Improving TLB Miss Handling with Page Table Pointer Caches

Michael Wu
Dept. of Electrical and Computer Engineering
Rice University
mikewu@rice.edu

Willy Zwaenepoel
Department of Computer Science
Rice University
willy@rice.edu

December 12, 1997

Abstract

Page table pointer caches are a hardware supplement for TLBs that cache pointers to pages of page table entries rather than page table entries themselves. A PTPC traps and handles most TLB misses in hardware with low overhead (usually a single memory access). PTPC misses are filled in software, allowing for an easy hardware implementation, similar in structure to a TLB. Since each PTPC entry refers to an entire page of page table entries, even a small PTPC maps a large amount of address space and achieves a very high hit rate.

The primary goal of a PTPC is to lower TLB miss handling penalties. The combination of a TLB with a small PTPC provides good performance even in situations where standard TLBs alone perform badly (large workloads or multimedia applications). The advantage of this design is that we can continue to use small fixed size pages with standard TLBs. Since PTPCs use traditional page table structures and page sizes, they are very simple to implement in hardware and require minimal operating system modifications.

Our simulations show that the addition of a PTPC to a system with a TLB can reduce miss handling costs by nearly an order of magnitude. Small PTPCs are extremely effective and the combination of small to medium sized TLBs coupled with small PTPCs are an efficient alternative to large TLBs.

1 Introduction

A translation lookaside buffer (TLB) acts as a cache of page table mappings used for virtual address translation. With each memory access, the TLB is presented a virtual address. If the address hits in the TLB, virtual address translation adds little or no overhead to the memory access. If it misses, a more costly hardware or software handler is invoked to load and insert the required page table entry into the TLB so that the access can proceed.

For many programs, a TLB provides a very high hit rate and address translation overhead represents an insignificant part of the overall execution time [3]. However, programs with large working sets or applications that spend significant time communicating with other address spaces can spend as much as 50% of total execution time handling TLB misses [2, 6, 9].

In the future, TLB miss handling time can become more of a problem because both hardware and software miss handlers have costs that are increasing relative to the cost of executing an instruction. Hardware handlers must search more levels of a page table directory as the address space increases, resulting in as many as 6 memory references to fill a single miss in a 64 bit address space. Beyond the obvious instruction processing costs involved with a software handler, performance is becoming more and more impacted by interrupt processing overhead in modern superscalar processors. In general, the more parallelism a superscalar processor can exploit in the steady state case, the more work is lost when execution is halted to handle an asynchronous interrupt (with precise exception handling).

Most recent work in TLB performance enhancement has revolved around the use of superpages to increase the reach of a basic TLB with some very simple extensions of the basic design and significant operating system support [3, 7, 9, 10]. There are several drawbacks to this approach. Larger pages reduce efficiency because they increase internal fragmentation and makes the unit of protection and sharing more coarse-grained. Attempts to remedy the situation using a combination of small pages and larger pages when possible have shown that it is possible, but requires extensive operating system modifications [7, 9].

Page table pointer caches cache pointers to pages of page table entries rather than page table entries as a TLB does. On a TLB miss the PTPC is checked and in most cases, it can refill the TLB with a single single memory access to the page table (see Figure 1). PTPC misses are handled in software. Overall performance in enhanced because the average number of memory references needed to fill a TLB miss is reduced and instruc-

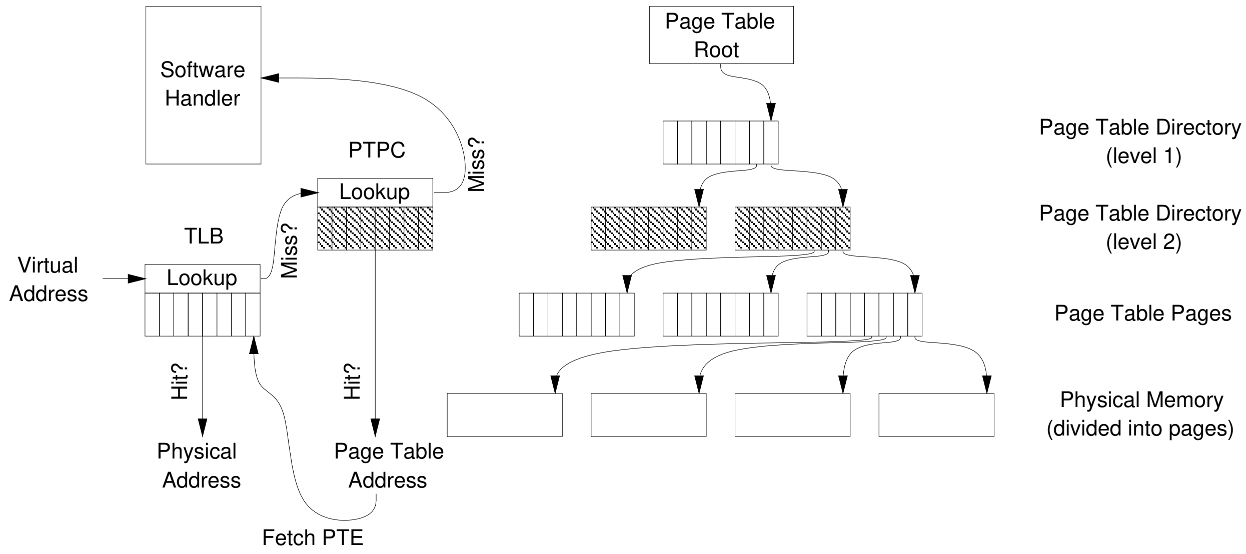

Figure 1: High level diagram of a system with a page table pointer cache. A PTPC caches entries in the page table directory that point to pages of page table entries rather than page table entries as a TLB does. On a TLB miss, the PTPC is checked and in most cases, it can refill the TLB with a single memory access to the page table.

tion processing and interrupt service overheads associated with software handlers are eliminated.

