To appear in the *Nineteenth International Symposium on Computer Architecture*, June 1992.

Tradeoffs in Supporting Two Page Sizes*

Madhusudhan Talluri,† Shing Kong,‡ Mark D. Hill,† David A. Patterson††

†Computer Sciences Dept.
University of Wisconsin
Madison, Wisconsin 53706
talluri@cs.wisc.edu

‡Sun Microsystems Laboratories, Inc.
Mail Stop: 29-225
2550 Garcia Ave.
Mountain View, CA 94043-1100

††CS Division, UC Berkeley
571 Evans Hall
Berkeley, CA 94720

ABSTRACT

As computer system main memories get larger and processor cycles-per-instruction (CPIs) get smaller, the time spent in handling translation lookaside buffer (TLB) misses could become a performance bottleneck. We explore relieving this bottleneck by (a) increasing the page size and (b) supporting two page sizes.

We discuss how to build a TLB to support two page sizes and examine both alternatives experimentally with a dozen uniprogrammed, user-mode traces for the SPARC architecture. Our results show that increasing the page size to 32KB causes both a significant increase in average working set size (e.g., 60%) and a significant reduction in the TLB's contribution to CPI, CPI_{TLB}, (namely a factor of eight) compared to using 4KB pages. Results for using two page sizes, 4KB and 32KB pages, on the other hand, show a small increase in working set size (about 10%) and variable decrease in CPI_{TLB}, (from negligible to as good as found with the 32KB page size). CPI_{TLB} when using two page sizes is consistently better for fully associative TLBs than for set-associative ones.

Our results are preliminary, however, since (a) our traces do not include multiprogramming or operating system behavior, and (b) our page-size assignment policy may not reflect a real operating system's policy.

Keywords: Address translation, page size, translation lookaside buffer, virtual memory, working set size

* Talluri is supported in part by a summer internship at Sun Microsystems Laboratories, Inc. and by National Science Foundation Award (MIPS-8957278); Kong is supported by Sun Microsystems Laboratories, Inc.; Hill is supported in part by a National Science Foundation Presidential Young Investigator Award (MIPS-8957278) with matching funds from A.T.& T. Bell Laboratories, Cray Research Foundation and Digital Equipment Corporation; Patterson is supported in part by DARPA/NASA Ames Research Center, Grant Number: NAG2-591.

1. Introduction

A *translation lookaside buffer (TLB)* is a fast buffer containing recently used virtual-to-physical address translations [ClE85, HeP90, SaB81, Smi82]. Most computers that support paged virtual memory [Den70] use TLBs to reduce average address translation time.

Ten years ago, TLB miss handling was responsible for only a small fraction of a machine's cycles-per-instruction (CPI). A TLB could map a substantial fraction of main memory (e.g., 0.5MB), machines had large CPIs (e.g., 10 cycles), and programs had small working sets. For example, Clark and Emer [ClE85] report that the VAX-11/780 loses only 5% of its performance to TLB misses. Wood *et al.* [WEG86] report TLB miss rates to be around 0.03 - 3% for some machines built in the early 1980s.

However, technological and architectural trends have led to increasing main memory sizes, decreasing CPIs, and programs with larger working sets. Today's workstations can have memories larger than 32MB and an average CPI of two cycles or less. In a few years, we expect main memories of 256MB and CPIs of 0.5 cycles to be common. The larger memories are needed to support programs with larger working sets or to keep resident the working sets of multiple programs. Thus, TLB miss handling may become a performance bottleneck unless TLBs miss less often despite having to map larger working sets.

One way of improving TLB performance is to make the TLB hold more entries. The extent to which the TLB size can be increased depends on whether the level-one cache uses physical tags. If it uses physical tags, then TLB access must complete before the cache access does (even if the cache uses virtual index as in the MIPS R4000). Furthermore, for superscalar machines that may do multiple memory references per cycle, the TLB may need to be multi-ported. Therefore, if the TLB gets too large, it will adversely affect the latency of every memory reference.

If the level-one cache uses virtual tags, it is much more straightforward to build a large TLB, since it is accessed only on level-one cache misses. To date, few computer designers have used virtually-tagged caches due to concerns about the complexity of handling synonyms (i.e., multiple virtual addresses that map to the same physical address), cache flushes on context switches and multiprocessor issues.

A second method for improving TLB performance is to make the page size larger. The advantages are that the TLB maps more memory "for free", the size of operating system data structures decreases, and disk paging is more efficient (since the delay of disk head movement is amortized over more data transferred). There are, however, several reasons why typical page sizes have not increased. First, page sizes are much harder to change than cache block sizes. Page size is an architectural parameter, not just an implementation issue, as it affects memory management by the operating system. Thus, established architectures like the IBM 370 and DEC VAX-11 still use their original page size. Second, as we will show, larger pages result in larger working sets due to internal fragmentation [Den70], i.e., memory wasted due to the page size being larger than what the program needs. Third, the protection granularity becomes coarser. Appel and Li [ApL91] describe some applications that would benefit from smaller pages.

A third method for improving TLB performance is to use two page sizes and to require page sizes to be powers of two and pages to be aligned (i.e., a page of size B must be placed in virtual and physical memory at an address that is a multiple of B). If the large page size can be judiciously used, this method can make the TLB map more memory without a significant impact on working set size and no effect on minimum protection granularity.

However, supporting multiple page sizes has several disadvantages. First, like a larger page size, this change impacts a computer's architecture. Second, new TLB implementations must be built to handle multiple page sizes, which is not straightforward (Section 2). Third, software must select a proper page-size assignment policy to take advantage of the larger pages. Fourth, memory management and page replacement policies must accommodate multiple page sizes. Fifth, external fragmentation is now possible, which does not exist with a single page size. External fragmentation is waste due to the page size being larger than a contiguous region of available memory.

Nevertheless, supporting two page sizes is much simpler than supporting Multics-style segments [Org72] or many page sizes. Supporting segments that can be of arbitrary length starting at arbitrary addresses, requires TLBs to form physical addresses by addition than by concatenation, and software for mitigating external fragmentation is harder. Supporting multiple page sizes that are aligned and all powers of two, allows physical addresses to be formed by concatenation, but still requires software to handle external fragmentation and makes the use of set-associative TLBs difficult.

