Level Two Translation Lookaside Buffers

Mark D. Callaghan Mohammed M. Hoque Eric Rotenberg

March 22, 1995

Abstract

The Translation Lookaside Buffer (TLB) is a key component of the memory hierarchy in that it hides the overhead of maintaining virtual memory. However, certain trends in computer design are increasing the demand on the TLB. The reach of current TLBs (amount of memory that the TLB maps) is not sufficient to support this demand. The consequence is that an unacceptable fraction of total execution time will be spent servicing TLB misses. Cycle time and area constraints prohibit the brute-force solution of scaling the size of the TLB with greater demand.

We begin with an analysis of the technological and functional trends which are creating the need for greater TLB reach. A timing model is then used to quantitatively examine cycle time constraints on the TLB, which results in a strong case for small, simple level one (L1) TLBs. Simulations are performed to study the effectiveness of level two (L2) TLBs as a means of increasing overall TLB reach. Both software L2 TLBs (located in physical memory) and hardware L2 TLBs (located on the processor chip) are considered. Further, some simulations using partial subblock TLBs are performed for comparison purposes. We claim that the L2 TLB can provide a significant performance improvement even if a partial subblock L1 TLB is used. There are two cases in which a partial subblock L1 TLB may not provide enough reach: high rates of multiprogramming, and memory intensive applications with irregular data access patterns. These cases may cause the performance of the partial subblock L1 TLB to degrade rapidly.

1 Introduction

The TLB reach problem can be attributed to both *technological* and *functional* trends in computer systems. The technological trends are:

- Raw processor speed has increased by a rate of 50% per year since 1987, while DRAM performance is improving by just 7% per year [11]. It is expected that this widening gap will soon rival the Federal Deficit. As a consequence, TLB misses are becoming relatively more expensive.

- The current trend in DRAM memory capacity is a four-fold increase every three years [11]. DRAM manufacturers are developing denser memories because there is demand for more capacity: applications will take advantage of more physical memory. (If this were not the case, DRAM manufacturers would begin concentrating on speeding up memory.) It is no coincidence that the latest wave of processors implement 64-bit datapaths —the address space is growing. Larger address spaces will make it imperative to find ways of increasing TLB reach.

The first factor—the widening CPU-memory speed gap—requires a TLB which misses less often to overcome the larger miss penalty. The second factor—larger address spaces—requires that the TLB map more memory to maintain the miss rate. These two factors work in concert to compound the TLB reach problem.

TLBs with superpage support, complete subblocks, or partial subblocks offset the performance penalty due to these trends, since each method improves the ability of the TLB to map a process's address space [1].

However, there are other functional factors that increase the number of contexts that a TLB must contain, and solutions aimed at the L1 TLB may be insufficient in these cases. The first functional trend is the distribution of operating systems across several contexts due to the use of microkernels [20]. This will increase the minimum number of contexts that a TLB must contain to provide the same level of functionality that a monolithic kernel provides. For example, a file system call can involve several processes (the file system server and the pager) in a microkernel based OS. The second functional trend is the distribution of system services across several machines to increase sharing. File systems are no longer restricted to a single machine [24]. In this case, a file system read will invoke a file system process on the local machine, and may also require network communication. The effect is a larger imprint in the TLB from the kernel for basic OS services. The third functional trend is the increase in the number of context switches due to communication intensive applications. Inexpensive and portable parallelism is being sought by utilizing workstations as the nodes of parallel machines [23] [22].

Increases in communication, the number of user processes that the average user requires, and the number of system processes required by the operating system increase both the rate of context switches and the number of contexts that a TLB must concurrently map. We hypothesize that the factor which ultimately determines TLB performance under such conditions is the number of *independent* TLB entries. Partial subblocking gives the illusion of more TLB entries, but subblocks must be in a contiguous region and are therefore not independent. An L2 TLB, on the other hand, provides a large number of independent entries.

Implementing an L2 TLB does not preclude the use of partial subblocking or any other scheme for improving L1 TLB reach. In this case, the non-traditional L1 TLB will reduce traffic to the L2 TLB, resulting in even better performance. The L2 TLB simply backs the L1 TLB when the number of concurrent contexts is overwhelming.

Previous work relevant to increasing TLB reach is looked at in Section 2. Experimental methods are covered in Section 3; including the simulation environment, workloads, timing and area models, methods for deriving miss penalties, and metrics. An analysis of L1 TLB timing constraints is presented in Section 4. Area cost considerations are described in Section 5. Section 6 describes the performance data that has been generated for L2 TLBs. Finally, Section 7 summarizes the results and discusses additional work that is required to support our hypothesis.

2 Previous Work

Talluri and Hill [1] take an extensive look at the problem of TLB reach and contrast available alternatives, including increasing the size of the L1 TLB, increasing the page size, complete subblocking, and using superpages. A fifth alternative called *partial subblocking* is introduced; this method is shown to peform better than medium-sized superpages with simpler operating system support, and nearly as well as complete subblocking without the prohibitive cost in chip area.

Another approach to the TLB problem is to tolerate L1 TLB misses but to reduce the penalty associated with TLB miss handling. Several papers within the last two years take this approach with some form of L2 TLB. DeLano et al [4] and Bala et al [2] both cached PTEs in a software L2 TLB. On an L1 TLB miss, this L2 TLB is searched first before a page table walk is initiated. An interesting result is that in addition to reducing miss handling time, an L2 TLB reduces the number of nested TLB misses which are apt to occur in hierarchical page table organizations (all but the top-most level of the page table hierarchy are located in virtual space, requiring accesses to the page tables to go through the TLB)[2]. Borkenhagen et al [15] describe a commercial processor implementation with a *hardware* L2 TLB (512 entry, 4-way associative), which has the advantage of only a 2-cycle penalty for an L2 hit.

