MAY 1992

WRL Research Report 91/2

A Simulation-Based Study of TLB Performance

J. Bradley Chen, Anita Borg, and Norman P. Jouppi

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, WRL-2
250 University Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`DECWRL::WRL-TECHREPORTS`
Internet:	`WRL-Techreports@decwrl.dec.com`
UUCP:	`decwrl!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

A Simulation-Based Study of TLB Performance

J. Bradley Chen, Anita Borg, and Norman P. Jouppi

May, 1992

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

Abstract

This paper presents the results of a simulation-based study of various translation lookaside buffer (TLB) architectures, in the context of a modern VLSI RISC processor. The simulators used address traces, generated by instrumented versions of the SPECmarks and several other programs running on a DECstation 5000. The performance of two-level TLBs and fully-associative TLBs were investigated. The amount of memory mapped was found to be the dominant factor in TLB performance. Small first-level FIFO instruction TLBs can be effective in two level TLB configurations. For some applications, the cycles-per-instruction (CPI) loss due to TLB misses can be reduced from as much as 5 CPI to negligible levels with typical TLB parameters through the use of variable-sized pages.

This research report is a preprint of a paper to appear in the
Proceedings of the 19th International Symposium on Computer Architecture.

1. Introduction

In computer systems with virtual memory, a TLB is typically used to provide fast translation of virtual addresses generated by instruction execution to physical addresses needed for cache tag comparisons. Both physically and virtually addressed caches require address translation. With physically addressed caches, the TLB lookup is in the critical path of cache access, so low latency and miss rates are crucial for memory system performance. The TLB is a cache, speeding up access to entries in the page table, where complete information on virtual to physical memory mappings is maintained. Most modern machines use split instruction and data caches, and this configuration is assumed (unless stated otherwise) in the remainder of this paper. Given this context, we consider several possibilities for the TLB implementation:

- **A single TLB shared between instruction and data caches.** To reduce contention, the TLB can be dual-ported. This introduces complex circuitry, doubling the size of the TLB without increasing its capacity.
- **Independent TLBs for instruction and data caches.** The instruction TLB should be made smaller than the data TLB, as instruction reference streams exhibit greater locality than those for data. The appropriate size tradeoff is difficult to determine and once made is fixed. If the instruction TLB is too small, performance will suffer. If it is too large, the space available for the data TLB is compromised and again performance suffers.
- **Two-level TLB architectures.** A small instruction TLB (i.e., *micro-TLB*), can be refilled from a larger single-ported shared TLB, primarily used for data references. This option is described in more detail below.

A micro-TLB is a fully associative TLB with a very small number of entries (probably less than eight) which is reloaded in hardware from a larger shared TLB. A number of recent machines use micro-TLBs, including the MIPS R4000 [7], though they are invisible at the architecture level. A micro-TLB is accessed in parallel with the instruction cache. On a miss, the micro-TLB is reloaded from the shared TLB. As the larger TLB is single ported, the CPU may stall for a few cycles, with data references suspended while the TLB is busy, but this penalty is much less expensive than that of a full TLB miss. We assume 3 cycles as the micro-TLB miss penalty in this paper. Because of the high locality in instruction reference streams, and the relatively small miss penalty, acceptable miss rates can be achieved with a small micro-TLB. The balance of instruction and data entries in the shared TLB is determined dynamically, unlike in the second option above. In addition the shared TLB need not be dual-ported, so the extra space can be used to increase its capacity. No previous research known to the authors characterizes the design space for the micro-TLB.

Because the TLB can be in the critical path of memory access, good TLB performance is essential to good overall performance of a machine. TLB design has been complicated in several recent architectures with split instruction and data TLBs. To date, such designs have received negligible attention in the research literature. Experimental results are presented herein to characterize the behavior of split as well as shared TLBs.

Another feature found in several recent architectures is TLB entries that can map variable size pages. When such a TLB entry is loaded with a new mapping, it is also loaded with the size of the page to be mapped. Typically, the size is restricted to a power of two and may range from

4K bytes to a gigabyte [5, 7]. Although there are several obvious applications of variable size pages, such as mapping operating system text and graphics frame buffers, it is not yet understood to what degree they can be used to improve the execution rate of application code. Performance of applications which concentrate references on a contiguous segment could improve if that segment were accessed as a single page with a single TLB entry. Applications which scatter references across a sparse address space have little hope of benefiting from large pages without significantly increased memory usage. Address traces and reference counting tools can be used to record dynamic patterns of memory access, to aid in understanding the applicability of these structures.

