failures on write backs, which would be more difficult to handle since the processor is not awaiting a response, cannot happen – the OS is required to flush the dirty data back to memory *before* swapping a page to disk and removing the corresponding mapping.

5. Related Work

Chen et al.[4] report the performance effects of various TLB organizations and sizes. Their results agree with our premise that the most important factor for minimizing the overhead induced by TLB misses is TLB reach. They studied several SPECmarks programs, which have much smaller memory requirements than our benchmark programs, and found that TLB misses increased the effective CPI (cycles per instruction) by anywhere from 1 to 5 for programs with large data sets. They suggested the possibility of using variable page sizes to improve TLB reach, but did not explore the implications of their use

Talluri et al.[16] describe a method to use two page sizes simultaneously, using 32-kilobyte pages on some regions of memory to increase TLB reach, while minimizing fragmentation by using 4-kilobyte pages on sections of memory without the locality to use large pages effectively. Like Chen et al., they found that judicious use of large pages could reduce the contribution of TLB miss overheads to overall CPI by as much as a factor of 8. Exclusive use of 32-kilobyte pages increased application working sets by as much as 60%, which can lead to inefficient use of main memory. Mixing both 4- and 32-kilobyte pages kept the increase in working set size around 10%. The ability to map superpages in the process TLB without requiring the underlying "small" (4-kilobyte) pages to all be present eliminates the problem of increasing working set sizes altogether. Talluri et al. did not report application running times, so it is not clear how closely our performance results compare with theirs.

Talluri et al.[15] report many of the difficulties attendant upon general utilization of superpages, most of which result from the requirement that superpages must map regions of physical memory that are contiguous and aligned. They propose *subblock* TLBs as a way to mitigate some of the cost of solving these problems in the OS' virtual memory system. Again, by using shadow memory and not requiring that an entire superpage be present in main memory, our design eliminates most of the problems that they attempted to address with subblock TLBs without requiring any changes to processor TLB designs. Like our system, their proposed *complete-subblock TLB* can create superpages from discontiguous physical pages. However, to accomplish this functionality, complete-subblock TLBs must contain a complete set of physical page mappings for each superpage, which will severely limit the maximum superpage size for an on-processor TLB. We move these mappings to the memory controller, where large tables can be stored easily.

Romer et al[18] address the problem of selecting regions that can effectively benefit from mapping via superpages. They propose a mechanism for dynamically *promoting* regions for mapping with superpages given the costs, especially copying, and potential benefits of the promotion. A similar mechanism would be useful in the kernel of a machine exploiting shadow memory, although the specific parameters would need to be tweaked to reflect the reduced cost of exploiting superpages in our design.

6 Conclusions

We propose to support *shadow memory* via an additional level of address translation performed by the main memory controller. By supporting shadow memory, *superpages* can be composed of discontiguous, unaligned pages of physical memory. Unlike conventional uses of superpages, the memory controller TLB (MTLB) mechanism avoids the need for superpage management of real memory by the OS, with its severe physical memory alignment and contiguity requirements. In addition, it largely avoids the loss of page-grained access information, such as *reference* and *dirty* bits, caused by the use of conventional superpages. This mechanism can be applied effectively to existing programs without restructuring them in any way, and does not require modifications to conventional CPU or MMU designs. All of these factors combine to result in system whereby superpages can be used aggressively and efficiently for user text/data, thereby significantly increasing the effective *reach* of CPU TLBs. This, in turn, improves the performance of programs with large working sets, such as those commonly used in commercial settings.

Simulations show that a combination of modest-sized CPU TLBs and MTLBs allows system builders to extend a CPU with a modest TLB reach to perform comparably to a high-performance CPU by adding TLB capacity in the memory controller. They further demonstrated that for fixed-sized CPU TLBs, the addition of an MTLB improves the performance of applications with non-trivial working sets by from 5-20%.

This work has been done in the context of the Impulse project [8], in which we are investigating ways to improve memory system performance without modifying conventional cache, memory bus, or DRAM organizations. In the future, we intend to investigate several ways to further exploit the existence of shadow memory and an MTLB. We are currently exploring ways to use shadow memory to implement no-copy page recoloring[1] and MMC-provided stream buffers[11, 13]. In addition, we continuing to investigate whether swapping of base pages within a superpage can be supported in the context of current CPU/memory interface limitations. We also are investigating the space of practical MTLB implementations to determine what organizations and latencies are feasible. Finally, many possible TLB extensions have been proposed but not found their way into general purpose processors - modifying an MTLB may be the way to finally realize some of these ideas.

References

[1] B. Bershad, D. Lee, T. Romer, and J. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, Oct. 1994.

[2] W. Bryg, K. Chan, and N. Fiduccia. A high-performance, low-cost multiprocessor bus for workstations and midrange servers. *Hewlett-Packard Journal*, 47(1):18–24, February 1996.

[3] S. Chandra, J. Larus, and A. Rogers. Where is time spent in message-passing and shared-memory programs? In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 61–73, Oct. 1994.

[4] J. B. Chen, A. Borg, and N. P. Jouppi. A simulation based study of tlb performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 114–123, May 1992.

[5] J. Edmondson, et al. Internal organization of the Alpha 21164, a 300-mhz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal*, 7(1), 1995.

[6] L. Gwennap. Hp pumps up pa-8x00 family. *Microprocessor Report*, 10(14), October 1994.

[7] T. Hotchkiss, N. Marschke, and R. McClosky. A new memory system design for commercial and technical computing products. *Hewlett-Packard Journal*, 47(1):44–51, February 1996.

[8] `http://www.cs.utah.edu/projects/impulse`.

[9] Intel Corporation. *Pentium Pro Family Developer's Manual*, January 1996.

[10] J.Huck and J. Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 39–50, June 1993.

[11] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[12] G. Kane. *PA-RISC 2.0 Architecture*, 1996.

[13] S. McKee and W. Wulf. Access ordering and memory-conscious cache utilization. In *Proceedings of the First Annual Symposium on High Performance Computer Architecture*, pages 253–262, Jan. 1995.

[14] MIPS Technologies Inc. *MIPS R10000 Microprocessor User's Manual, Version 2.0*, December 1996.

[15] M.Talluri and M. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 171–182, Oct. 1994.

[16] M.Talluri, S. Kong, M. Hill, and D. Patterson. Tradeoffs in supporting two page sizes. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 415–424, May 1992.

[17] S. E. Perl and R. Sites. Studies of Windows NT performance using dynamic execution traces. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, pages 169–184, October 1996.

[18] T. H. Romer, W. H. Ohrlich, A. R. Karlin, and B. Bershad. Reducing tlb and memory overhead using online superpage promotion. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 176–187, June 1995.

[19] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.