Taylor et al [18] implemented a unique form of L1 TLB called the *TLB slice*. The cache in this implementation is physically indexed, so the virtual address must be translated fast. This goal is met with the small TLB slice, containing only 8 entries. As one might expect, the performance of the TLB slice is poor and is consequently backed by a larger secondary TLB.

In a *Hashed Page Table* organization [19], bits of the virtual address are hashed to index into a table which holds page table entries. This scheme has a significant advantage over hierarchical page tables in terms of the number of memory references required to translate. Conflicts due to hashing can be resolved by chaining into another table. The PowerPC architecture goes a step further by introducing Page Table Entry Groups, each of which contain 8 page table entries for collisions. In essence, this hashed page table looks like an 8-way set-associative L2 TLB.

3 Methodology

This section details the simulation environment used to obtain TLB miss counts, workloads measured, the analytical access time and area models, the derivation of miss penalties, and metrics.

3.1 Simulation Environment

A modified operating system kernel, Foxtrot [1], was used to count the number of TLB misses. Foxtrot allows TLBs to be simulated that are larger or smaller than the physical TLB of the host machine by trapping to a user handler on TLB misses. The speed of trap based simulation allows a large number of applications and configurations to be measured. Foxtrot also allows multi-application workloads to be measured.

Foxtrot was run with a variety of TLB configurations to obtain TLB miss data. The number of entries in the TLB was varied from 32 to 4096 and the associativity was varied from 4-way set-associative to fully-associative. We had planned on modifying the TLB miss handler to simulate L2 TLBs, but time constraints limited our use of Foxtrot to simulation. We made an assumption in order to judge the effectiveness of L2 TLBs since we did not directly simulate L2 TLBs. Our assumption was that the number of misses for an L2 TLB of a specific configuration would be close to the number of misses that an L1 TLB of the same configuration has. The L2 TLB is assumed to have significantly more entries in this case, and both TLBs use LRU replacement. Trace based simulation using the first one million references from each of the SPEC92 [25] benchmarks confirmed the validity of this assumption.

3.2 Workloads

The SPEC92 floating point benchmarks were used to study memory system performance. These benchmarks were used because of their acceptance by the benchmarking community. Additionally, the SPEC92 benchmarks have been extensively studied and memory system performance data has been published for the benchmarks running on several architectures. These benchmarks are scientific applications and are not necessarily representative of behavior that occurs in other environments, such as software development or database use. Most of the floating point benchmarks did not use large amounts of memory. We feel that this type of behavior is not indicative of all workloads for which performance is important.

One application from the NAS benchmark [26] suite, fftpde, was also used. This application had the largest working set, 58MB, of all the applications run. A workload more similar to a software development workload was also run. Several files from a research project [27] were compiled using g++ [28]. These files used the template features of C++ [29] and caused the compiler to have a much larger working set than the gcc benchmarks in the SPEC92 integer suite.

3.3 Analytical Timing Model

An analytical timing model [9] was used to generate all cache and TLB timing data. Slight modifications, having to do primarily with tag sizes, were made to the source code to support TLBs and virtual index caches. The model was run using a 4K page size, 64-bit virtual address, 32-bit physical address, cache bandwidth of 16 bytes/cycle, and process parameters for a 0.5 micron process.

The timing model calculates the delay through each circuit block of a conventional CMOS set-associative cache design:

- address decoder
- wordlines
- bitlines
- sense amplifiers
- comparators
- multiplexor drivers
- output drivers

CAM cells are not modeled and therefore the access times for fully-associative TLBs are not included in this paper.

A potential concern of the analytical model is accuracy. Due to the attention to detail and verification against SPICE circuit simulation at each logic block in the cache datapath, accuracy was not a concern for the cache timing model.

What may be of concern, however, is the generality of the model. The problem is that circuit performance is very implementation-dependent. The design space for even a 16-bit adder is very large, let alone a tlb-cache design. In the case of the cache timing model, certain basic assumptions were made about the structure of the SRAM arrays, the basic cell design, the sizes of drivers, transistor diffusion regions, and so on. However, if we assume that a given design style, organization, or optimization is applied in every circuit, then absolute numbers are not as important as relative numbers. This makes comparisons of access times, for example, possible.

3.4 Analytical Area Model

An area model [6] was used to determine the area cost of L1 TLBs as well as to consider the feasibility of hardware L2 TLBs. Since the area model was developed for caches, slight modifications had to be made so that it could be applied to TLBs (similar to the timing model modifications). Also, the area of the PLA control logic was not included for several reasons. First, area of control logic is a secondary effect when compared to dataflow and arrays. Second, area of random logic is not a number which can be easily arrived at. As with the timing model, a 4K page size, 64-bit virtual address, and 32-bit physical address were used.

3.5 Miss Penalties

The number of TLB misses is only part of the equation for calculating the TLB contribution to CPI (our primary metric, CPI, is described in the next section). In this section the penalties for accessing the software L2 TLB, hardware L2 TLB, and page tables are derived.

All penalties are derived from the 16 assumptions about the memory system listed in Figure 1. The factor of 20 disparity between processor and DRAM speed is typical of current technology (the Alpha 21064 has a 5 ns cycle time, DRAMs have a latency on the order of 100 ns [11]). Most of the other assumptions, such as fast bypass, split address and data buses, pipelined memory bus, critical word first, and castout buffers, can be categorized as typical performance optimizations. Item 7 determines the number of bus transfers required for a cache block. The dual, cache block interleaved memory banks allow for one bank to be accessed while the other recovers after a read access (destructive reads result in the cycle time being 20 ns longer than the access time). Assumptions 14 and 15 are based on the observation that the contents of the cache changes drastically between relatively infrequent TLB misses (infrequent relative to cache misses), so the probability of finding any PTEs in the cache on a TLB miss is low. Also note that the TLBs are managed by hardware.