We are aware of two supercomputers and four recent microprocessor architectures that support multiple page sizes. However, there has been little software support for general use of the larger pages. The CDC CYBER 200 [CDC81] supported two page sizes (64KB and a selectable smaller page size) using a fully associative page table. The ETA10 [ETA86] supercomputer supports two page sizes — a large page size of 64KW or 256KW and a small page size of 1KW, 2KW or 8KW (word is 64 bits). The R4000 [Sla91] supports thirteen page sizes (4KB to 16MB) and has a 48-entry fully associative TLB. SuperSPARC [BlK92] supports four pages sizes (4KB, 256KB, 16MB and 4GB) using a 64-entry fully-associative TLB. Hewlett Packard's PA-RISC 1.1 Architecture [HP90] has a 4-entry fully associative Block TLB for large pages (256KB to 16MB), and a separate fully-associative TLB for 4KB pages. The Intel i860 XP microprocessor [INT91] supports one very large page size with a 16-entry TLB for 4MB pages and a separate 64-entry TLB for 4KB pages (both TLBs are four-way set-associative). All require pages to be aligned.

In this study we compare use of two page sizes versus a single page size. We explore the utility of a medium-sized page (e.g. 32KB) instead of megabyte-sized pages that will be used in specific applications only. We consider examining multiple page sizes beyond our scope for two reasons. First, we think it prudent to initially examine the simpler extension. Second, we do not know of a good operating system policy for selecting among many page sizes.

In this paper we show that increasing page size dramatically improves TLB performance at the cost of a large increase in working set size. By using a combination of small and large pages, i.e., 4KB and 32KB, preliminary studies show that it is possible to improve TLB performance with little increase in working set size. For some programs, we show that using two page sizes can result in a better performance than using a single page size of 8KB. We also explore ways to build set-associative TLBs to support two page sizes.

This paper leaves open research issues related to efficient TLB miss handling, page-size assignment policies, memory management and page replacement policies for multiple page size systems.

The remainder of this paper consists of five sections. Section 2 explores some ways to build set-associative TLBs to support two page sizes. Section 3 describes our experimental methodology, metrics and page-size assignment policy. Section 4 discusses the variation of working set size with page size. Section 5 presents results of TLB performance for different page sizes. Finally, we conclude mentioning some open problems that need to be solved before multiple page sizes can be used.

2. Implementing TLBs that Support Two Page Sizes

2.1. Fully Associative TLBs

The most straightforward way to support more than one page size is to use a fully associative TLB. The tag in each TLB entry contains the page size, in addition to enough bits to store the virtual page number for the smallest page size. Hit/miss detection logic uses the page size to select the actual page number for virtual address tag comparison. The page size is also used when forming the physical address.

However, while the fully associative TLB solution is conceptually simple, it can be very expensive to implement. Each TLB entry must (logically) have its own comparator for the virtual address tag comparison logic. Although this may be the only feasible solution if one wants to support more than two page sizes, it may be overkill if one wants to support only two page sizes.

2.2. Set-Associative TLBs

If we restrict ourselves to two page sizes, which are powers of two and aligned, it is possible to build a set-associative TLB. Given that we have two page sizes, which bits do we use to index the set-associative TLB? We could use (a) the page number of the small page, (b) the page number of the large page or (c) the exact page number.

We discuss the three options next, where we assume the small page size to be 4KB and the large page size to be 32KB. For purposes of illustration, we use a 16-bit virtual address, byte addressing and assume bit<0> is the least significant bit of the address.

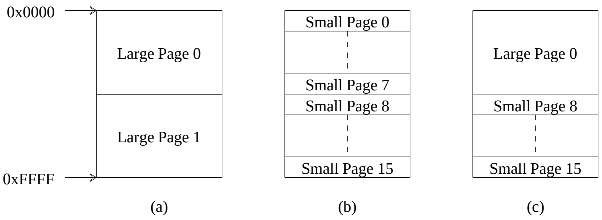

Figure 2.1. Virtual Address Space.

The 32KB page must be aligned on a 32KB boundary. The 16-bit address space can consist of: (a) two 32KB pages, or (b) 16 4KB pages, or (c) a combination of one 32KB and eight 4KB pages.

Indexing the TLB by the Page Number of the Small Page. Consider a direct-mapped TLB with two entries. The TLB is indexed with the least significant bit of the small page number, bit<12>, of the virtual address. This works fine in the case shown in Figure 2.1(b), where the entire address space consists of small pages. On the other hand, if the address space has large pages as in Figures 2.1(a) or 2.1(c), each large page can be mapped to *both* entry 0 and entry 1, depending on the value of bit<12>, which is part of its page offset. This negates the very reason to support both large and small pages. We conclude that using the least significant bits of the small page number to index a set-associative TLB supporting two page sizes does not work.

Indexing the TLB by the Page Number of the Large Page. Again, consider a direct-mapped TLB with two entries. The TLB is indexed with the large page number, bit<15> of the virtual address. This works fine when the entire address space consists of large pages only, as in Figure 2.1(a). On the other hand, if small pages are used, as in Figures 2.1(b) or 2.1(c), eight consecutive small pages compete for the same TLB entry. For example in Figure 2.1(b), Small pages 0-7 all compete for TLB entry 0. This causes many conflict misses due to collisions within a TLB set. We expect this collision cost is not as large as it sounds, because:

a) If the consecutive small pages are indeed being used together, the operating system should allocate them together as a large page, as shown in Figure 2.1(c).

b) If all the small pages are accessed sequentially (but not in a loop), then only entry 0 is used instead of overwriting the rest of the TLB. Hence, it is a good idea to index the TLB with the large page number if one is using small pages for a sequential scan.

c) We can reduce the collision cost by increasing the degree of associativity. In this example, if we increase the degree of associativity to eight, then each of small pages can have their own TLB entry, though they all map to set 0.

If only 32KB pages are used, this scheme degenerates to using a TLB supporting 32KB pages only. If only 4KB pages are used (i.e., no large pages are used, though the hardware supports them), then the collision cost of a set-associative TLB becomes significant and this TLB performs worse than TLBs supporting only 4KB pages (Section 5.2.1).

Indexing the TLB by the Exact Page Number. Again, consider a direct-mapped TLB with two entries. The TLB is indexed by the large page number (bit<15>) when we have a large page and by the least significant bit of the small page number (bit<12>) when we have a small page. However, we do not know the page size, when we access the TLB. We can follow any of three strategies to index the TLB :

(a) Parallel access: Build a dual-ported or replicated TLB that can be indexed by both the small and large page numbers simultaneously.