The goal of a PTPC is to reduce TLB miss handling time while minimizing operating system modifications and retaining the ease of use and granularity benefits of using small fixed size pages. An additional benefit of our design is that it is backwards compatible with existing software.

This paper describes page table pointer caches. Section 2 gives an overview of basic page table structures and TLB functionality. The design of a page table pointer cache is described in Sections 3 and 4. We simulate the design and evaluate design parameters in Sections 5 and 6. Contributions from related work are summarized in Section 7. Finally, we draw some conclusions about the design in Section 8.

2 Background

A system with virtual memory support consists of an array of physical memory divided into fixed sized pages and a page table that maps a virtual address space to the physical address space. The page table is a simple array of virtual to physical address translations called page table entries (PTEs). A PTE holds the physical frame address of a page (the starting address of the page in physical memory) along with some status bits that are used to provide protection and access control. A given virtual address can be translated into a physical address by searching the page table for the PTE that corresponds to its virtual page number. The physical frame address found in the PTE is substituted for the virtual page number and the offset within the page is added to form a complete physical address. The the root of the page table is changed with each context switch, allowing each process to have an independent virtual address space.

Various page table organizations exist, each of which satisfies different constraints in two main areas, space efficiency and search time. Forward mapped page hierarchical page tables have been commonly been used in modern operating systems because they represent a good tradeoff between size, speed, and simplicity. We will focus on hierarchical page tables for this discussion, relevance of our work to other organizations is discussed further in Section 7.

A typical forward mapped page table splits the page table proper into smaller page tables, each containing a single page worth of page table entries. This makes the page table itself pageable and able to be relocated just as with any other memory structure. A hierarchical page table directory is imposed on top of the page table with each level consisting of wide page sized nodes that contain arrays of pointers to the next level in the tree called page table pointers (PTPs). The directory allows the page table to be searched quickly and provides a mechanism for handling sparse address spaces (the nodes may be sparsely populated).

An example of a forward mapped page table which translates a 48 bit virtual address space to a 48 bit physical address space is shown in Figure 2. In this example, a fixed page mapping size of 4K is used with eight byte page table entries. Since a page is 4096 bytes long and each PTE is eight bytes, 512 PTEs exist on each page, mapping a total of 2Mb of address space. Other organizations exist based on different page sizes, virtual address size, and physical address size. Each level represents a different portion of the virtual address space. For example, the first level represents the most significant 9 bits, the second represents the next 9, and so on.

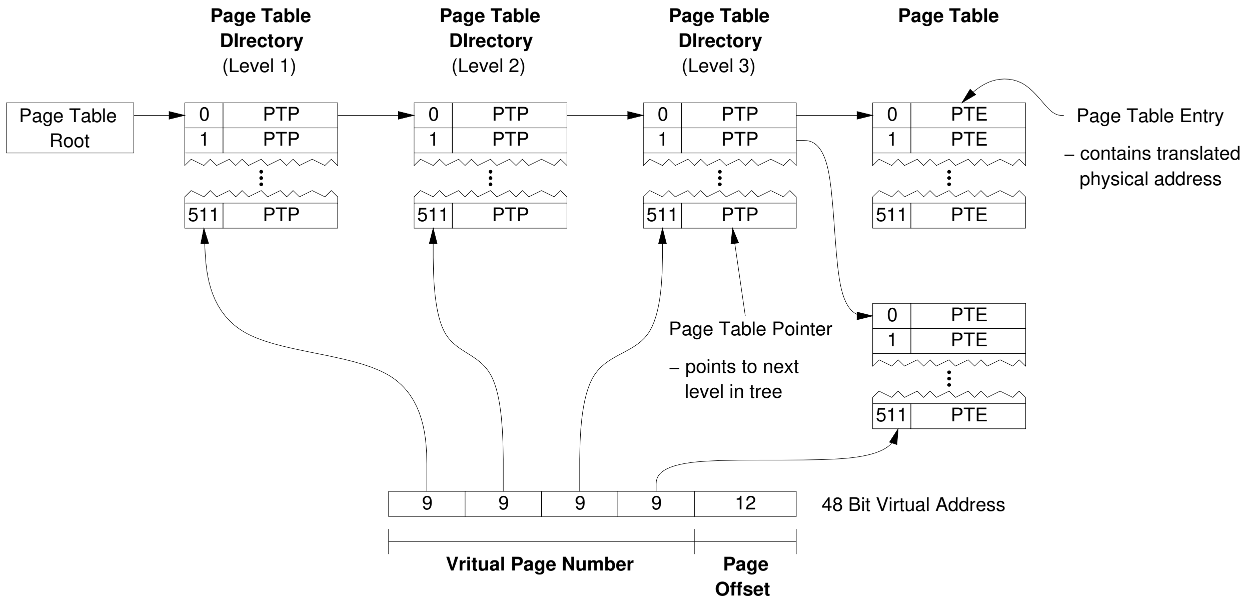

Figure 2: A forward mapped hierarchical page table.

Each pointer in the final level of the directory points to a single page of the page table which, in our example, represents the final 12 bits of the address space.

When using a forward mapped page table, a virtual address is translated into a physical address by walking down the page table directory structure. The cost of this walk is determined by the number of levels in the page directory. Each transition between levels requires one memory reference to load the pointer to the next level. In the example given in Figure 2, the processor must perform three memory accesses to get enough information to translate a full virtual address into a physical address. The structure of the page table directory is very simple and regular, allowing a relatively small amount of hardware to walk through the page directory.

The cost of searching even a shallow page table structure is prohibitive considering the high frequency of memory accesses in a program so virtual address translations are often stored in a TLB. Common TLBs are fully associative content addressable memories that return a physical page frame number when given a virtual page number (the offset within a page remains the same). Each entry in the TLB typically maps a page of virtual address space to a page of physical address space. The TLB is in the critical path of each memory operation in most systems and therefore must be made as fast as possible. This limits the practical size of a TLB to around 64 or 128 entries.