1. 200 MHz processor clock rate (5 ns cycle time)
2. DRAM: 100 ns access time, 120 ns cycle time
3. memory bus clocked at half the processor rate (10 ns transfer time)
4. split address and data buses
5. pipelined memory bus
6. no bus arbitration
7. 8 byte data bus, 64 byte cache block (8 bus transfers for a block fill)
8. critical word first
9. fast bypass
10. castout buffers (for cache and TLB castouts)
11. assume no other types of interference: store queue, snooping, other
12. dual memory banks which are cache block interleaved
13. no L2 cache
14. probability that there are any L2 TLB entries in cache is zero
15. probability that there is any page table information in cache is zero
16. TLBs are hardware accessed and managed (management, such as replacing TLB entries, is outside of the access path and consequently does not affect penalty)

Software L2 TLB

Assoc	Cycles
1	29
2	29
4	31
8	35
16	43
32	61
64	97

Hardware L2 TLB: 4 cycles

3-Level Page Table: 83 cycles

Figure 1: Assumptions used for deriving miss penalties. Tables show the derived access times for the software L2 TLB, hardware L2 TLB, and 3-level page table.

Cycle-by-cycle timing diagrams for hardware accessed L2 TLBs and 3-level page tables have been generated based on the above assumptions and are shown in Appendix A. A few key points can be extracted from the timing diagrams. First, the number of cycles it takes to access the L2 TLB increases with associativity, due to the sequential searching of successive PTEs returned from memory. (Note: in deriving actual penalties for the software L2 TLB, it was assumed that on average A/2 entries are searched if the TLB is A-way associative.) Second, the number of cycles to walk a 3-level page table is the sum of three back-to-back memory accesses with some overhead. This number may be optimistic because it is assumed that all page table accesses hit in the TLB (recall that the bottom two levels of the page table hierarchy are in virtual space and must go through the TLB). The hardware L2 TLB is assumed to be on the processor chip and has a 2-cycle access time.

3.6 Performance Metric: CPI_{TLB}

The ideal performance measure is program execution time. However, it is rarely practical to measure execution time because it requires either running experiments on an existing machine or having a detailed simulator. Using an existing machine is impractical because the hardware is fixed, whereas the purpose of the experiment is to study many configurations. A cycle-by-cycle simulator is too complex to develop.

For this reason, many papers in the literature simply use miss rates as a metric. However, miss rates alone would be insufficient for this paper since the effect of an L2 TLB is to reduce the L1 TLB miss handling time. The TLB contribution to cycles-per-instruction is a metric which captures both miss rate and miss penalty:

$$CPI_{TLB} = \frac{L1misses}{N_i}\left[T_{L2} + \left(\frac{L2misses}{L1misses}\right)T_{PT}\right]$$

where N_i is the number of dynamic instructions, $L1misses$ and $L2misses$ are the number of misses in the L1 and L2 TLBs respectively, T_{L2} is the number of cycles to access the L2 TLB, and T_{PT} is the number of cycles to walk the page table. Instruction counts for the SPEC92 benchmarks were obtained from [5]. Note that the equation assumes the L2 TLB and page table are not searched in parallel, although this optimization could be used to hide at least some part of L2 TLB misses (assuming the memory system could handle the simultaneous requests).

CPI_{TLB} still is not a complete measure of performance. For one thing, it is an unknown fraction of total CPI. More seriously, however, is that it does not incorporate cycle time. In our experiments we assumed a fixed cycle time for all hardware configurations. The result is that the same miss penalties can be used for all configurations. However, in a real design increasing the size of the TLB may stretch the processor cycle time—for a fixed memory subsystem, the miss penalty *in cycles* would actually decrease. The bottom line is that CPI_{TLB} and processor cycle time are not independent. We assumed independence in order to make the derivation of miss penalties feasible.

4 TLB and Cache Access Times

The access time of the L1 TLB is an important consideration for processors with physically tagged caches, because the TLB must provide the physical page number to the cache tag comparator. This places the L1 TLB in the cache access path. Since cache access is typically at or near the top of the list of processor critical paths, there is the chance that the L1 TLB will impact cycle time. This is a highly undesirable scenario, since it would mean that the TLB—an artifact of virtual memory—is limiting the speed of *all* operations in the processor. This scenario can be circumvented by making the cache access a multi-cycle path, but this in turn impacts CPI and places more burden on the compiler by virtue of a longer load delay.

It is possible to implement a virtually tagged cache, thereby removing the TLB from the critical path. However, there are two major drawbacks to virtual tags (which may explain the prevalence of physically tagged caches):

1. Virtual tags are considerably larger than physical tags. For example, the PowerPC architecture specifies an 80-bit virtual address space, whereas the PowerPC 620 implementation has a 40-bit address bus [14]. The graph in Figure 2 shows the effect of using virtual tags versus using physical tags on cache cycle time, assuming a 64-bit virtual space and a 32-bit physical space. Area is also affected: the tag storage area for a 32KB, 4-way set-associative cache with a 16 byte block increases by a factor of 2.6.

2. The cache has to be flushed on context switches unless every cache block is tagged with a context identifier. Adding a context id for every tag again will affect access time and expend area. The cost of flushing the cache is two-fold. First, a restarted context will suffer repeated compulsory misses. Second, flushing the cache may take time due to casting out dirty blocks.

The analytical timing model allows us to quantify the impact of the TLB on cycle time. The interaction of the TLB and cache access is shown in Figure 3. Note the segments labelled "precharge". Accessing a memory composed of static RAM cells involves a precharge phase in addition to the actual access. Bitlines, comparators, and other elements of the array must be precharged before a new access can be initiated [9]. Figure 3 shows the scenario where the TLB has as much time possible to provide the physical page number to the cache comparator: the cache is precharged at the beginning of each cycle, whereas the TLB precharge is overlapped with logic after the cache comparators. The race between TLB and cache is plotted in Figure 4 for various sizes and associativities. Note that for even the smallest TLB (32 entry, 2-way associative), the TLB is in most cases late in providing the physical address to the cache, and by a significant margin. This data supports the case for having a small TLB provided there is a way to recover from the lack of sufficient reach.