The many recent studies on memory system behavior and performance have concentrated almost exclusively on cache design [10, 9]. Little attention has been given to TLB performance. Early studies have shown that TLB miss penalties consume 6% of all machine cycles [4] and 4% of execution time [3], and hence can have a significant impact on machine performance. However, these results were for VAX computers with 512 byte page sizes, an order of magnitude smaller than is typical today, and main memory sizes two orders of magnitude smaller than those considered in this study.

Wood [12, 11] proposed in-cache address translation as an alternative to a TLB. His work has shown that such methods are effective for programs such as Lisp applications and operating systems, where the working set is spread over a large address space. They are less useful for behavior such as is seen with typical C programs, where memory activity is concentrated in the bottom of several segments. His methods also become less applicable when memory access times become large with respect to processor speed.

Finally, previous TLB studies [3, 11] have considered set-associative or direct-mapped organizations. These were common when TLBs were made from discrete MSI and LSI RAMs. Recently, however, VLSI RISC microprocessors (e.g., [5, 7]) typically make use of fully-associative TLBs, since these require about the same area as set-associative TLBs when implemented within a VLSI chip. Thus the TLB implementations studied in this paper are all fully-associative designs.

This paper is concerned with TLB performance. Understanding the relation between TLB performance and overall machine performance is a different question, involving the balance of compulsory to capacity misses and the relative ability of caches and TLBs to map multiple localities. The first access to a page results in a *Compulsory* TLB miss. In this situation, cache misses also occur, the cost of which might overshadow the TLB penalties. Other TLB misses are *capacity* misses, when a program returns to a locality that has been replaced out of the TLB, though it might still be present in the cache. This phenomena becomes more common as cache sizes increase. In this paper, CPI effects of TLB performance are discussed assuming perfect memory system performance. This can be deceptive, as it trivializes the question of cache performance. More exact results require simulation of a more complete memory system.

The remainder of the paper is structured as follows. First we give some brief details on our methodology. This is followed by a discussion of instruction TLB behavior, both for micro-TLBs and for independent instruction TLBs. Data and shared TLB results are presented next, followed by a discussion of variable size pages, and finally some concluding notes.

2. Methodology

The tracing and simulations were done on a DECstation 5000, using an updated version of the WRL address tracing tools [2]. Application programs are instrumented by inserting code at the start of each basic block and before each load or store instruction to make entries in a per-process trace buffer. The contents of the per-process buffer are appended to a system trace buffer each time the kernel is invoked, and whenever the per-process buffer becomes full. The kernel makes additional entries in the system buffer to record context switches, system calls, and other such events. When the system buffer becomes full, traced processes are suspended and an analysis program, in this case a TLB simulator, is started. The analysis program runs until the contents of the kernel buffer have been consumed, at which time tracing may continue. The system permits multi-process traces and on-the-fly analysis of trace data with minimum distortion to the traces. System references are not included in the traces.

In simulating the various TLB configurations, a variety of workloads were used, including each of the SPECmarks, plus a number of other workloads meant to anticipate more realistic future workloads. *Tree* is a recursive, data intensive benchmark written in C-Scheme [1]. *Magic* [8] is a VLSI layout tool. In this run it was extracting the MultiTitan CPU chip [6]. A multi-tasking mix was also used, running the following programs:

- *gcc*
- *magic* extracting the MultiTitan CPU chip
- *ld* loading *magic*
- *tree* with a 10 megabyte heap
- a loop running the shell programs *cp*, *cat*, *sed*, *ls*, *ps*, and *rm*.

Short running programs were put in loops, so that their execution would continue throughout the entire run. This mix is meant to be comparable to the mix used in previous trace-based studies by Borg et.al. [2].

3. Instruction TLB results

Instruction reference streams place lesser demands on TLB resources than data reference streams. Instruction references generally exhibit higher locality, both spatial and temporal. Also there is generally less memory involved. The largest text segment of the SPECmarks, when compiled for the DECstation, is the Gnu C compiler *gcc* with 688K bytes. The average text segment size is around 200K bytes. Data segments are frequently much larger. Data references for *nasa7*, a benchmark for numeric computation, range over a space of over three megabytes.

Because of the different performance characteristics of TLBs and micro-TLBs, they are discussed separately.