(b) Sequential access: First assume the incoming address belongs to a small page and index the TLB with the small page number. If we have a miss, reprobe [KJL89] the TLB with the large page number.

(c) Split TLBs: Have a separate TLB for each page size. This is similar to supporting split instruction and data TLBs. Access both TLBs in parallel using different page numbers (small and large).

Option (a) can be expensive and it is not clear whether it is much easier to build than a fully associative TLB. Option (b) results in the TLB hit cost of one page being higher than the other. It is not clear this gives any performance advantage for using the larger page size. Option (c) will result in unused hardware if pages are not appropriately distributed between the two page sizes.

2.3. How to Handle Misses in a TLB that Supports Two Page Sizes

The time taken to handle a miss, the *miss penalty*, is an important parameter in determining TLB performance [HeP90]. Supporting two page sizes makes miss handling more difficult as we do not know the page size for the memory reference that caused the miss. We can expect the miss penalty for a TLB supporting two page sizes to be larger than for a TLB supporting a single page size.

The precise impact of two page sizes on the miss penalty can be determined only by knowing the data structures used by the operating system. We assume that a TLB miss causes a software exception, which invokes a TLB miss-handling routine to scan the software data structures and supply the required translation entry. We estimate that miss-handling routines that support two page sizes take about 25% longer to execute than similar routines for a single page size. This estimate is based on routines written in assembly code for the SPARC architecture [SPA91].

Typical miss handlers use the least significant bits of the page number to index into a data structure, and this is more complicated with two page sizes. A multi-level table or split tables accessed by trying all page sizes in some order may be candidates for page-table data structures. Use of a software cache of translation entries indexed using techniques similar to those discussed above might be advantageous. Precise miss-handling techniques and software data structures that can be used in page tables for two page sizes are beyond the scope of this paper.

3. Methodology

We conducted extensive trace-driven simulations to study the working set sizes and TLB performance when using a single page size and two page sizes. This section details the workloads, metrics, simulation techniques and policy for page-size assignment used in the experiments.

3.1. Workloads

Table 3.1 lists the details of the workloads used to generate the traces for the simulations. All the workloads are uniprogrammed, user-only traces for the SPARC architecture. The programs represent different application categories, reference a large amount of memory and are relatively long. Trace lengths are number of memory references made by the programs. "WS Size" is the average working set size for the program when using 4KB pages, with the working set parameter T equal to ten million references. We simulated the SPEC benchmarks [SPE89] by executing the programs and dynamically generating traces of their memory references, using the tracing tools **shade** [Cme91] and **shadow** [Hsu89].

We would prefer traces with multiprogrammed and operating-system behavior to exercise large TLBs. Since our tracing tools did not allow us to generate such traces, we could not consider them in this study. The only public-domain traces with operating system behavior that we are aware of are the ATUM traces [ASH86]. We do not use these traces as they are too short to exercise the TLBs and the page-size assignment policy.

3.2. Metrics

In choosing a new page size scheme — a single larger page size or a two page size scheme — we tradeoff increased memory demands against better TLB performance. We quantify increased memory demands with the metric *normalized working set size*, $WS_{Normalized}$, which represents the increase in average working set size with respect to using 4KB pages. We measure TLB performance using contribution to CPI due to TLB miss handling, CPI_{TLB}.

Program	Application	Trace Length (million)	Average WS Size (KB)	Average refs. per instruction
li	int SPEC	6512	146	1.313
espresso	int SPEC	632	168	1.280
fpppp	FP SPEC	2166	221	1.497
doduc	SPEC	1698	246	1.291
x11perf	graphics	83	433	1.207
eqntott	int SPEC	1662	753	1.151
worm	graphics	104	1115	1.207
nasa7	FP SPEC	9247	1124	1.395
xnews	X server	644	1259	1.187
matrix300	FP SPEC	2347	1484	1.385
tomcatv	FP SPEC	2297	2948	1.413
verilog	logic sim.	729	15059	1.264

Table 3.1. Workloads.

We derive $WS_{Normalized}$ as follows. $W(t, T, ps)$ is the set of distinct pages referenced in the interval $[t-T+1, t]$, where T is a parameter of the working set algorithm [Den68] and ps specifies either a single page size or multiple page sizes and how to select between them. The *working set size*, $w(t, T, ps)$, is the sum of the sizes of pages in $W(t, T, ps)$. The *average working set size* is $s(T, ps) = \frac{1}{k}\sum_{t=0}^{k} w(t, T, ps)$, where k is the number of memory references in the program. Finally, *normalized working set size*, $WS_{Normalized}(ps)$, is:

$$WS_{Normalized}(ps) = \frac{s(T, ps)}{s(T, 4KB)}.$$

$WS_{Normalized}$ represents the increase in average working set size when using a new page size combination, *ps*, with respect to the average working set size for 4KB pages. For example, a $WS_{Normalized}(32KB)$ of 1.5 means that the average working set size increased by 50% by changing the page size from 4KB to 32KB. It is difficult to relate $WS_{Normalized}$ directly to a change in program execution time without considering many system-specific parameters such as physical memory size, page replacement policy, multiprogrammed workload, etc. However, unless memory is underutilized, increased working set size would either require more physical memory to run the same program or would increase the page fault rate.

We measure TLB performance as the contribution to CPI due to TLB miss handling, CPI_{TLB}. This metric can be directly related to program execution time [HeP90]. CPI_{TLB} is calculated as follows :

$$CPI_{TLB} = (TLB\ misses\ per\ instruction) \times (TLB\ miss\ penalty).$$

From trace-driven simulations, we calculate the *TLB misses per instruction (MPI)* factor. The TLB miss penalty depends on both hardware and software and is more difficult to estimate (see Section 2.3). In this paper we assume the miss penalty to be twenty cycles, which is a reasonable estimate for TLB misses handled in software. Other metrics used to measure TLB performance can easily be calculated from CPI_{TLB} and RPI (References per instruction from table 3.1),

$$TLB\ misses\ per\ instruction\ (MPI) = \frac{CPI_{TLB}}{20},$$

$$TLB\ miss\ ratio = \frac{MPI}{RPI}.$$

Our studies assume a 25% increase in miss penalty for a TLB supporting two page sizes.(Section 2.3). The increase also will account for the costs of *page promotion* (Section 3.4). Further, the results do not change significantly with moderate changes in the miss penalty.