It should be noted that as an optimization, all possible physical page numbers can be sent to the cache compare logic before it is known which physical page number is correct. The comparisons which are valid can be selected after the fact. This collapses 4 levels of logic blocks into 3 levels, shown in Figure 5. This isn't for free: the number of cache tag comparators is increased by a factor of TLB associativity. This optimization may or may not take the TLB out of the critical path, but regardless it does add a 2:1 multiplexor directly in the cache tag path.

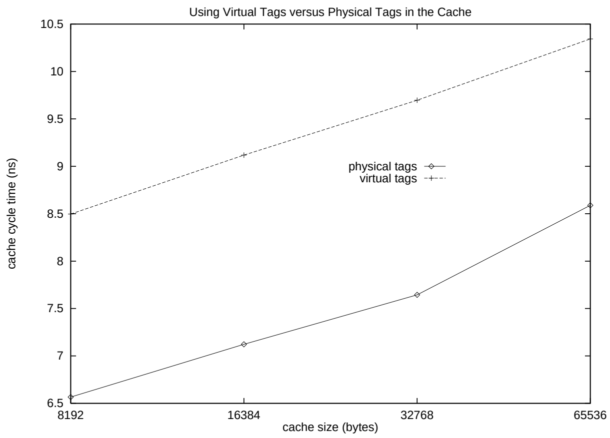


Figure 2: This graph shows a substantial cycle time increase—about 2 ns—if the cache uses virtual tags instead of physical tags. The data is for 4-way set-associative caches with a block size of 16 bytes.

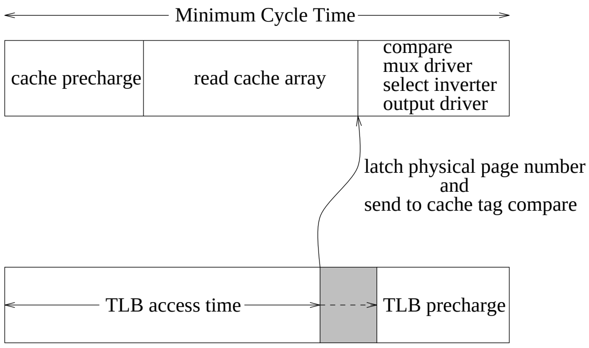


Figure 3: The L1 TLB is in the cache access path for physically tagged caches.

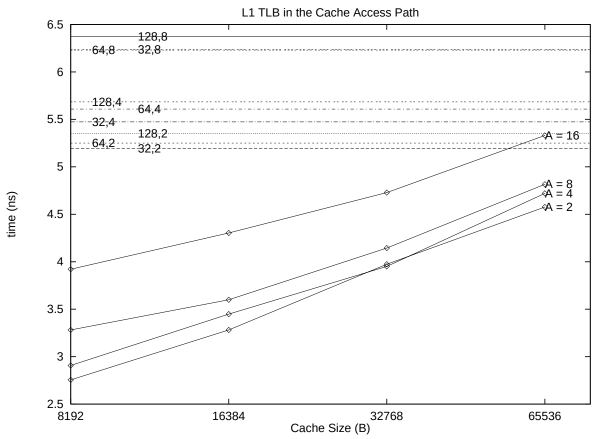


Figure 4: This graph shows the time it takes for the TLB to produce the physical page number needed by the cache tag compare logic, and the time it takes for the cache to get to that point. Each horizontal line corresponds to a TLB configuration—the notation (E,A) indicates the number of TLB entries followed by the associativity. The solid lines correspond to cache associativities of 2, 4, 8, and 16; cache size is varied along the x-axis; cache block size is 16 bytes.

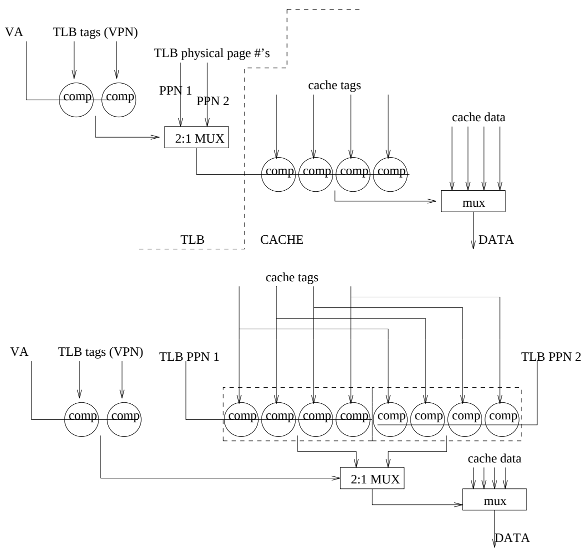


Figure 5: Merging parts of the TLB and cache access may help the critical path.

5 TLB Area Considerations

The purpose of this section is to determine at what point the area of the L1 TLB becomes a concern, as well as to determine the feasibility of implementing a *hardware* L2 TLB. All TLB areas are normalized to the area of a 32K, 4-way set-associative cache. (Normalizing to a typical on-chip cache gives a better feel of the area cost than absolute rbe's.)

5.1 L1 TLB Area

Figure 6 shows the relative area of TLBs ranging from 32 to 512 entries in size. L1 TLBs fall in this range. The conclusion we can draw from this graph is that for TLB sizes at which access time becomes a concern (64 to 256), the area cost relative to the chip area devoted to the memory interface is minor. In other words, TLB access time becomes a concern long before area is a factor.

5.2 L2 TLB Area

Figure 7 shows the relative area of L2 TLBs ranging from 512 to 4K entries. The areas range from about 20% to 120% of the reference cache area. Therefore, even for medium sized L2 TLBs the area cost is fairly significant: 1K entries takes up one-third of the reference cache area, and 2K entries takes up two-thirds of the reference cache area.