3.1. Micro-TLBs

Micro-TLBs were simulated with sizes varying from one to eight entries and *Least Recently Used* (LRU) or *Least Recently Replaced* (FIFO) replacement algorithms. Page sizes of 4K and 16K bytes were used for the simulations.

Figure 1 is a plot of the simulation results for the *eqntott* benchmark, illustrative of micro-TLB behavior. The number of entries in the micro-TLB varies along the *x* axis. The *y* axis is scaled in instructions per miss, the reciprocal of the miss rate. Plots of instructions per miss are used because they illustrate interesting behavior more clearly than plots of miss rate, with data points corresponding to good performance towards the top of the graph, poor performance toward the bottom, and points spaced in a meaningful way, rather than disappearing along the X axis.

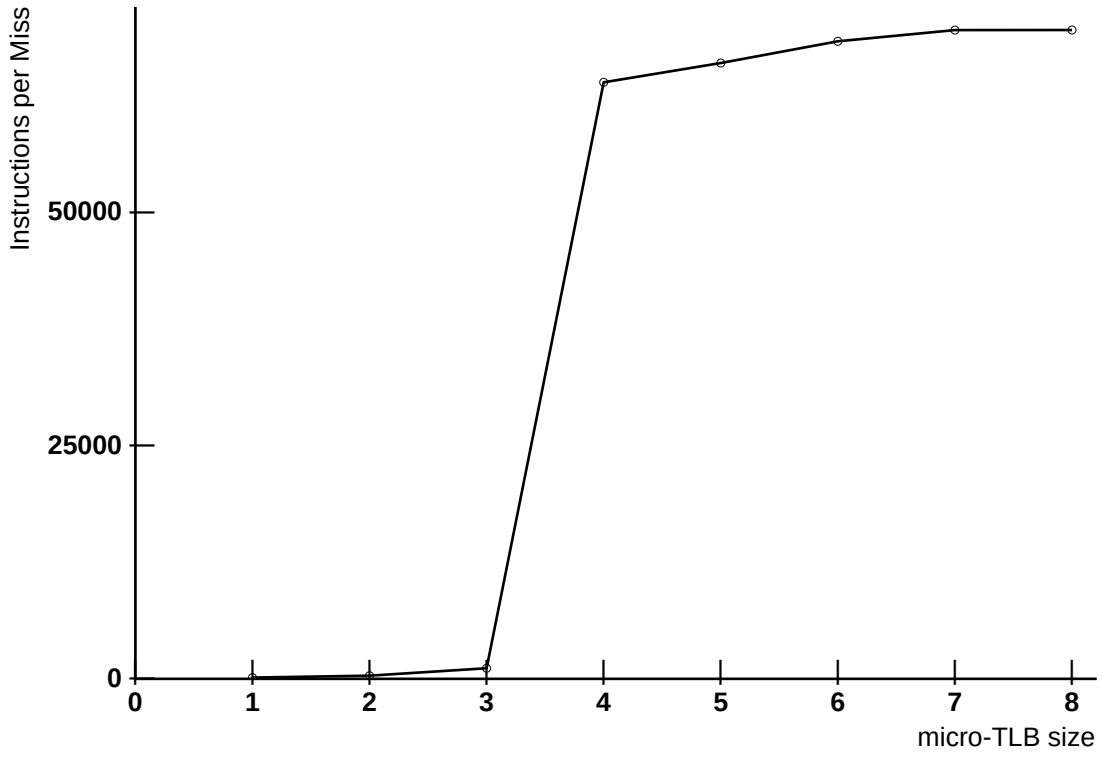

Figure 1: *Eqntott* **micro-TLB behavior**
FIFO replacement, 4K byte pages

There are three general regimes of behavior to be observed. Notice the flatness of the curve to the left side of the graph. In this region, thrashing is occurring. The number of micro-TLB entries is well under the number of instruction pages in the working set of the program, and consequently micro-TLB performance is relatively poor: 442 instructions per miss with two micro-TLB lines, 1490 instructions per miss with three.

After the flat part of the curve comes a region where performance improves rapidly. For *eqntott* this occurs between three and four micro-TLB entries. Lastly comes another relatively flat region, where the micro-TLB has enough entries to map the entire working set of the program. In this region, additional micro-TLB entries do little to improve performance.