Another metric that is of interest is the increase in miss penalty that can be tolerated when using two page sizes to give the same CPI_{TLB} as using a single page size of 4KB, called the *critical miss penalty increase*, $\Delta\hat{mp}(ps) =$

$$\left(\frac{MPI(4KB)}{MPI(ps)} - 1\right) \times 100\% = \left(1.25 \times \frac{CPI_{TLB}(4KB)}{CPI_{TLB}(ps)} - 1\right) \times 100\%$$

For the two page sizes, we use 4KB and 32KB pages. We also have similar data for combinations of 4KB/16KB and 4KB/64KB, but space constraints prevent us from presenting them here.

3.3. Simulation Technique

Due to the large number of different TLB configurations that needed to be simulated and the long running time of the trace-driven simulations, it was not feasible to perform simulations one at a time.

We performed TLB simulations with the cache simulator **tycho**. **Tycho** implements all-associativity simulation [HiS89], a variation of stack simulation [MGS70]. We modified **tycho** to handle using index bits other than the least significant bits of the page number and to support two page sizes. Using **tycho** it was possible to simulate many TLB configurations (84 in our case) in one simulation in about double the simulation time for a comparable single TLB simulation.

We also used stack simulation techniques to calculate average working set sizes. Slutz and Traiger[1] [SlT74] show that working set size calculation for different page sizes and different values of T can be done using stack simulation. We modified the algorithm, which requires T counters, to use very few counters, making it feasible to run simulations for large values of T (namely 100 million).

For each trace, we simulated more than one thousand TLB configurations and calculated average working set sizes for sixty-three page size/T combinations. We consumed 5.5 months of CPU time to collect the data, a subset of which we present in this paper.

1. We do not use Slutz's algorithm for multiple page sizes [Slu75] because it does not allow page sizes to change dynamically, which we assume in our page-size assignment policy (Section 3.4).

3.4. Policy for Page-Size Assignment

To test the performance of TLBs supporting two page sizes, we need a policy to assign page sizes to regions of the address space. We need a complete implementation of the operating system to do an accurate page-size assignment. In the absence of any real operating system policy, we use the following method to arrive at a page-size assignment. We do the assignment dynamically during the simulation, looking at the last T references.

We treat the virtual address space as chunks of 32KB each. Each chunk consists of eight blocks of 4KB. During the simulation we maintain a list of all the blocks accessed in the last T references. We map each chunk as either one large page of 32KB or eight small pages of 4KB each. The decision on whether or not to promote a chunk to a large page depends on the number of blocks accessed within the window of the last T references. If all the blocks in a chunk have been accessed, then the chunk should certainly be mapped as a large page. If only one block has been accessed then the chunk should remain mapped as small pages. The threshold we use to promote is whether half or more of the blocks in a chunk have been accessed. In this way, at worst we only double the working set size (in reality the increase is much less).

This policy uses the inherent spatial locality in the programs without the compiler or operating system taking care to align data structures on page boundaries. With software support for proper packing of data and code, more large pages would be used, which would lead to better TLB performance and smaller working set size increases.

There are some costs associated with promoting a chunk from small pages to a large page : (a) cost of updating the mapping data structures and invalidating TLB entries for the small pages, (b) copying the small pages already in memory to one large page and (c) paging in/zeroing the small pages not resident in memory. This will increase the CPI_{TLB} for the two-page-size schemes. We expect these costs to be low as page promotions are not frequent. Further, these costs can be folded into the extra miss penalty we assume for the two-page-size schemes.

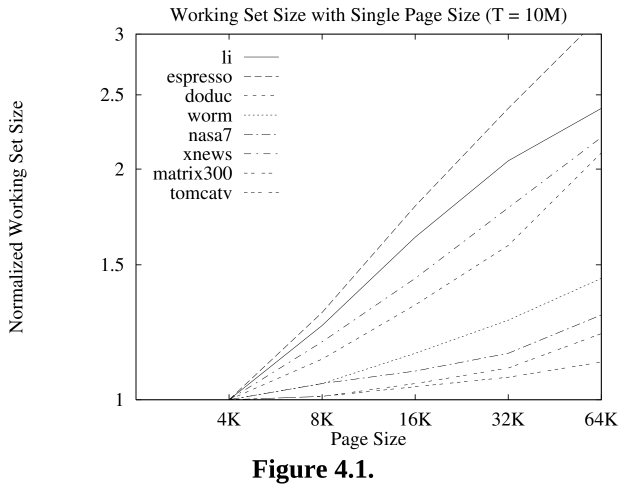

Figure 4.1.

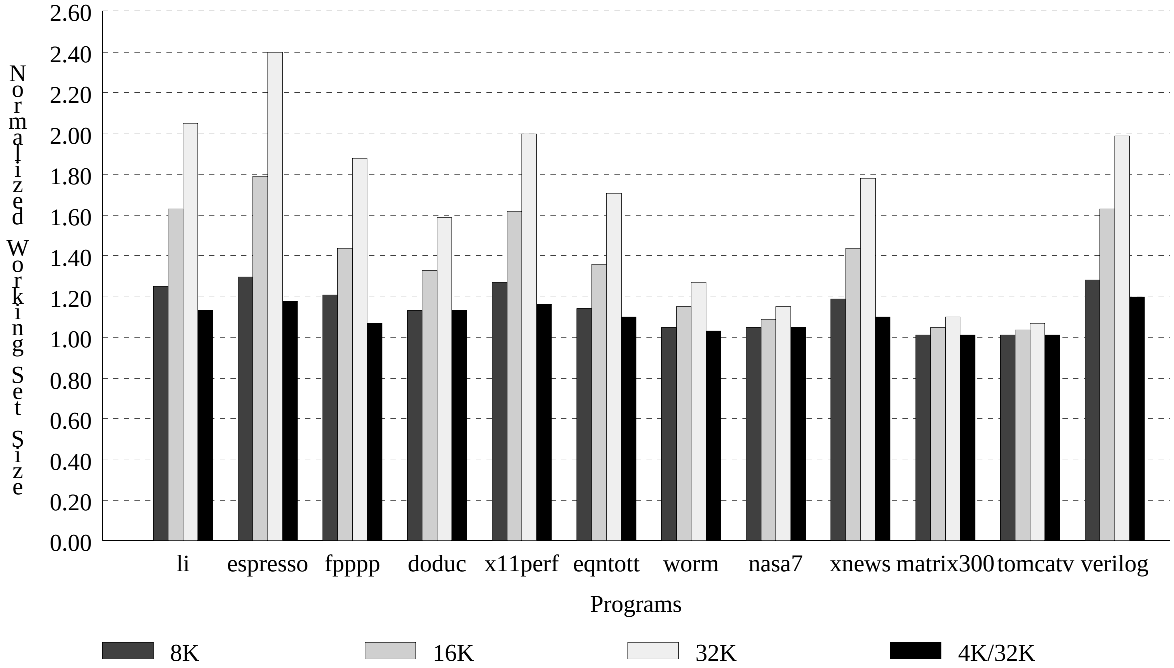

Figure 4.2. Normalized Working Set Sizes (T = 10M).