The decision to include an L2 TLB in hardware depends on the performance benefit and on technology trends. VLSI trends will almost certainly make hardware L2 TLBs feasible. It becomes a matter of deciding what function to devote extra transistors to. Amdahl's law dictates that hardware be spent on function which will yield the greatest benefit. The trends discussed in Section 1 indicate that translation could become the performance bottleneck. In that case, a designer may be better off choosing to add an L2 TLB rather than adding a cache associativity class, for example.

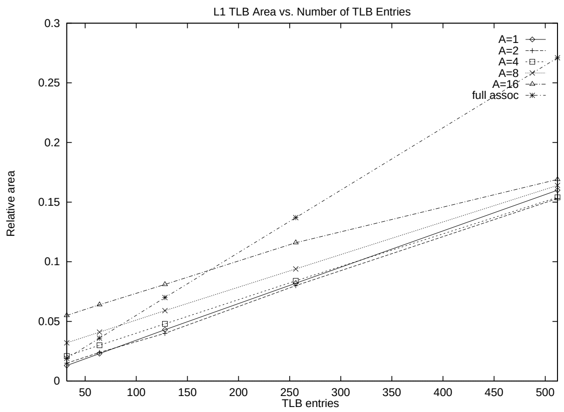


Figure 6: This graph shows the area of L1 TLBs relative to a 32K, 4-way set-associative cache.

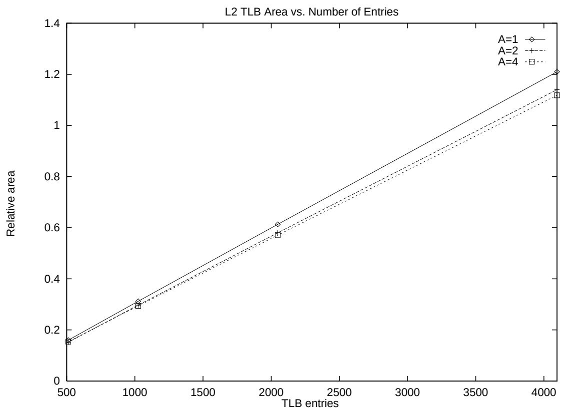


Figure 7: This graph shows the area of L2 TLBs relative to a 32K, 4-way set-associative cache.

6 Performance

We present performance results for four types of TLB changes:

- Larger L1 TLB
- Partial Subblock TLB
- Software L2 TLB
- Hardware L2 TLB

The first option is unacceptable due to size and speed constraints, but can provide a best case figure against which the performance of the practical solutions can be compared. The second option has been extensively studied [1]. We include measurements for subblock TLBs to show where such TLBs are successful and where the additional reach provided by a subblock TLB is insufficient. While there are many factors in which hardware and software L2 TLBs differ, the only one that is considered here is the time to service an L1 TLB miss.

The results are presented for three types of applications:

- SPEC92 floating point
- Memory intensive, single application
- Memory intensive, multi-application

The number of TLB entries determines the TLB reach. We claim that L2 TLBs are important since they effectively provide more TLB entries by reducing the TLB miss penalty. A partial subblock TLB provides the illusion of a large number of independent entries when memory access patterns are amenable to page reservation and subblocking. However, there are two classes of workloads that may break partial subblocking. The first case is when there are multiple address spaces contending for TLB entries. The second case is when an application uses a large amount of memory, but does not use large contiguous regions of memory like arrays or matrices. L2 TLBs should provide the best performance in these cases.

6.1 SPEC92

The graph in Figure 8 shows the number of TLB misses for the floating point suite. The table in Figure 9 shows the number of TLB misses for a partial subblock TLB. The graph is not normalized for the number of instructions that is executed by each application. The point at which each curve becomes horizontal approaches the number of compulsory misses. The floating point benchmarks can be divided into four equivalence groups according to the number of TLB entries at which they approach this limit. 015.doduc, 048.ora, 094.fpppp, 034.mdljdp2, 056.ear and 77.mdljsp2 require a 64 entry fully associative TLB. 052.alvinn and 090.hydro2d require a 256 entry TLB. 093.nasa7, 047.tomcatv, 078.swm256, 013.spice2g6 and 089.su2cor require a 1024 entry TLB. 039.wave5 requires a TLB with more than 4096 entries.

Two conclusions can be drawn from the graph. First, a 1024 entry TLB would remove most of the non-compulsory misses for 13 of the 14 benchmarks. Second, the number of compulsory misses is much less than the number of conflict and capacity misses for realistic L1 TLB configurations. The large number of capacity and conflict misses is an opportunity for improved performance that requires an L2 TLB. Removing this much larger number of conflict and capacity misses may drastically improve performance.

The utility of the SPEC92 benchmarks for testing memory system behavior is questionable. The majority of the benchmarks do not stress the memory system since their reach is small. Using such benchmarks may not reflect the actual workloads that are used on many machines.

6.1.1 Larger L1 TLB

The graphs in Figure 10 show CPI_{TLB} for L1 TLBs with a variety of configurations. The different levels of associativity can be grouped into three levels of similar performance when the TLB has less than 256 entries, two levels of performance when the TLB has 256 entries, and one level of performance when the TLB has 512 or more entries.

Evident from the graphs is the limited value of associativity greater than 16-way. Figures 11 and 12 also display the limited value of high levels of associativity.

6.1.2 Partial Subblock TLB

Large L1 TLBs are not achievable, but Figure 9 shows that partial subblock TLBs also remove most non-compulsory misses. For the single application benchmarks, a partial subblock TLB negates the need for an L2 TLB.

6.1.3 Hardware L2 TLB

If a partial subblock TLB were not used, then an L2 TLB would provide a performance improvement. The reason for the benefit of an L2 TLB can be seen in Figure 8. A TLB with a large number of entries removes most non-compulsory misses. An L2 TLB could be larger than the L1 because it would have neither the timing constraints nor the area constraints that an L1 TLB has. The graph in Figure 11 shows the peformance of an L2 TLB in hardware.