Figure 2 illustrates micro-TLB performance for the SPECmarks. Notice a log scale is used for the *y* axis of this graph, while the *y* axis in Figure 1 used a linear scale. Although the log scale tends to obscure the different domains of behavior, it makes it possible to compare all the SPEC-

marks on the same graph. If a micro-TLB miss penalty of 3 cycles is assumed and the average number of instructions per miss is 333 or less, then about one machine cycle per one hundred instructions (i.e. 0.01 cycles per instruction - CPI) is lost to micro-TLB misses. With one micro-TLB entry, more than half the SPECmarks have this much of a penalty. With a two entry micro-TLB, 40% are at this penalty level.

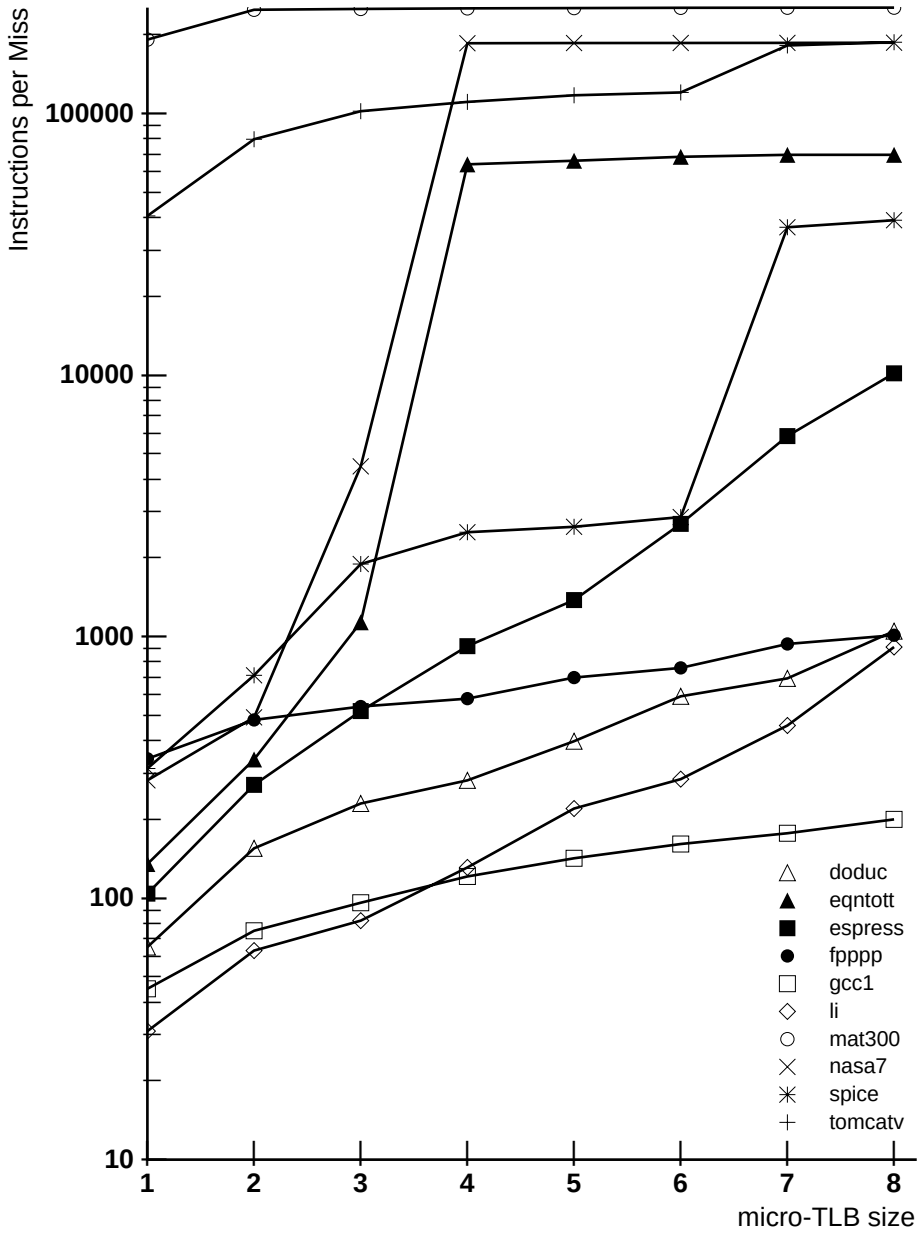

Figure 2: SPECmark micro-TLB Behavior
FIFO replacement, 4K byte pages

Four of the SPECmarks have particularly mediocre micro-TLB performance. *Fpppp* and *doduc* are both floating-point benchmarks. Both of them are noted for their poor instruction

cache performance, predictive of the observed micro-TLB behavior. The miss rate for *fpppp* for a simulated 4K byte instruction cache with 16 byte lines is 23%, due to its particularly long basic blocks (an average of 130 instructions per branch for the entire run). Computation in *doduc* is spread over a large number of procedures, and has a simulated 4K byte I-cache miss rate of 11%.