4. Results of Working Set Size Calculations

In this section we study the variation of working set sizes when using page sizes larger than 4KB. The working set size of a program is a measure of the memory cost of using larger pages. Though current operating systems do not use the working set policy for page replacement, we consider the working set size a good estimate of programs' memory demands.

Figure 4.1 shows the variation of $WS_{Normalized}$ with page size for T = 10 million references. The X-axis shows the page size and the Y-axis shows $WS_{Normalized}$ (both axes are logarithmic). The graph shows that the working set size increases quite significantly by using larger page sizes.

Doubling the page size increases working set size by only 1% to 30%. Doubling the page size does not double the working set size because programs exhibit spatial locality [Den70]. Programs like *matrix300* and *tomcatv*, that traverse all their address space in a linear looping fashion, already have most of their address space in the working set and increasing page size does not affect their working set sizes much. Programs that have a sparse address space (e.g., *li*) or have good temporal locality in a small region of the address space (e.g., *espresso*) show large increases in working set sizes.

$WS_{Normalized}$ is approximately proportional to the page size. Though the exact values may change, the qualitative trend is not sensitive to varying T through 10, 25 and 50 million references. Averaging across all the traces, $WS_{Normalized}(32KB)$ is 1.67 and $WS_{Normalized}(64KB)$ is 2.03, for T = 10 million references.

Figure 4.2 shows $WS_{Normalized}$ for the various programs for single page sizes from 8KB to 32KB and the two-page-size scheme using a combination of 4KB and 32KB pages. Recall that $WS_{Normalized}$ is 1.0 for 4KB pages.

The working set size increase resulting from two page sizes is less than for any single page size larger than 4KB. Across all the programs, $WS_{Normalized}(4KB/32KB)$ is only 1.01 to 1.22 (average 1.1). Further, this increase is insensitive to the value of T, varying by only a few percent.

Our results show that there is a significant increase in working set size by using a larger single page size. We conclude that the memory cost of using two page sizes is small and might be a better option than using a single larger page size, even if that single page size is 8KB.

5. Results of TLB Simulations

The main motivation for using large pages is to improve the effectiveness of TLBs. In this section we present results of preliminary studies comparing single-page-size TLBs with those that can support two page sizes. Space constraints allow us to present data for only three sample TLB organizations, one fully associative and two set-associative.

We divide the traced programs into two categories based on their working set sizes. The small programs have a working set size of less than 1MB and the large programs have working sets larger than 1MB (Table 3.1). The traces are displayed in ascending order of working set size, so that effects due to working set size, if any, may be observed.

We do not use large TLBs ($\geq$ 64 entries) in our study. Large TLBs in combination with large pages have negligible miss rates for our workloads, making it impossible to make comparisons. Nevertheless, our current results are our best estimate of the relative behavior of future TLBs,

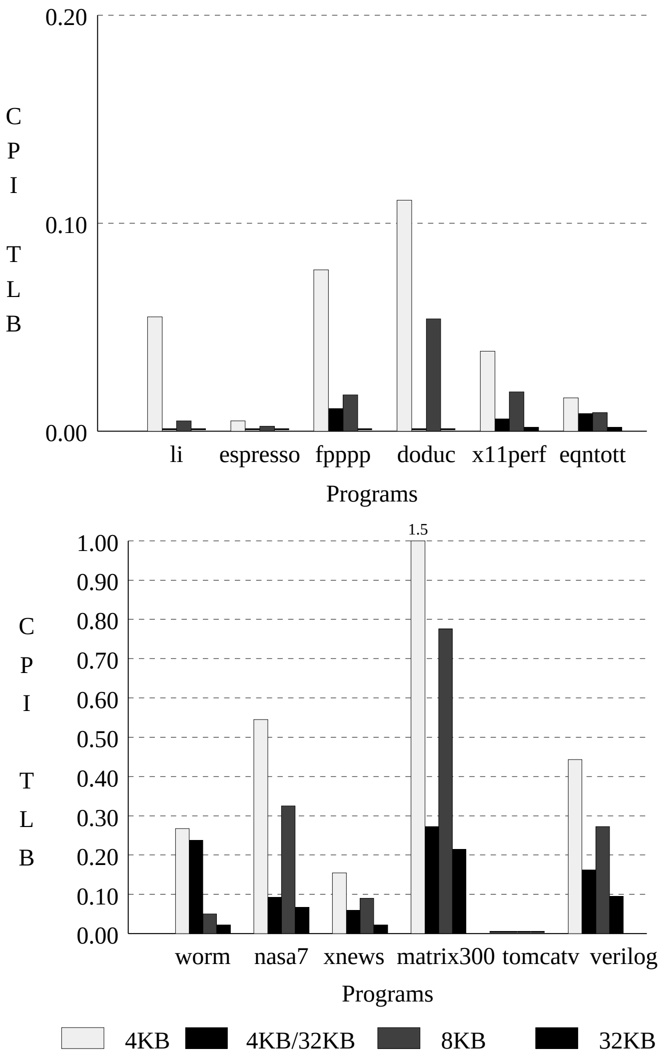

Figure 5.1. CPI contribution for 16-entry fully associative TLB.

which will be larger, but exercised by user programs with larger working set sizes in multiprogrammed workloads that include operating system behavior.

5.1. Fully Associative TLBs

Figure 5.1 shows CPI_{TLB} for a 16-entry fully associative TLB. Using a single large page size (32KB) gives the best CPI_{TLB}. The TLB now maps eight times more memory than when using a single page size of 4KB and the CPI_{TLB} reduces by approximately a factor of eight. Using a single page size of 8KB doubles the mapped area and nearly halves the CPI_{TLB}.

CPI_{TLB} when using two page sizes are only slightly worse than that obtained when using a single large page size of 32KB. The difference in CPI_{TLB} is mostly due to the increase in miss penalty when using two page sizes. The results show that there is a significant benefit in using a combination of large and small pages. Further the improvement does not require a large increase in working set size (Section 4).