When using small pages, as is standard in current operating systems, a TLB can exhibit very poor behavior and TLB miss time can contribute significantly to overall program execution time. Poor performance arises when the reach (amount of memory mapped) of the TLB is not enough to map the working set of a program and the TLB starts thrashing. Two categories of programs show the worst behavior. The first are programs with large data sets whose working sets also tend to be larger. The second are processes that interact heavily with the operating system or other processes, effectively forcing the TLB to map a larger address space encompassing multiple contexts.

Research in the area has supported the notion that extending the reach of a TLB is crucial to achieving high hit rates in the future, and that increasing the page size in some way has potential for reaching this goal [3, 11]. Since the number of TLB entries is somewhat fixed because of speed constraints, the most common approach to increase the reach of a TLB is to enhance each entry to support larger pages called superpages. Usually the size of a superpage may range from as little as 4 kilobytes to as large as 4 megabytes. Superpages have been successfully used to map large static structures such as frame buffers and various kernel data structures. However, using larger pages for general purpose user data has several drawbacks including more internal fragmentation as the page size is increased, reduced granularity for protection and sharing, and reduced backwards compatibility because of historical architectural dependencies on page size. Simply put, while the ability to extend TLB page size support can be done very cheaply in hardware, the finer granularity of using smaller (4K-8K) pages is still very desirable since it allows for very simple, efficient, and convenient utilization of a virtual memory system.

Operating system support for concurrent variable sized pages attempts to overcome the limitations of fixed size superpages by allowing several small pages to be grouped into a single larger superpage when possible, but adds significant complexity with questionable benefit [9]. This relatively simple concept of "page promotion" greatly complicates several resource management decisions and kernel data structures that have tradi-

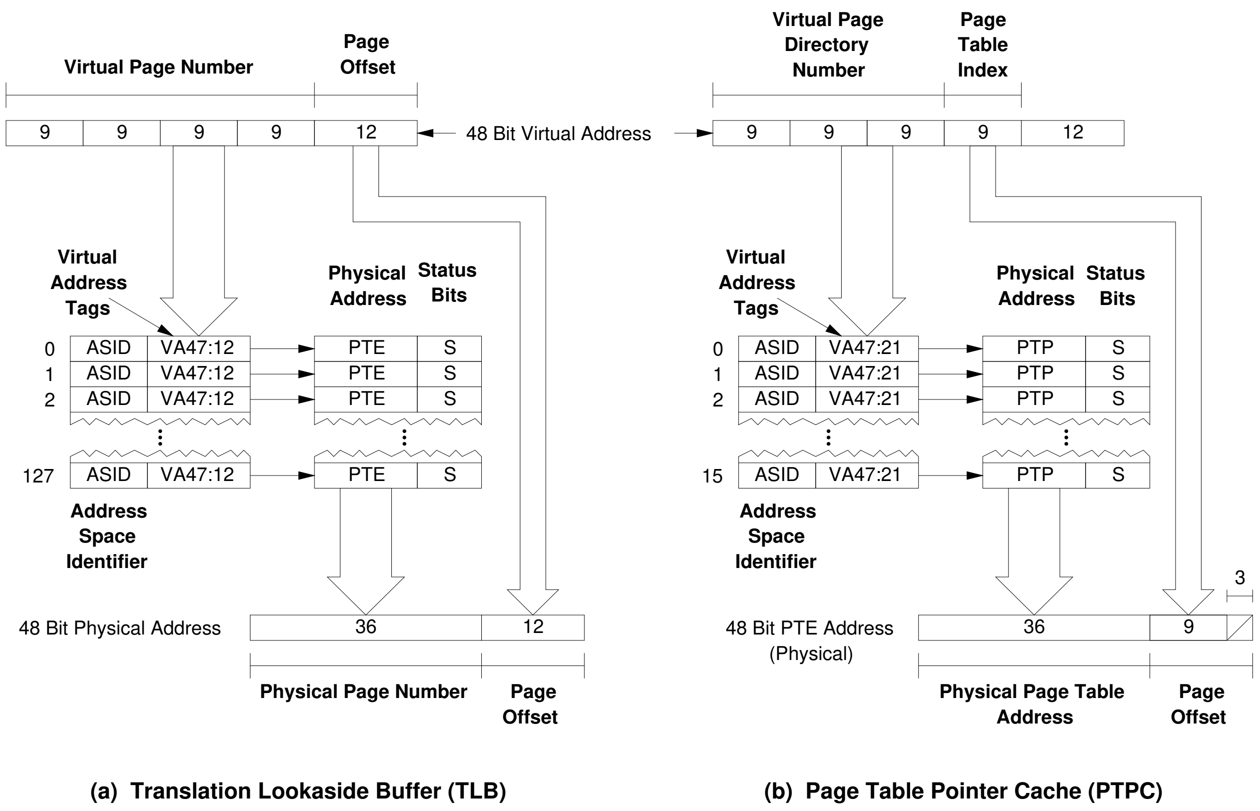

Figure 3: Detailed architecture of TLB and PTPC.

tionally been simple because memory cannot continue to be treated as a pool of homogeneous pages. [7, 9, 11]. These areas include memory allocation policies, paging policies, and protection guarantees made by the virtual memory system. Even grouping and ungrouping policies, the core of the implementation, are difficult and potentially costly. Superpages must be made up of pages that are contiguous in both virtual and physical address spaces and start on an address boundary which is a multiple of the superpage size. The operating system must constantly reallocate and move pages to meet these mapping constraints in order to gain the benefits of superpages. Partial sub-block TLBs relax some constraints by allowing more flexible mappings but still require similar operating system support.