Two language processing programs, *gcc*, the Gnu C compiler, and *li*, a lisp interpreter, have the worst micro-TLB performance. The I-cache miss rates for *gcc* and *li* are 10% and 2%, respectively. Their micro-TLB behavior is explained in considering the structure of the programs. *Gcc* for example, has a large amount of code and it tends to make many nested procedure calls. We believe that the observed behavior results from there being eight or more procedures involved in most of *gcc*'s localities, and that these procedures tend to be spread over more than eight pages.

The SPECmarks were also simulated for micro-TLBs using an instruction page of size 16K bytes. The results are shown in Figure 3. At this page size, with 7 micro-TLB entries, the working set of virtually all the programs appears to have been reached, with the exception of *gcc*. The amount of memory fragmentation induced by the change to 16K byte pages can be inferred from the change in TLB resource demands of the programs. For example, *eqntott* uses 4 $\times$ 4K = 16K bytes of instruction memory with 4K pages, and 48K bytes with 16K byte pages. Spice grows from 28K to 64K bytes. Fragmentation for 16K byte pages could be reduced with compilers and loaders that used heuristics or feedback information to locate the most active code adjacent in memory.

One issue which turned out to be uninteresting is the effect of context switches on the micro-TLB. One alternative in simulations is to be pessimistic about preserving the contents of the micro-TLB, and flush it after every context switch. An optimistic alternative is to preserve the contents of the micro-TLB between context switches. It was found in the optimistic simulations that behavior improved in regions of already good behavior, but that there was little change where behavior was poor. In regions of good behavior, all micro-TLB misses are compulsory. In the optimistic simulations, only one compulsory miss is taken for each referenced page. In the pessimistic simulations, the micro-TLB is flushed, and so a compulsory miss occurs for every context switch. Preserving the micro-TLB reduces the number of compulsory misses, hence improving good behavior. During poor micro-TLB behavior, the vast majority of misses are not compulsory but capacity misses, and so changing the number of compulsory misses has a negligible effect on the overall results. The simulations used to generate data for this paper use the pessimistic model.

Another micro-TLB design parameter that was considered is replacement policy. For two entries, LRU is easily implemented in hardware. For more than two entries, hardware LRU becomes more difficult. An interesting alternative for micro-TLBs is *least recently replaced*. This has the advantage of a relatively straightforward hardware implementation as a first-in first-out queue. *FIFO* is used in this exposition to refer to this replacement policy.

Figure 4 shows relative performance of LRU and FIFO replacement policies for the ten SPECmarks. LRU is uniformly better, but not by a large amount. Again, the log scale makes it possible to compare all the benchmarks in the same graph, although it tends to obscure the real difference in performance, sometimes as much as a factor of two. Nonetheless, FIFO performance is always comparable with LRU performance when the overall affect on CPI is con-