For many programs the two-page-sizes scheme has lower CPI_{TLB} than using a single page size of 8KB. The use of large pages (32KB) in the two-page-sizes scheme has the potential to map a much larger area than a single page size of 8KB. Further, using two page sizes has a penalty comparable in working set size to using a single page size of 8KB. It should be noted, however, that the choice of page sizes is an architectural issue and the decision should not be based on any particular implementation.

5.2. Two-Way Set-Associative TLBs

Figure 5.2 shows the CPI_{TLB} for 16-entry and 32-entry, two-way set-associative TLBs. The histograms compare CPI_{TLB} when using the exact index for the two-page-sizes scheme with CPI_{TLB} for a single page size TLB. We use exact indexing for the two-page-sizes TLBs as we expect this to do the best. Section 5.2.1 compares the different indexing schemes. The results for two-way set-associative TLBs are, however, not as regular as in the fully associative case.

Using large pages lowers the CPI_{TLB} quite significantly because the same TLB now maps a larger region of memory. *Matrix300* is an extreme example: CPI_{TLB} reduces from 1.56 for 4KB pages to 0.11 for 32KB pages (for a 32-entry TLB). Using a single page size of 32KB has the lowest CPI_{TLB}. A single page size of 8KB, also produces a significant improvement in CPI_{TLB}.

For eight of the twelve programs, using two page sizes delivers lower CPI_{TLB} than using a single page size of 4KB, even with the increased miss penalty for the two-page-sizes scheme. The improvement in CPI_{TLB} varies from very large (e.g., *matrix300, nasa7*) to moderate (e.g., *verilog, xnews*). This shows that it is possible to support two page sizes using set-associative TLBs while improving TLB effectiveness. However, for a couple of programs (e.g., *espresso, worm*) there is a degradation in CPI_{TLB} by using two page sizes. This is due to insufficient use of large pages during page-size assignment and the larger miss penalty for the two-page-sizes scheme.

$WS_{Normalized}$ for the two-page-sizes scheme and for a single page size of 8KB are comparable. Comparing CPI_{TLB} in the two cases, however, there are no clear trends. For a 16-entry TLB, the small programs seem to benefit more from using two page sizes. On the other hand, for a 32-entry TLB, the small programs seem to benefit more from a single 8KB page size. For large matrix-manipulation programs (e.g., *nasa7* and *matrix300*) using two page sizes is a better option as they use mostly large pages and get CPI_{TLB} comparable to using 32KB pages. The results might suggest that increasing the page size to 8KB is better than using two page sizes. However, as programs continue to get larger, there may be more benefit in moving to a combination of small and large pages than to a single page size of 8KB.

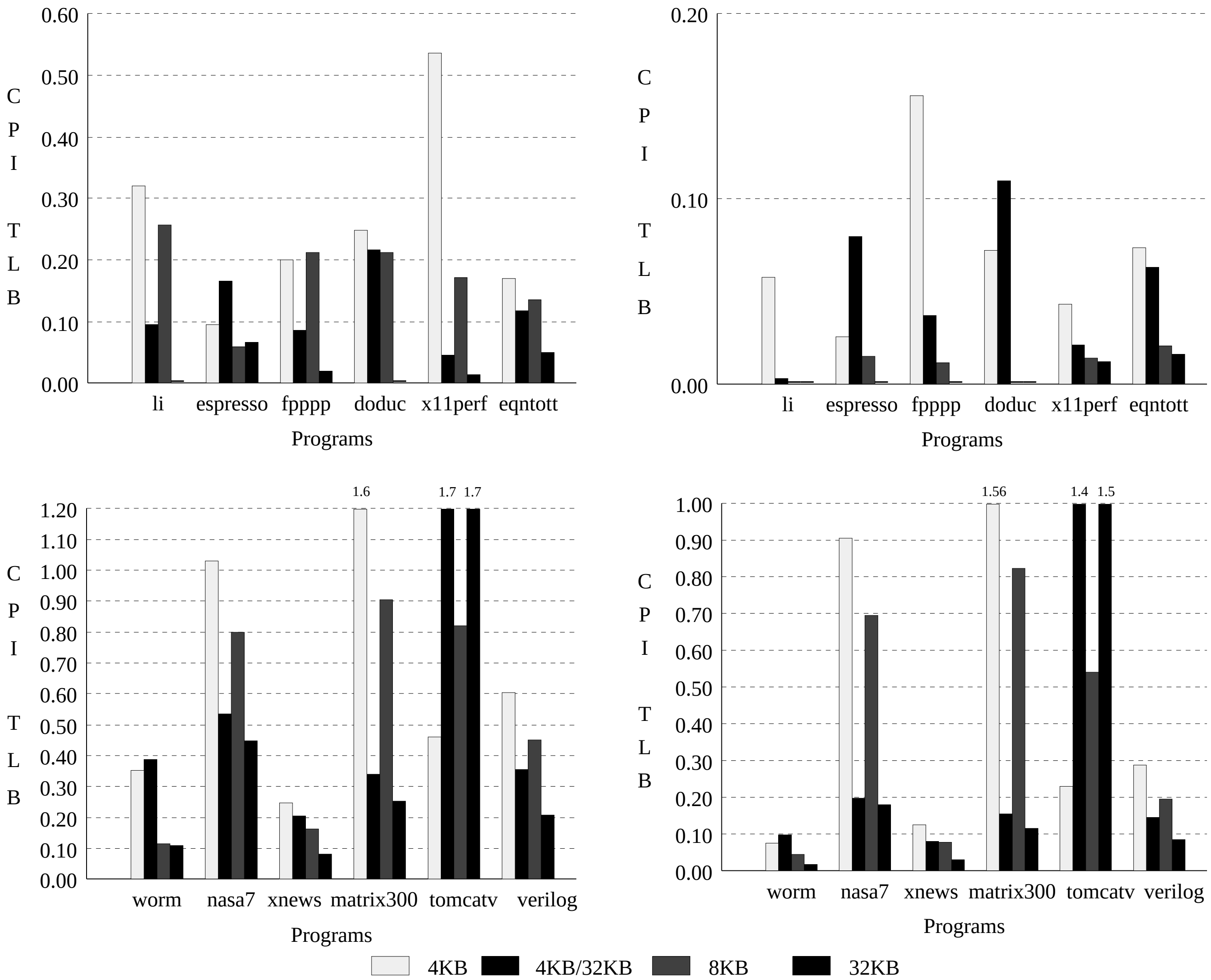

Figure 5.2. CPI contribution for set-associative TLB.
(16 entry, two-way) (32 entry, two-way)

To consider the effect of increase in miss penalty for TLBs supporting two page sizes, we look at the critical miss penalty increase. $\Delta \hat{mp}(4KB/32KB)$ varies from 30% to 1200% for programs that show an improvement in CPI_{TLB} when using two page sizes. This shows that even with a 30% increase in miss penalty for two page sizes, the same trends/results will hold. We conclude that a moderate increase in miss penalty when using two page sizes can be tolerated to give better performance than when using a single page size of 4KB.