3 Page Table Pointer Caches

Given that small, fixed sized pages are easiest for both the user and operating system to deal with, and that it is difficult to further increase a TLB's hit ratio by adding more entries, we focus on improving TLB performance by reducing its miss handling cost. Since TLB misses are relatively infrequent and performance is influenced more significantly by other factors such as cache behavior, TLB miss handling time can only become a significant portion of overall run time if the miss handler is slow.

Both hardware and software miss handlers have growing page table lookup costs. A hardware miss handler is a small piece of hardware that walks down the page table directory and fills the TLB automatically. It must perform the minimum of one memory access per level in the page table directory. On older with 32 bit address spaces, this cost is minimal because the page table directory is only 2 or 3 levels deep. However, as the address space is increased, the number of memory accesses also grows. A full 64 bit version of the page table in Figure 2 requires 6 memory accesses per miss. Processors that handle TLB misses in hardware include the Intel 80x86 processors and most Sun SPARC processors.

A software miss handler incurs two additional overheads. On each miss, an interrupt is generated and a software handler is executed. Since software is slower than specialized hardware, this handler takes more time to search the page table directory. Interrupt latency is also a problem. Modern superscalar processors must wait for all outstanding instructions to complete before servicing an interrupt, which can take a substantial number of cycles. After handling the interrupt, more cycles are spent restarting the instruction that caused the miss. All in all, many cycles are wasted, during each of which several instructions could have potentially been executed. Software handlers have the advantage that the hardware doesn't need to know about the structure of the page table structures. This, along with the fact

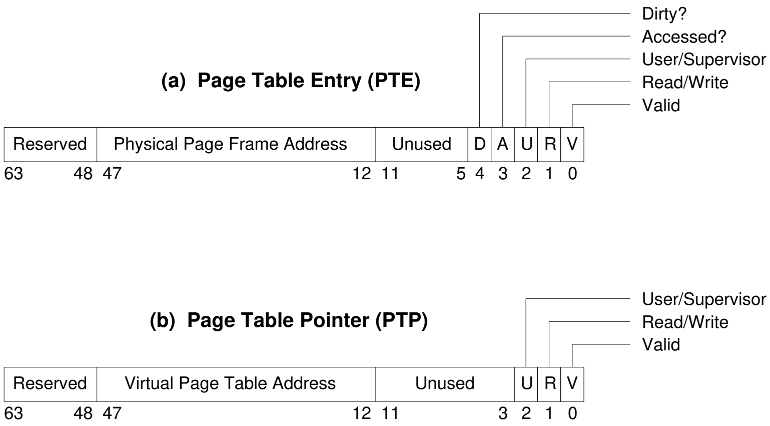

Figure 4: PTE and PTP field formats.

that less hardware support is needed in the processor makes software TLBs common among newer processors such as the MIPS R10000, Sun UltraSPARC, and DEC Alpha.

A page table pointer cache (Figure 3) is a small fully associative memory similar to a TLB that caches page table pointers rather than page table entries. Each entry represents one element (PTP) in the final level of the page table directory and points to an entire page of the page table (see Figure 1), which contains many PTEs. When a TLB miss occurs, the virtual address is passed to the PTPC. If a virtual address match is found, a hardware handler looks at the page table pointed to by the PTP (indexed by the lower bits of the virtual page number) and loads the appropriate PTE from memory into the TLB. If there is no match in the PTPC, a software handler is called to walk the full length of the page table directory.

In most cases, a PTPC can reduce the cost of handling a TLB miss to a single memory access. In the example page table structure given in Figure 2, each page of the page table maps 2MBytes (512 PTEs * 4K) of virtual address space to physical address space. Since each PTP references a page of the page table, it too maps 2MBytes of space, although a memory access to the page table must be made to complete the translation. When a virtual address hits in the PTPC, the associative lookup essentially removes the need to search the entire hierarchical page table directory for the desired PTP. Since each entry maps so much space, a PTPC can have hit rates much higher than that of a TLB. In fact, a single PTPC entry maps an amount of address space equivalent to the combined contents of 4 to 8 conventional TLBs.

By combining simple hardware with simple software, a PTPC can greatly reduce the amount of time spent handling TLB misses. Since the hardware can satisfy most TLB fills on the fly, the instruction stream never gets stalled to handle an interrupt. In modern superscalar processors with dynamic execution and non-blocking loads [5], only the instruction that caused the TLB miss actually gets stalled. The cost of the PTPC lookup and associated PTE fetch from memory can be partially, and in some cases entirely, hidden by overlapping the operation with the execution of other instructions. In addition, there is no software overhead. The operating system still handles certain cases such as a PTPC miss (which is rare) and cascading faults (see Section 4) so the hardware involved can remain extremely simple.

A PTPC is particularly attractive because it can be implemented as an unobtrusive, backwards compatible extension of a normal memory management unit (MMU). Unless it is enabled, neither the user or operating system software needs to be aware of a PTPC's presence and can use the normal software or hardware handler associated with that particular processor architecture.

PTPCs require about the same amount of hardware to implement as a TLB with the same number of entries. It is slightly smaller and faster than a TLB of the same size since it checks fewer bits of the virtual address, so we do not expect the introduction of a PTPC to hinder the performance of any other part of the processor.

4 Implementation Details

Each process operates a separate virtual address space and just as with a cache or TLB, a PTPC must be aware of which address space a memory reference applies to. The cheapest way to do this is to flush the PTPC with each context switch but this is wasteful. All cached entries would have to be reloaded when the process is resumed. We handle the multiple address space problem by including an Address Space IDentifier (ASID) with each virtual address tag. The ASID is set on a context switch and is unique to each process. Most TLBs use the same solution so support for an ASID is usually already part of the processor architecture.