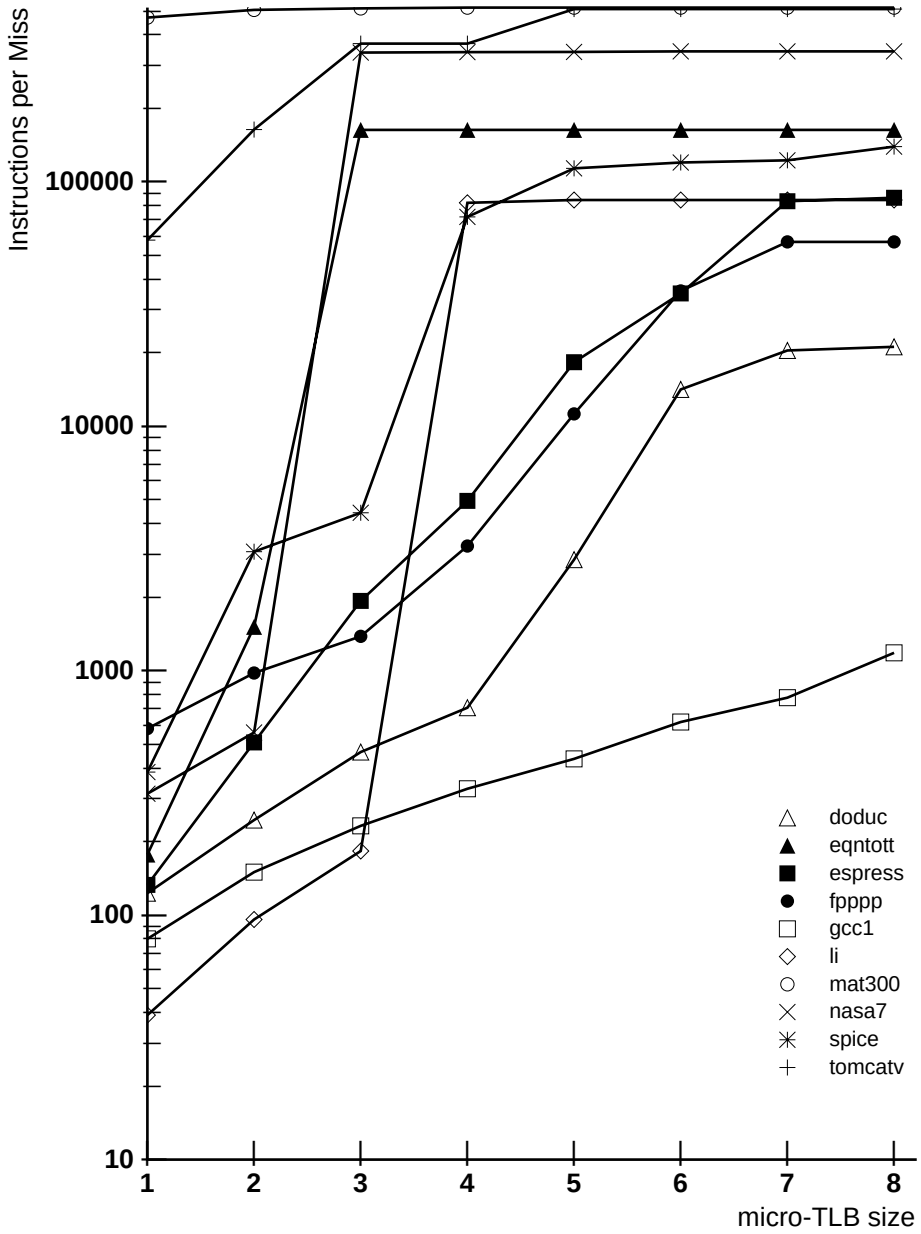

Figure 3: SPECmark micro-TLB Behavior
FIFO replacement, 16K byte pages

sidered, establishing it as an interesting replacement policy for micro-TLBs of size greater than two.

Simulations were also run to compare LRU, FIFO and Random replacement policies in full size data TLBs. It was found that FIFO performs uniformly better than Random, failing only in pathological worst case situations. For most machines with hardware to support Random replacement, FIFO can be easily implemented, although it is expected that Random replacement will be used to avoid pathologic worst case behavior.