Though there is a general trend of decreasing CPI_{TLB} when using larger pages, there are some exceptions. For example, *Tomcatv* has a quickly degrading CPI_{TLB} for increasing page sizes for two-way set-associative TLBs. The program's access pattern causes the TLB to thrash even with larger pages. Alexander *et al.* [AKB85] report similar behavior for different traces. We do not see any such anomalies for higher associativities, which indicates that they are due to interaction between the access pattern and the bits used to index the TLB.

The results show that there is potential for improving CPI_{TLB} by using a combination of small and large page sizes. We have also shown that a moderate increase in miss penalties for the more complicated TLBs does not affect the trends. Since the policy for allocation of pages (Section 3.4) is not realistic however, these results are only preliminary. Furthermore, with a real operating system using two page sizes, we expect the allocation pattern to change and result in different TLB behavior — set conflicts are sensitive to allocation of data structures in the virtual address space. Exact performance gains for using two page sizes can only be estimated with realistic policies for page-size assignment, more accurate estimates for miss penalty, estimates of the impact of increased working set size on the system page fault rate, and studies of larger and multiprogrammed workloads.

Program	4KB	4KB large index	4KB/32KB large index	4KB/32KB exact index
16-entry, two-way				
li	0.320	0.452	0.026	0.095
espresso	0.095	0.310	0.170	0.166
fpppp	0.201	0.174	0.082	0.086
doduc	0.248	0.488	0.058	0.216
x11perf	0.536	0.360	0.039	0.045
eqntott	0.170	0.207	0.178	0.118
worm	0.352	0.468	0.489	0.389
nasa7	1.029	1.149	0.635	0.537
xnews	0.247	0.297	0.178	0.205
matrix300	1.624	1.604	0.322	0.339
tomcatv	0.461	2.608	2.246	1.752
verilog	0.604	0.821	0.357	0.357
32-entry, two-way				
li	0.058	0.273	0.013	0.003
espresso	0.026	0.113	0.032	0.080
fpppp	0.156	0.141	0.041	0.037
doduc	0.072	0.294	0.016	0.109
x11perf	0.043	0.352	0.033	0.021
eqntott	0.074	0.094	0.052	0.063
worm	0.077	0.376	0.353	0.100
nasa7	0.907	0.815	0.296	0.199
xnews	0.126	0.211	0.086	0.080
matrix300	1.568	1.557	0.152	0.156
tomcatv	0.232	2.464	2.020	1.437
verilog	0.288	0.619	0.180	0.145

Table 5.1. Comparison of indexing schemes.

5.2.1. Comparison of Indexing Schemes for Set-Associative TLBs

Table 5.1 compares CPI_{TLB} using different indexing schemes for 16- and 32-entry two-way set-associative TLBs supporting two page sizes.

Exact indexing is intuitively the better indexing scheme. Although there are more cases where the exact index does better than the large page index, in more than 50% of the programs the CPI_{TLB} are roughly comparable. The two indexing schemes differ only in how they handle small pages. This comparable behavior might be due to the following (a) There may be few small pages (e.g., for *matrix300*). Hence, the CPI_{TLB} is similar to using all large pages and the indexing scheme for the small pages has little effect. (b) In our page-size assignment, small pages are used only if 1, 2 or 3 of the blocks in a large-page region are accessed. Associativity covers the cases of one or two blocks. So the large page index does worse only if three blocks in a large page region are accessed. We expect this to be infrequent.

Use of large page index is based on assumptions of allocation of enough large pages (Section 2.2). Suppose that a system supports two page sizes and uses the large page index, but the software does not allocate any large pages. The simulations show that this results in a severe degradation in CPI_{TLB} (compare the first two columns in Table 5.1). The degradation occurs because the bits used as the large page index are a poor choice if one is using only small pages. Thus, the effectiveness of hardware to support two page sizes depends on the operating system to do a good page-size assignment. Without software support, hardware supporting two page sizes may do worse than when supporting a single page size of 4KB. This is an important issue if the new hardware has to work with older versions of the operating system.

6. Summary and Conclusions

As main memories get larger and processor cycles-per-instruction (CPI) get smaller, the time spent in handling TLB misses is becoming more significant. It may not be possible to increase the number of TLB entries to map the gigantic memories of the future when using a physically tagged level-one cache without adversely affecting the latency of all memory references. Two alternatives for making a TLB more effective are (a) increasing the page size and (b) supporting two page sizes.

We experimentally examined both alternatives with a dozen uniprogrammed, user-mode traces for the SPARC architecture. Our results show that increasing the page size to 32KB reduces the TLB's contribution to CPI, CPI_{TLB}, by factors of about three to eight (sometimes more) compared to using 4KB pages. However, this CPI decrease is accompanied by a significant increase in average working set size (e.g., 60%). Without making many additional assumptions it is not possible to translate the increased working set size to a CPI penalty. Nevertheless, unless main memory is underutilized, larger working sets either demand a larger main memory, cause a higher page fault rate, or both.

Our results show that the use of two page sizes — 4KB and 32KB — keeps the increase in average working set size modest (about 10%), and makes CPI_{TLB} reductions that depend on whether the TLB is fully associative or set-associative. Results for a 16-entry fully associative TLB show that performance of two page sizes is mostly comparable to using the large page size. With two-way set-associative TLBs, due to the complexity of indexing with two page sizes, the results range from no improvement to CPI_{TLB} comparable to using 32KB pages. In a few cases, there is even a degradation in CPI_{TLB} resulting from using two page sizes. Furthermore, CPI_{TLB} is adversely affected by using the large page index if the operating system does not allocate any large pages.