A few modifications to the operating system must also be made to support PTPCs. First, the PTPC must be kept consistent with the page table. If the physical page table referenced by a PTPC entry is moved or

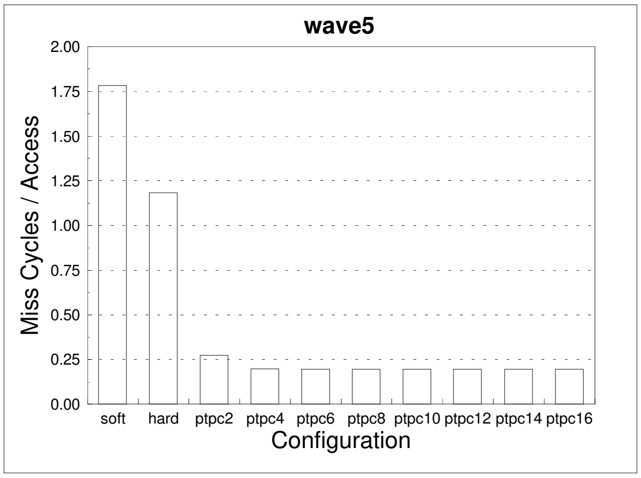

Figure 6: Miss handling overhead for wave5.

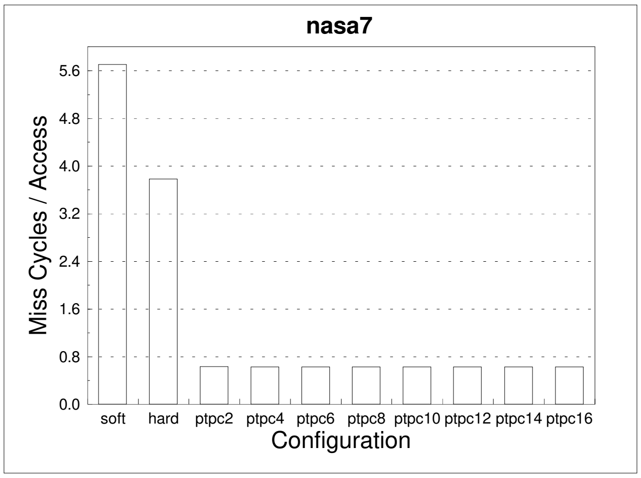

Figure 7: Miss handling overhead for nasa7.

swapped out (i.e., the page mapping is changed), the PTPC entry must be updated or invalidated. For the PTPC to be effective, the page table it references must also be present, so it is probably prudent to pin it in physical memory while it is cached. TLBs and PTPCs are both caches of the page table and suffer from similar problems. PTPC consistency operations can be included as part of the TLB support.

There are situations where the handling of a PTPC entry is slightly different than that of a TLB entry. First, when an access is done to a physical page that has been swapped out, the address will miss in the TLB but can hit in the PTPC. The PTPC will try to load the appropriate entry into the TLB. In this case, the PTPC should generate a page fault when it loads a PTE whose valid bit is clear (Figure 4), indicating that it is not present rather than install a useless entry. The second case arises when a PTPC miss is being filled. The page table directories contain pointers to other nodes in the page table directory tree which resides in mapped kernel space. The PTPC entry must be loaded with the physical address of the page table page it refers to, not the virtual address of the page specified by the PTP in the last level of the directory. If the PTPC stored virtual addresses, it would have to do a TLB lookup when filling a TLB miss to locate the page table in physical memory before the PTE fetch is initiated. This creates additional pressure on the TLB and reduces its performance for real processes. It is better for the kernel to handle the virtual to physical translation during miss handling than pay a penalty for each hit in the PTPC.

A PTPC makes it easy to handle cascading page faults. Since hierarchical page tables are commonly placed in virtual memory, it is possible for a TLB miss to cause other TLB misses while walking down a page table. Handling these cascading faults completely in hardware requires the hardware to save enough state so that it can recover from a worst case situation involving cascading faults. This state grows with the number of levels in the page table directory, becoming more expensive as the size of the address space increases. Software solutions can easily solve this problem by storing a small amount of extra state on the kernel stack. Since a PTPC searches the page table directory in software, it does not require additional hardware to handle cascading faults.

5 Simulation Methodology

In order to test the effectiveness of PTPCs, we gathered data access traces from four benchmark programs running on an SGI Power Challenge with four 75MHz MIPS R8000 processors. Wave5 and nasa7 are numeric, data intensive programs from the SPEC92 benchmark suite. Vortex is an object oriented database from the SPEC95 benchmark suite. Tpca is a hand-coded version of the TPC-A database benchmark which simulates accesses to the main data structures in a scaled down (16 Mbyte) subset of the database.

Full traces are gathered from all programs using the SGI/MIPS program pixie [8]. Pixie is a post-compilation tool that annotates each load and store with code that records their reference addresses at run time. Traces are consumed on-the-fly by our simulator. It emulates a fully associative 64 entry TLB with page table pointer caches varying in size from 2 to 16 entries. A 4K fixed page size and random replacement is used for all structures. We simulate a full 64 bit address space using the page table directory structure defined in Figure 2 with the page directory structure extended to five levels to accommodate the extra address bits.

These results only include the data accesses because the instruction TLB miss rate is low and does not significantly to overall performance. The only noticeable effect of including instruction accesses is that an additional PTPC entry is needed for each process. Since a PTP maps such a large amount of space, a single entry will almost always map all of the code.

				PTPC Misses (# of entries)							
Workload	**Total Accesses**	**TLB Misses**	**%Miss**	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**
wave5	1,200,110,000	11,260,200	0.94%	536,589	6781	100	20	15	13	11	10
nasa7	5,088,420,000	152,797,000	3.00%	19,554	6	6	7	7	4	5	5
tpca	2,176,010,000	12,109,300	0.56%	5,513,470	3,039,260	1,398,480	14	20	11	9	10
vortex	45,957,100,000	254,784,000	0.56%	141,111,000	77,267,600	30,238,100	18	12	12	12	10

Figure 5: Summary of simulation results.