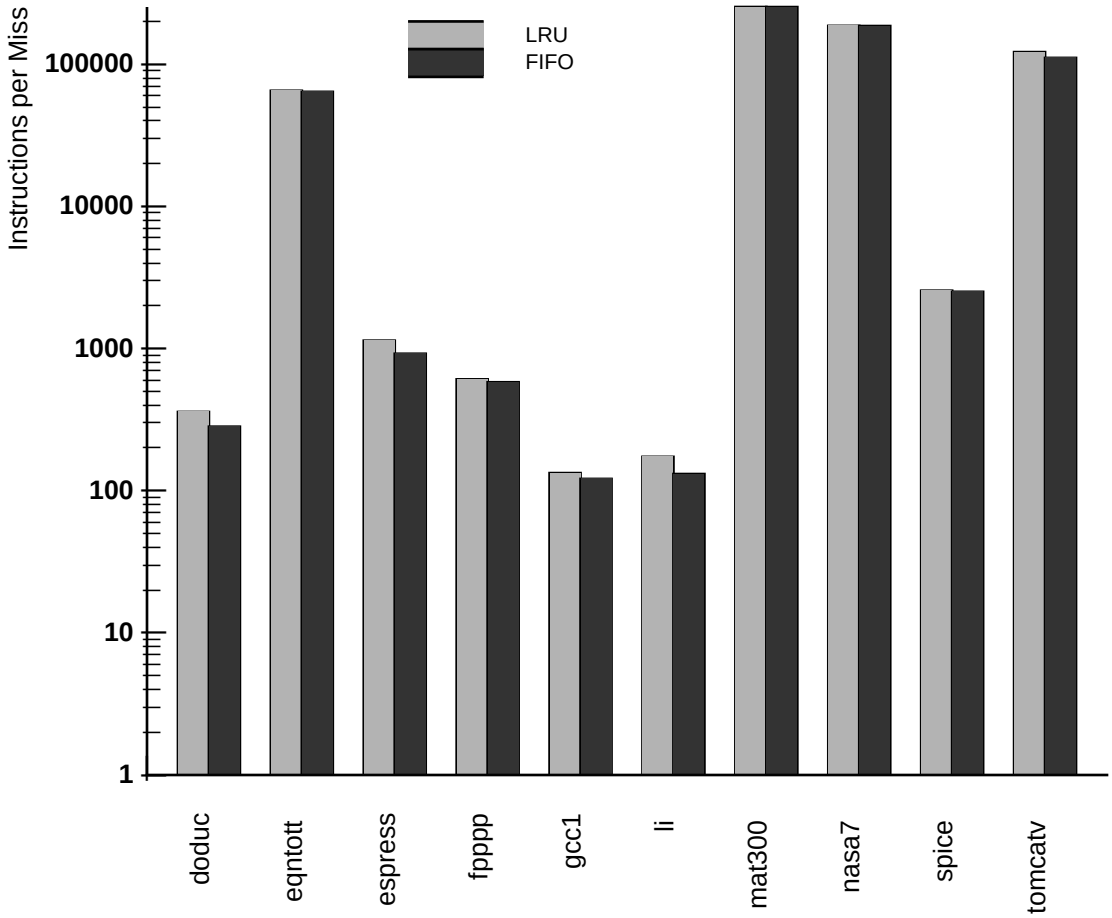

Figure 4: LRU vs. FIFO Replacement
4 entry micro-TLB, 4K byte pages

Table 1 shows miss rates for the ten SPECmarks, plus *tree* and *magic*. Under the assumption of a 3 cycle miss penalty, figures in bold-face indicate where the micro-TLB penalty is greater than 0.01 CPI. Again, with 4K byte pages, it is observed that most of the SPECmarks achieve reasonable performance with a two entry micro-TLB, although improvement is observed with larger sizes. Figure 5 shows how data from Table 1 can be used to estimate CPI contribution for a given SPECmark, micro-TLB configuration, and miss penalty.

Note that the micro-TLB performance for *tree* is particularly poor. With 4K byte pages, a two entry micro-TLB absorbs about 0.10 CPI. Micro-TLB performance for *tree* begins to improve rapidly beyond four micro-TLB entries. There are two effects that could conceivably be contributing to the degraded performance: the working set size of the computation, and conflicts between the garbage collector and the rest of the computation. The garbage collector and the program behave as independent co-routines or threads in the single address space. In the case of *tree*, most of the observed miss rate is due to locality properties of the compiled Scheme code, as co-routine exchanges between the garbage collector and the program execution are much too

	1×4KB page 1×16KB page	2×4KB page 2×16KB page	4×4KB page 4×16KB page	8×4KB page 8×16KB page
doduc	**0.015415**	**0.006433**	**0.003552**	0.000953
	0.008111	**0.004078**	0.001416	0.000047
eqntott	**0.007415**	0.002951	0.000016	0.000014
	0.005647	0.000661	0.000006	0.000006
espress	**0.009610**	**0.003687**	0.001089	0.000099
	0.007500	0.001966	0.000202	0.000012
fpppp	0.002936	0.002085	0.001727	0.000988
	0.001723	0.001022	0.000309	0.000018
gcc1	**0.021989**	**0.013269**	**0.008288**	**0.005010**
	0.012539	**0.006648**	0.003042	0.000844
li	**0.031792**	**0.015781**	**0.007661**	0.001097
	0.025473	**0.010456**	0.000012	0.000012
mat300	0.000005	0.000004	0.000004	0.000004
	0.000002	0.000002	0.000002	0.000002
nasa7	**0.003544**	0.002037	0.000005	0.000005
	0.003180	0.001788	0.000003	0.000003
spice	0.003199	0.001407	0.000399	0.000026
	0.002598	0.000326	0.000014	0.000007
tomcatv	0.000025	0.000013	0.000009	0.000005
	0.000017	0.000006	0.000002	0.000002
ccom	**0.030306**	**0.016114**	**0.007844**	**0.004098**
	0.017705	**0.008653**	0.001761	0.000174
sed	**0.033778**	**0.008355**	0.000070	0.000064
	0.009710	0.000031	0.000031	0.000031
tree	**0.074999**	**0.034553**	**0.017515**	0.001264
	0.062228	**0.021160**	0.002427	0.000215
magic	**0.014811**	**0.007654**	0.002853	0.000606
	0.013080	**0.006197**	0.001599	0.000287