6.1.4 Software L2 TLB

The performance benefit of software L2 TLBs can also be seen from Figure 10. Associativity becomes less important as the number of entries in a TLB is increased. This implies that a software L2 TLB would not suffer a large penalty due to the number of comparisons that a high degree of associativity would require. The graph in Figure 12 shows the peformance of an L2 TLB in software.

6.2 Memory Intensive, Single Application

An application, fftpde from the NAS benchmark suite, was run with a working set as large as 58M to determine the performance of the various TLB enhancements. A partial subblock TLB should perform well since this application used very large data structures that could be subblocked. This would enable the 64-entry partial subblock factor 16 TLB to approach the performance of a (64 * 16) entry TLB. Table 1 displays the performance data. The partial subblock TLB performed significantly better than TLBs with low levels of associativity and large number of entries. The partial subblock TLB appears to have been able to provide more entries via subblocking.

Another memory intensive application was run to test the effectiveness of partial subblocking when less regularly accessed data structures were used. A C++ source file was compiled using

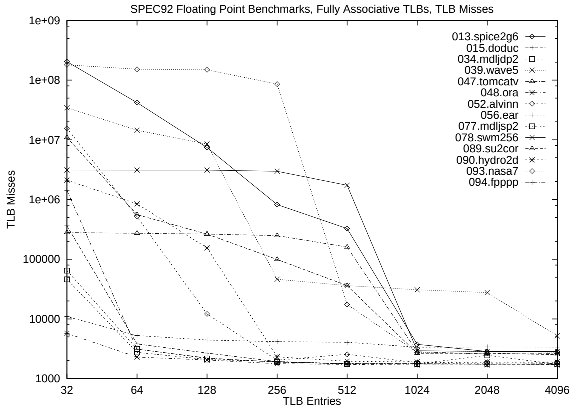


Figure 8: The number of TLB misses for the SPEC92 floating point benchmarks with fully associative TLBs.

Benchmark	Partial Subblock TLB Misses [1]
nasa7	9000
wave5	33000
spice	5000

Figure 9: The number of TLB misses with a 64-entry fully-associative partial subblock 16 L1 TLB.

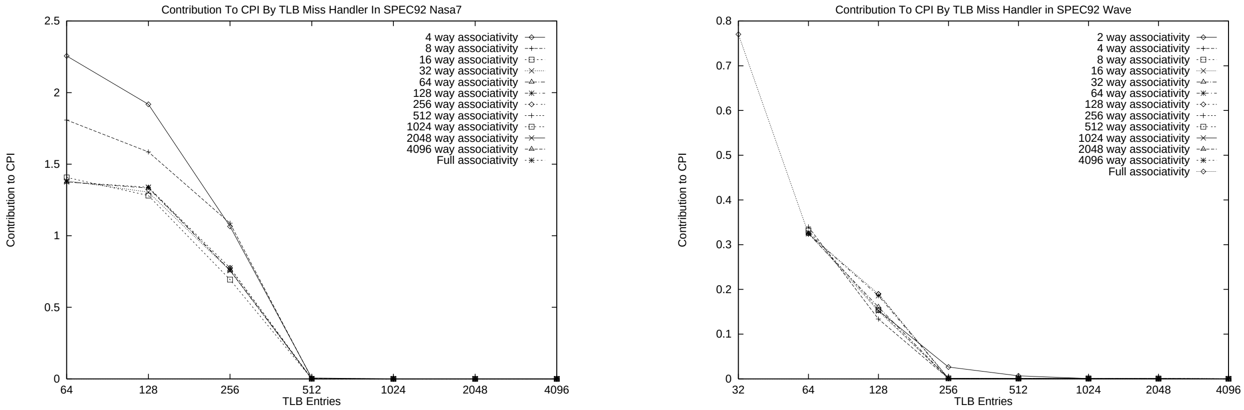


Figure 10: These graphs show the performance for L1 TLBs. CPI_{TLB} is plotted for nasa7 and wave5 as the L1 TLB configuration varies.

version 2.6.0 of g++. In this case, large TLBs performed much better than partial subblock TLBs. Table 1 displays the performance data.

6.3 Memory Intensive, Multi-application

A set of applications was run to simulate a software development process. The execution of the applications was staggered, but their run times overlapped. Four files were compiled, one scientific code (fftpde) was run, and one functional code, a Scheme run-time system performing a Fast Fourier Transfrom, was run. The total memory used by all of the applications at any given time was less than 70M, which prevented the applications from paging. C++ source code from a research project was used as input for the compiler. The source code files were large and caused the compiler to have a large working set size. The SPEC92 gcc benchmark compiles C code and has a significantly smaller working set.

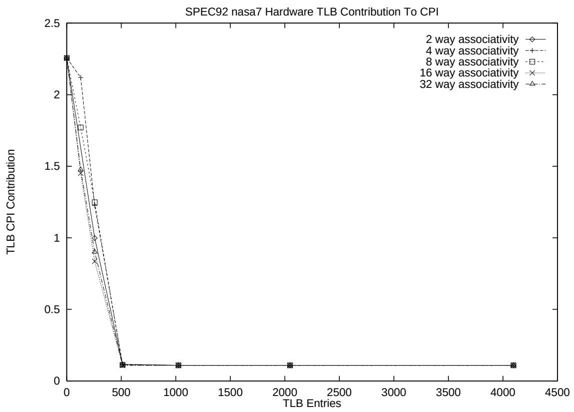


Figure 11: CPI_{TLB} for hardware L2 TLBs. The L1 TLB is 64-entry 4-way set-associative. The benchmark is nasa7.