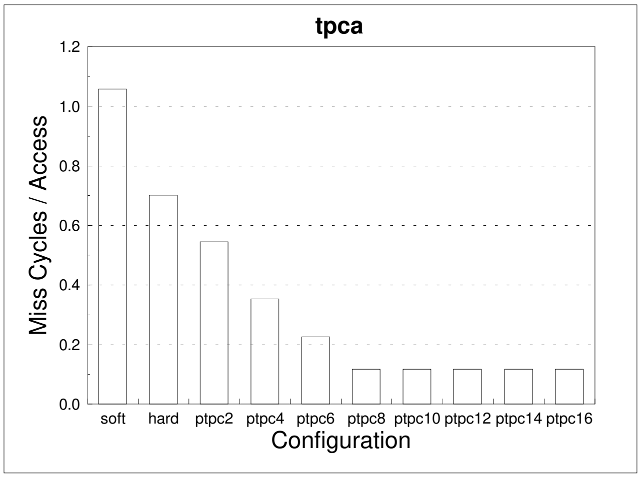

Figure 8: Miss handling overhead for tpca.

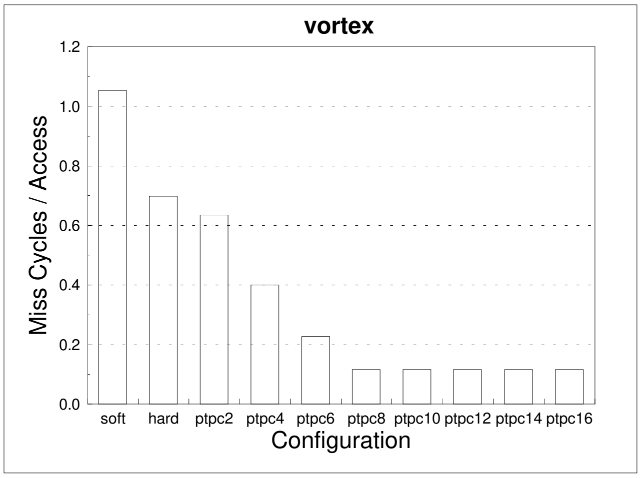

Figure 9: Miss handling overhead for vortex.

6 Simulation Results

We simulated the miss behavior of the TLB under the four workloads with and without the addition of various sizes of page table pointer caches. A summary of our results is given in Figure 5. All of the accesses of each workload were analyzed by our simulator, which determined which accesses missed in the TLB. These accesses were then passed to independent simulations of each size of PTPC. The overall TLB miss rate is relatively low, on the order of a few percent, but the cost of handling each miss is large. The time spent in the miss handler is significant. As the number of PTPC entries approaches eight, the number of PTPC misses drops to close to zero. A more detailed performance profile of each workload is shown in Figures 6 to 9. In these graphs, the miss handling overhead for standard TLBs with software and hardware handlers is compared to a TLB enhanced with a PTPC.

The cost of miss handling in each TLB/PTPC configuration is determined by amortizing the total number of cycles wasted handling TLB misses among all data accesses performed by the workload. This cost represents the average number of cycles added to each memory access by address translation overhead. A perfect hit rate in the TLB gives a cost of zero, a cost of one indicates that latency was increased by one cycle, etc.

A simple model is used to calculate the miss handling times used in the simulation. In this model, it is assumed that software handlers will have a single interrupt vector dedicated to handling all types of TLB or PTPC misses and that PTPC lookups and hardware handling decisions can be made in a single cycle. Software overhead is estimated at 10 cycles for interrupt overhead and 10 cycles of instruction overhead to fetch a PTP or traverse a page directory level (derived from the MIPS R3000 instruction set). The main cost of TLB miss handling, the actual memory access, is assumed to be 20 cycles.

Wave5 and nasa7 are scientific benchmarks with fairly high degrees of locality but working sets that are too large to fit in a small secondary cache (256K). Even though they have high TLB miss rates (especially nasa7), even small PTPCs can map their entire working sets. As Figures 6 and 7 show, the cost of address translation is reduced by about a factor of nine, almost an order of magnitude, with the introduction of just a four entry PTPC.

Tpca and vortex (Figures 8 and 9) are database benchmarks that still have a fair amount of fine grained locality but whose access patterns are less defined, blurring the distinction between their working sets and their entire data sets. For these workloads, a larger number of PTPC entries (eight) are needed to reduce miss penalties to a single memory cycle, but overall miss handling times can still be reduced by roughly an order of magnitude. In addition, the data shows that performance of a PTPC degrades gracefully as the number of entries is decreased and that a PTPC can improve performance even in cases where they cannot map the entire working set.

Figures 10 through 13 show how the miss handling costs vary as the size of the TLB is increased or decreased. These graphs expose more subtle differences between the applications.

Nasa7 exhibits the worst TLB behavior of all the benchmarks, the miss rate with a 64 entry TLB is three times greater than the second worst, wave5. Even with a large 128 entry TLB, miss handling penalty still contributes heavily to memory latency. The poor behavior arises because it makes more uniform use of a medium sized working set (just under 4MBytes). Since this behavior cannot be contained in a simple TLB, large numbers of misses are inevitable and the reduced miss handling time of a PTPC greatly improves performance. Tpca and vortex show similar behavior but it is not as pronounced because their use of compact indexing structures still allows for acceptable TLB hit rates.

In general, until the working set is entirely mapped by the PTPC, adding just a couple of entries to the PTPC is comparable in effect to doubling the TLB size. Even for programs with large data sets, such as the ones we are presenting, very small PTPCs can have a large effect on performance. In particular, the difference between having no PTPC and a four entry PTPC is enormous. Those four entries provide almost all of the order of magnitude type benefits offered by a PTPC for a single process. Ultimately though, peak performance is still determined by the miss rate of the TLB.