We also compared using two page sizes with increasing the single page size to 8KB. Results show that (a) both cause a similar, modest increase in average working set size, (b) using two page sizes leads to a lower CPI_{TLB} with a fully associative TLB, but (c) results are mixed for set-associative TLBs due to indexing issues.

Even if all our assumptions are accurate, our results neither conclusively reject nor conclusively support the use of two page sizes. Furthermore, there are three important reasons not to infer too much from our results. First, the two-page-size results critically depend on the operating system's policy and mechanism for selecting between the two page sizes. Instead of modifying the operating system, we used traces generated with current software and dynam-

-- --

ically remapped page sizes in response to usage patterns. A real page-mapping policy may perform much better (e.g., by reorganizing code and data for the new page sizes) or much worse (e.g., mapping policies might use less dynamic information). Second, despite consuming nearly 5.5 months of CPU simulation time, our traces are inadequate to exercise large TLBs, in part, because they do not include the effect of multiprogramming and operating systems behavior. Third, the selection of page size or sizes is determined by more factors than we have examined, including disk performance, paging performance and protection granularity. Furthermore, the selection of page size is typically an architectural, not an implementation, decision. Thus, one should not be satisfied that a page size is good for a particular machine, but ask whether it is likely to be good for most implementation technologies and workloads the architecture is likely to encounter in the future. One should not, for example, select page sizes that require the use of a fully associative TLB unless one expects that most future technologies will support fully associative TLBs well.

We hope that this paper motivates more research, particularly operating system research, to solve the problems associated with supporting multiple page sizes and to see whether they should be the preferred alternative.

7. Acknowledgements

We would like to thank R. Cmelik, D. Ditzel, Ed Kelly, Y. Khalidi, S. Kleiman, S. Richardson, G. Taylor, D. Williams and M. Wing among others at Sun Microsystems, Inc. and Sun Microsystems Laboratories, Inc. for their help with this research; D. Chenevert, M. Choudhary, S. Jain, R. Lee, V. Matena, G. Maturana, R. Subrahmaniam and R. Yung for helping with the traces; S. Adve, S. Chessin, S. Muchnick, A. Pai, S. Vajapeyam and D. Wood for their comments and help in writing this paper.

References

[ASH86] A. AGARWAL, R. L. SITES and M. HOROWITZ, ATUM: A New Technique for Capturing Address Traces Using Microcode, *Proc. Thirteenth Intl. Symp. on Computer Architecture* , June 1986, 119-129.

[AKB85] C. A. ALEXANDER, W. M. KESHLEAR and F. BRIGGS, Translation Buffer Performance in a UNIX Environment, *Computer Architecture News*, December 1985, 2-14.

[ApL91] A. W. APPEL and K. LI, Virtual Memory Primitives for User Programs, *Proc. 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, April 1991, 96-107.

[BlK92] G. BLANCK and S. KRUEGER, The SuperSPARC Microprocessor, *COMPCON*, San Francisco, February, 1992, 136-141.

[ClE85] D. W. CLARK and J. S. EMER, Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement, *ACM Transactions on Computer Systems 3*, 1 (February 1985), 31-62.

[Cme91] R. F. CMELIK, Introduction to SpixTools, Sun Microsystems Technical Memorandum, June 1991.

[CDC81] *CDC CYBER 200 Model 205 Computer System, Hardware Reference Manual*, Control Data Corporation, 1981.

[Den68] P. J. DENNING, The Working Set Model for Program Behavior, *Communications of the ACM 11*, 5 (May 1968), 323-333.

[Den70] P. J. DENNING, Virtual Memory, *Computing Surveys 2*, 3 (September 1970), 153-189.

[ETA86] *Mainframe Subsystem Instruction Specification for the ETA10, Rev: B*, ETA Systems, Inc., March 1986.

[HeP90] J. L. HENNESSY and D. A. PATTERSON, *Computer Architecture A Quantitative Approach*, Morgan Kaufmann Publishers Inc., 1990.

[HP90] *PA RISC 1.1 Architecture and Instruction Set Reference Manual*, Hewlett Packard, November 1990.

[HiS89] M. D. HILL and A. J. SMITH, Evaluating Associativity in CPU Caches, *IEEE Trans. on Computers C-38*, 12 (December 1989), 1612-1630.

[Hsu89] P. Y. HSU, Introduction to SHADOW, Sun Microsystems Technical Memorandum, July 1989.

[INT91] *Overview of the i860 XP Supercomputing Microprocessor*, Intel Corporation, 1991.

[KJL89] R. E. KESSLER, R. JOOSS, A. LEBECK and M. D. HILL, Inexpensive Implementations of Set-Associativity, *Proc. 16th Symp. on Computer Architecture* , June 1989, 131-139.

[MGS70] R. L. MATTSON, J. GECSEI, D. R. SLUTZ and I. L. TRAIGER, Evaluation Techniques for Storage Hierarchies, *IBM Systems Journal 9*, 2 (1970), 78-117.

[Org72] E. J. ORGANICK, *The Multics System: An Examination of Its Structure*, MIT Press, Cambridge, MA, 1972.

[SPA91] *The SPARC Architecture Manual, Version 8*, SPARC International Inc., Menlo Park, CA., 1991.

[SPE89] SPEC, Newsletter, Vol. 1, 1989.

[SaB81] M. SATYANARAYANAN and D. BHANDARKAR, Design Trade-offs in VAX-11 Translation Buffer Organization, *IEEE Computer 14*, 12 (December 1981), 103-111.

[Sla91] M. SLATER, MIPS Previews 64-Bit R4000 Architecture, *Microprocessor Report 5*, 2 (February 6, 1991), 1,6-9,18.

[SlT74] D. R. SLUTZ and I. L. TRAIGER, A Note on the Calculation of the Average Working Set Size, *Communications of the ACM 17*, 10 (October 1974), 563-565.

[Slu75] D. R. SLUTZ, A Relation Between Working Set and Optimal Algorithms for Segment Reference Strings, IBM Research Report RJ 1623, July 1975.

[Smi82] A. J. SMITH, Cache Memories, *Computing Surveys 14*, 3 (September, 1982), 473 - 530.

[WEG86] D. A. WOOD, S. J. EGGERS, G. GIBSON, M. D. HILL, J. PENDLETON, S. A. RITCHIE, R. H. KATZ and D. A. PATTERSON, An In-Cache Address Translation Mechanism, *Proc. 13th Intl. Symp. on Computer Architecture*, Tokyo, Japan, June 1986.