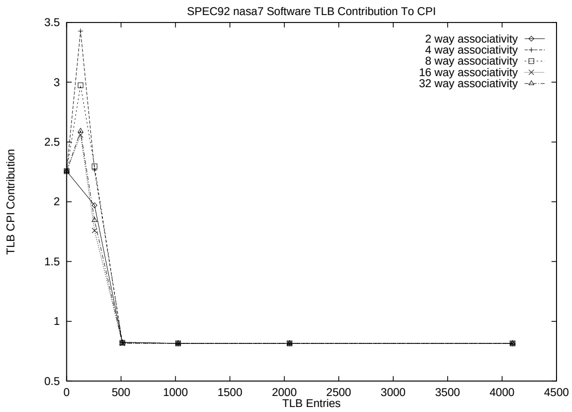


Figure 12: CPI_{TLB} for software L2 TLBs. The L1 TLB is 64-entry 4-way set-associative. The benchmark is nasa7.

The programs in this test did not use significantly more memory than one copy of the memory intensive version of fftpde. However, relative to L2 TLBs, the partial subblock TLB performed significantly worse. Table 2 displays the performance data. Competition for TLB entries should limit the effectiveness of subblocking.

While we claim that multiprogramming workloads can limit the effectiveness of a partial subblock TLB, this example cannot serve as evidence. The applications that contributed the majority of TLB misses also had large numbers of TLB misses when they were run alone. However, the applications do serve as valid examples of applications for which the performance of partial subblock TLBs can be improved.

Table 1: Memory Intensive applications (NAS fftpde, g++). The partial subblock TLB (subblock factor 16) performs better than TLBs with less than 1024 entries and low associativity.

Benchmark	Subblock TLB?	TLB Entries	Associativity	TLB Misses
fftpde	Yes	64	64	44858462
fftpde	No	64	2	178502074
fftpde	No	128	2	105959350
fftpde	No	256	2	101536685
fftpde	No	512	2	99325406
fftpde	No	1024	2	15667432
g++	Yes	64	64	15066404
g++	No	1024	4	739425

Table 2: TLB Performance for a multi-application workload. The partial subblock TLB has a subbblock factor of 16.

Benchmark	Subblock TLB?	TLB Entries	Associativity	TLB Misses
Multi	Yes	64	64	69669408
Multi	No	1024	2	19780038
Multi	No	1024	4	12560018
Multi	No	1024	8	4991340

7 Conclusions

- We have shown that the L1 TLB is likely to impact cycle time even for smaller configurations. This stresses the importance of keeping the L1 TLB small and simple.
- Timing of the L1 TLB becomes critical before its area is even a concern.
- An L2 TLB on chip takes up a significant fraction of the total area devoted to the memory interface. However, this cost may be warranted in the future if translation becomes a bottleneck.
- The performance benefit of associativity in TLBs decreases as the number of entries increases.
- For the workloads used, 16-way set-associative TLBs performed as well as fully-associative TLBs.
- The determining factor for TLB performance is the number of entries provided. We claim that there are two cases in which a partial subblock TLB will degrade to the performance of a non-subblock TLB.
  - A memory intensive application with irregular data access patterns.
  - Multiple memory intensive applications running concurrently (not validated).
- Combining a partial subblock L1 TLB with a software L2 TLB brings together the best of both alternatives. The partial subblock L1 will minimize CPI_{TLB} for workloads which subblock well, and the L2 TLB can provide the extra reach when the realizable subblocking factor is less than ideal.

If work were continued on the project, several areas would need additional attention. First, the Foxtrot TLB miss handler should have code added to model L2 TLBs. Second, the behavior of contention for TLB entries needs to be measured and characterized in greater detail. Third, non-SPEC workloads need to be run on Foxtrot. Fourth, multiprogramming workloads need to be thoroughly tested. These include database applications and memory intensive software development workloads.

References

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

[2] Kavita Bala, M. Frans Kaashoek and William E. Weihl. Software Prefetching and Caching for Translation Lookaside Buffers. In *Proceedings of the First Symposium on Operating System Design and Implementation*, November 1994.

[3] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge and Richard Brown. Design Tradeoffs for Software-Managed TLBs. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 358-369, Chicago, IL, April 1994.

[4] Eric DeLano, Will Walker, Jeff Yetter and Mark Forsyth. A High Speed Superscalar PA-RISC Processor. In *Proceedings of the IEEE Computer Society International Computer Conference*, 1992.

[5] Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos and Alan Jay Smith. Cache Performance of the SPEC92 Benchmark Suite. *IEEE Micro*, To Appear.

[6] Johannes M. Mulder, Nhon T. Quach and Michael J. Flynn. An Area Model for On-Chip Memories and its Application. *IEEE Journal of Solid-State Circuits*, February 1991.

[7] Tomohisa Wada, Suresh Rajan and Steven A. Przybylski. An Analytical Access Time Model for On-Chip Cache Memories. *IEEE Journal of Solid-State Circuits*, August 1992.

[8] Norman P. Jouppi and Steven J.E. Wilton. Tradeoffs in Two-Level On-Chip Caching. In *Proceedings of the 21st International Symposium on Computer Architecture*, Chicago, IL, April 1994.

[9] Steven J.E. Wilton and Norman P. Jouppi. An Enhanced Access and Cycle Time Model for On-Chip Caches. DEC Western Research Lab, Tech Report 93/5, July 1994.

[10] Steven A. Przybylski. *Cache and Memory Hierarchy Design: a Performance-Directed Approach*, Morgan Kaufmann Publishers, 1990.

[11] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, 1994.

[12] George Paap, Ed Silha. PowerPC: A Performance Architecture. In *Proceedings of the IEEE Computer Society International Computer Conference*, 1993.

[13] Michael S. Allen, Michael C. Becker. Multiprocessing Aspects of the PowerPC 601. In *Proceedings of the IEEE Computer Society International Computer Conference*, 1993.

[14] Tom Thompson, Bob Ryan. PowerPC 620 Soars. *BYTE*, November 1994.

[15] John M. Borkenhagen, Glen H. Handlogten, John D. Irish, Sheldon B. Levenstein. AS/400 64-bit PowerPC-Compatible Processor Implementation. Submitted to *ICCD*.

[16] Mark D. Hill. A Case for Direct-Mapped Caches. IEEE Computer. December, 1988.