In is interesting to note that the combination of a small to medium sized TLB (32 entries) and a small PTPC provides close to optimal performance. This result would seem to suggest that a PTPC may be attractive to cost and power sensitive designs such as embedded controllers and mobile processors where cost efficiency and other constraints are more important than peak performance.

7 Related Work

The Sun SPARC processor walks the page directory in hardware and has three specialized registers to hold recently used page table pointers. One register contains the root of the page table for the current context. Another register holds the last PTP used to fill a TLB miss caused by an instruction. The final register holds the last PTP used to fill a TLB miss caused by a data reference. These registers reduce TLB miss time with a minimal amount of hardware but are too few in number to help with TLB reach problems or reduce costs when switching between contexts. The fixed function nature of the registers also limit their effectiveness. The SPARC processors that incorporate this feature use virtually tagged caches and only need to reference the TLB on a miss, reducing the need to have a fast TLB. Wood

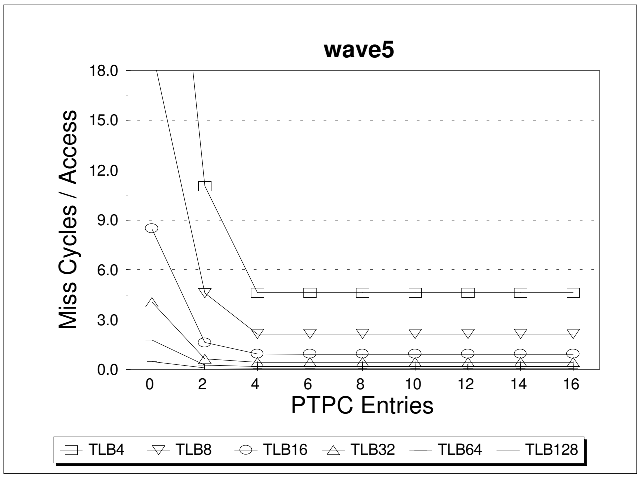

Figure 10: PTPC vs. TLB size tradeoffs for wave5.

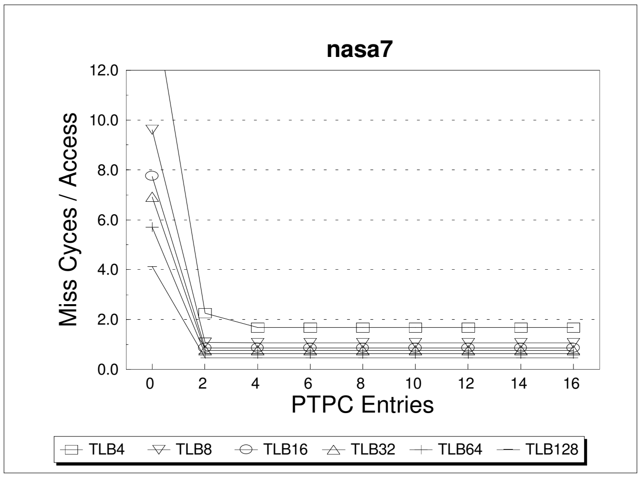

Figure 11: PTPC vs. TLB size tradeoffs for nasa7.

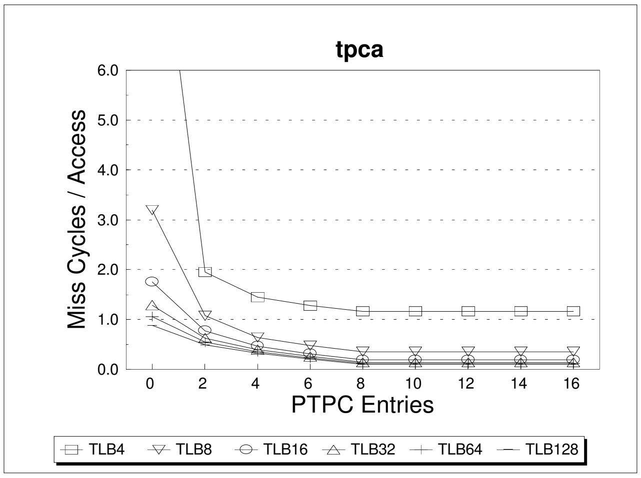

Figure 12: PTPC vs. TLB size tradeoffs for tpca.

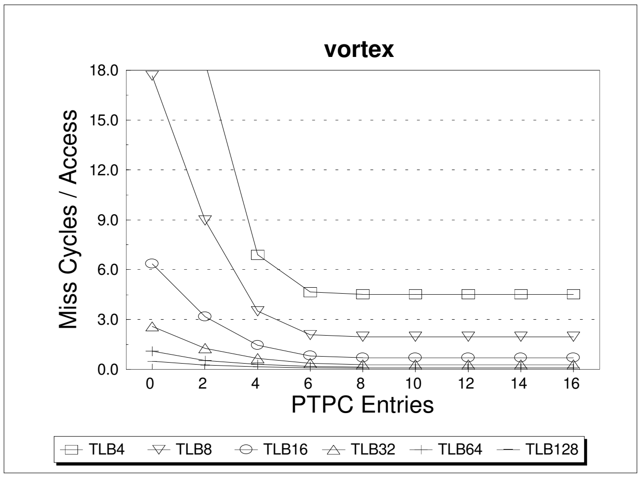

Figure 13: PTPC vs. TLB size tradeoffs for vortex.

et al. [13] also promote the use of virtually addressed caches as a way to ease TLB performance requirements. Systems with virtual caches are rare since the need to eliminate virtual synonyms in software complicates the operating system and virtual tags make shared memory coherence much more difficult and expensive.