[17] J. Bradley Chen, Anita Borg, and Norman P. Jouppi. A Simulation Based Study of TLB Performance. In *The 19th Annual International Symposium on Computer Architecture.* May, 1992.

[18] George Taylor, Peter Davies and Michael Farmwald. The TLB Slice – A Low-Cost High-Speed Address Translation Mechanism. REFERENCE INCOMPLETE.

[19] Jerry Huck and Jim Hays. Architectural Support for Translation Table Management in Large Address Space Machines. In *Proceedings of the IEEE Annual Symposium on Computer Architecture*, 1993.

[20] Mike Rozier et al. Overview of the CHORUS distributed operating system. In *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures.* USENIX, April 1992.

[21] David R. Cheriton. The V distributed system. In *Communications of the ACM*, 31(3):314-333, March 1988.

[22] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam, *PVM 3 User's Guide and Reference Manual*, Oak Ridge National Laboratory, May 1993.

[23] K. Castagnera, D. Cheng, R. Fatoohi, E. Hook, B. Kramer, C. Manning, J. Musch, C. Niggley, W. Saphir, D. Sheppard, M. Smith, I. Stockdale, S. Welch, R. Williams and D. Yip. NAS Experiences with a Prototype Cluster of Workstations. *Supercomputing '94 Proceedings*, November 1994.

[24] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51-61, February 1988.

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

[26] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS Parallel Benchmarks. Report RNR-91-002 Revision 2, Ames Research Center, August 1991.

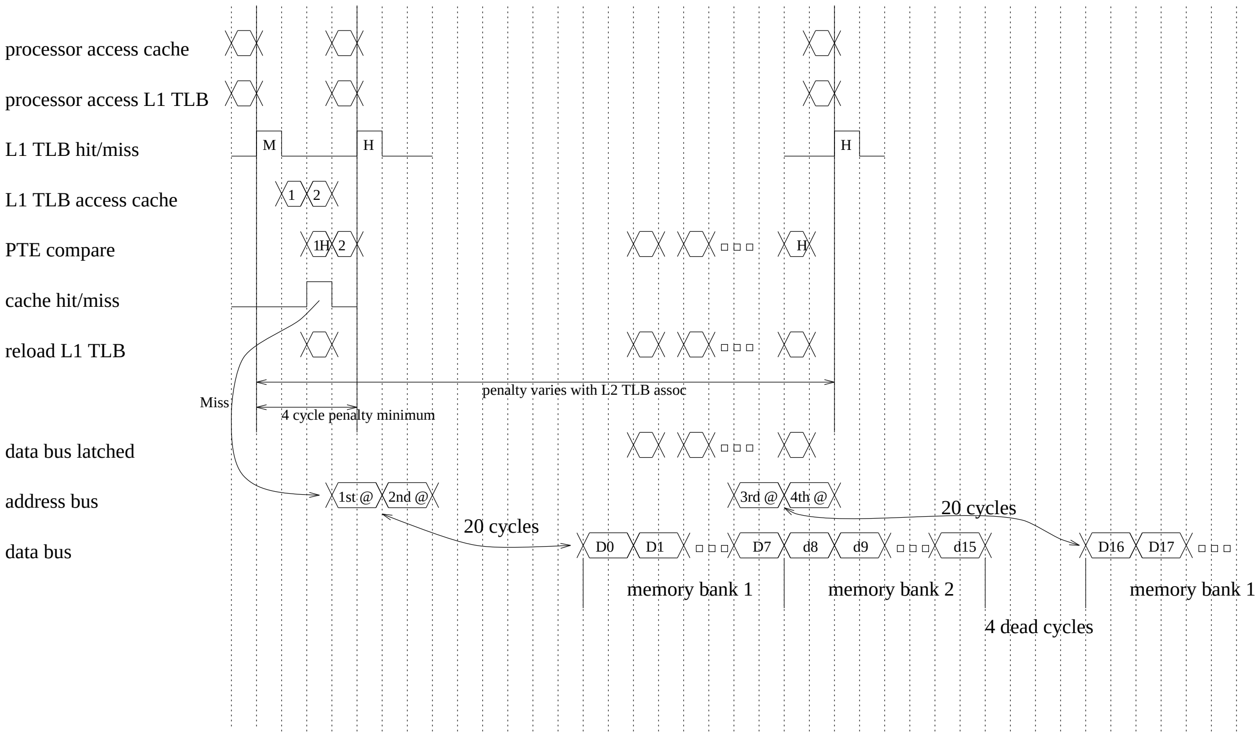
[27] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tools, Submitted for publication.

[28] Online manual and reference for g++. December 1994.

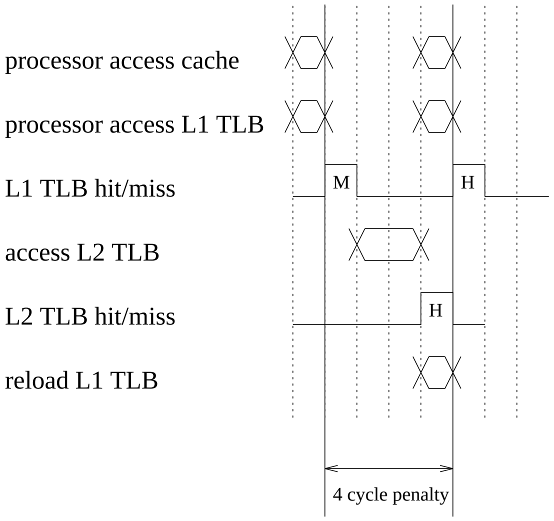
[29] Stanley B. Lippman. *C++ Primer, Second Edition*, Addison-Wesley Publishing Company, New York, 1994.

A Miss Penalty Timing Diagrams

A.1 Accessing the Software L2 TLB



A.2 Accessing the Hardware L2 TLB



A.3 Accessing the 3-Level Page Table