Bala *et al.* [1] have developed an interesting software handler that caches and prefetches PTEs and PTPs in software. They achieve significant reductions (almost 50%) in the amount of time spent handling page directory misses for a group of operating system intensive applications. Their work is oriented towards decreasing miss handling time for page directories because their particular experimental setup (MIPS R3000 under Mach) generates many of these faults and handles them very slowly. While slower than hardware alternatives, their solution represents a very good software only alternative. In their system (and others based on MIPS processors), the lower 8 entries of the MIPS R3000 TLB are used to cache certain types of PTPs. It does not fill the TLB hardware, it caches to reduce cascading TLB faults in the software miss handler.

Sub-block TLBs [9] have been suggested as an enhancement to TLB design that shares the comparators used for address matching among several TLB entries, similar to what is done with cache blocks. This amortizes the expensive part of the search, both in time and hardware, but introduces dependencies among the related entries which can constrain their set of usable mappings. Complete sub-blocks allow small pages to be used without significant operating system modifications, but require large amounts of hardware. Partial sub-blocks provide better performance than pure superpages because they allow more flexible mappings to be used but require almost as much operating system support.

Inverted page tables provide an alternative to forward mapped page table designs by using an inverse mapping from physical memory to virtual memory rather than a virtual to physical mapping. The amount of memory required for virtual address translation is then proportional to the size of physical memory rather than virtual memory. Hashed page tables [4] improve on the basic inverse page table mechanism by using the inverse mapping as a convenient method of page table lookup rather than as a fixed page table structure. Like PTPCs, hashed page tables can greatly reduce the number of memory accesses needed to service a TLB miss. The biggest constraint of these designs is that address aliasing is limited. Inverse page tables cannot have virtual address aliases. Hashed page tables support aliases, but with decreasing efficiency as the number of aliases increases. Also, some per page information such as dirty bits cannot be accurately maintained with aliased pages, requiring special operating system support. As suggested in [4], a hashed data structure may be useful in speeding up software TLB miss handling with traditional page tables.

8 Conclusion

This paper introduces the use of page table caches as an architectural extension of the MMU in order to reduce TLB miss handling times. PTPCs significantly reduce the number of memory accesses needed to handle a TLB miss and makes it very simple to add hardware support for TLB refills, eliminating instruction and interrupt processing overheads. With a reduced TLB miss penalty there is less need to have extremely high TLB hit rates for good performance. PTPCs provide a low cost alternative to superpages that is both friendly to existing operating system management policies and data structures and backwards compatible with existing software.

We have examined the performance of PTPCs in various configurations and compared them with generic software and hardware handlers. Our results show that PTPCs can reduce the cost of miss handling by a factor of about 900% over a software handler and a by a factor of about 600% over a hardware handler when translating a full 64 bit address space. Our results also show that only a small PTPC (on the order of four to eight entries) is needed to have a large impact on performance. Finally, the combination of a medium sized TLB and a small PTPC is shown to be a very cost effective alternative to a large TLB when dealing with fixed sized pages.

9 Future Work

A major drawback of our work is that we can only simulate the user level behavior of a single process. In practice, the problem of poor TLB behavior is more of a problem when several contexts must communicate such as programs that rely heavily on operating system functions or programs that do a great deal of streaming I/O. The communication and context switching reduces the locality of reference in the data stream which reduces the effectiveness of TLBs. In this case, PTPCs are also useful because only a few PTPC entries are needed to map each context so the working sets of several contexts can easily mapped at once. The reduction in locality of reference in communications intensive workloads should give a system with PTPCs even more of an advantage when compared to larger TLBs.

Another drawback of our simulator is that we cannot currently simulate the effectiveness of non-blocking loads with PTPCs. Processors that implement non-blocking loads are able to overlap the latency of a memory access with the execution of other independent operations. Such a system will also be able to hide a portion of the cost of the memory access needed to fetch a PTE from memory after a PTPC hit. Any hardware TLB handling mechanism will benefit from non-blocking loads but a system with PTPCs should perform especially well since the amount of latency that needs to be hidden at any give time is relative small (about one memory access). We expect the relative performance of PTPCs to increase when non-blocking load costs are modeled.

We are currently modifying Mike Johnson's superscalar processor simulator ssim [5] to model PTPC costs in a dynamic instruction execution environment. In the near future, we will be investigating tracing alternatives that will allow us to simulate multi-process and operating system interaction behavior. Among the alternatives are trap-driven simulation [12] and other tracing tools such as ATOM running on Digital Alpha workstations.

References

[1] K. Bala, M. Frans Kaashoek, and W. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, pages 243–253, November 1994.

[2] J. Bradley Chen. Memory behavior for an X11 windows system. In *Proceedings of the 1994 Winter Usenix Conference*, pages 189–200, January 1994.

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

[4] J. Huck and J. Hays. Architectural support for translation table management in large address space machines. In *SIGARCH93*, pages 39–50, May 1993.

[5] M. Johnson. *Superscalar Processor Design*. Prentice Hall, 1991.

[6] D. Nagle, R. Uhlig, and T. Stanley. Design tradeoffs for software-managed TLBs. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 27–38, May 1993.

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

[8] M. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford, November 1991.

[9] 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, October 1994.

[10] M. Talluri, M. Hill, and Y. Khalidi. A new page table for 64-bit address spaces. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, October 1995.

[11] 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.

[12] R. Uhlig, D. Nagle, T. Mudge, and S. Sechrest. Tapeworm ii, a new method for measuring OS effects on memory architecture performance. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 132–144, October 1994.

[13] D. A. Wood, S. J. Eggers, G. Gibson, M. Hill, J. Pendleton, S.A. Ritchie, R. H. Katz, and D. A. Patterson. An in-cache address translation mechanism. In *IEEE Computer Society*, pages 358–365, June 1986.