Table 1: SPECmark FIFO micro-TLB Miss Rates

2 entry FIFO micro-TLB with 4K byte page
3 cycle miss penalty

CPI Contribution = 3 * 0.0133 = 0.0399 CPI

Figure 5: Estimating micro-TLB CPI Contribution for *gcc*

infrequent[1] to account for the observed miss rates.

Although multi-thread effects were not important with *tree*, it is worth noting that, in as much as threads tend to execute in independent code localities, tightly coupled threads executing in a single address space will lead to degraded micro-TLB performance. This effect will be exaggerated with very small micro-TLBs.

[1]The *tree* execution takes about 147 seconds, of which 2 seconds are spent collecting garbage. Garbage collections were observed to occur every 8-15 seconds.

3.2. Instruction TLBs

We simulated instruction TLBs with sizes in powers of two from 8 to 64 entries, and page sizes in powers two from 4K to 64K bytes. In examples used to illustrate TLB performance, an approximation of 100 cycles is used for the TLB miss penalty. It is assumed that a TLB miss will cause two memory references, corresponding to a page table lookup, and that the latency of these memory references will dominate the miss penalty. The penalty of 100 cycles corresponds to a futuristic microprocessor with a cycle time of under five nanoseconds, and is meant to be somewhat less than the time for two references to main memory and somewhat more than the time for two references to an off chip cache. Under these assumptions, with TLB performance of 10000 instructions per miss, 0.01 CPI is lost to the TLB. This is used as a somewhat arbitrary lower bound on reasonable TLB performance.

The behavior of single-benchmark workloads in instruction-only TLBs is mostly uninteresting, as *gcc* is the only SPECmark that presents a significant demand on resources. Figure 6 illustrates the behavior for *gcc*. Again log scales are used. Solid lines connect TLB configurations with the same page size. Dashed lines connect TLBs that map the same amount of memory. The amount of memory mapped by a TLB will be referred to as its *mapping size*, to discriminate between that measure of size and others, such as the number of lines in a TLB.

As with the micro-TLB, there are three regimes of behavior to be observed. The placement of data points is more compact in the lower portion of the graph, with relatively little improvement for larger TLB configurations. This represents thrashing, trying to operate with TLB resources well below the working set size of the program. In the next region, performance improves quickly as working set size is approached. This figure shows that *gcc* approaches the TLB resources to map its working set with an instruction TLB mapping size of 512K bytes. Once the working set size of the program has been reached, increasing the mapping size has a reduced effect on performance, and the points again become more closely spaced. Such behavior is observed at the top of the graph. These three regimes of behavior become more pronounced for shared and data TLBs, as in Figure 8.

Note that in the lower part of the graph, the dashed lines that connect TLBs with the same mapping size tend to slope upward slightly, while at the top of the graph they slope down. An application that uses sparse, non-contiguous data tends to have better TLB performance with more smaller pages of memory than with a few larger pages. Such behavior is observed in the lower part of the graph. In the upper part of the graph, the dashed lines tend to slope down, hence better performance with fewer larger pages. As a TLB becomes large enough to map all of a program's working set, smaller pages mean that TLB misses occur for each of several small pages, rather than once for a single large page. Similar behavior is observed when a program accesses contiguous data, as in Figure 8.

Figure 7 illustrates instruction TLB performance for the multi-task mix. For this workload, the flat dashed lines suggest that TLB performance is determined entirely from mapping size, and that the configuration of page size and number of entries has little effect. TLB performance crosses the 10000 instruction per miss performance boundary at a mapping size of 512K bytes, and appears to have reached the working set size for mapping sizes over 2 megabytes.

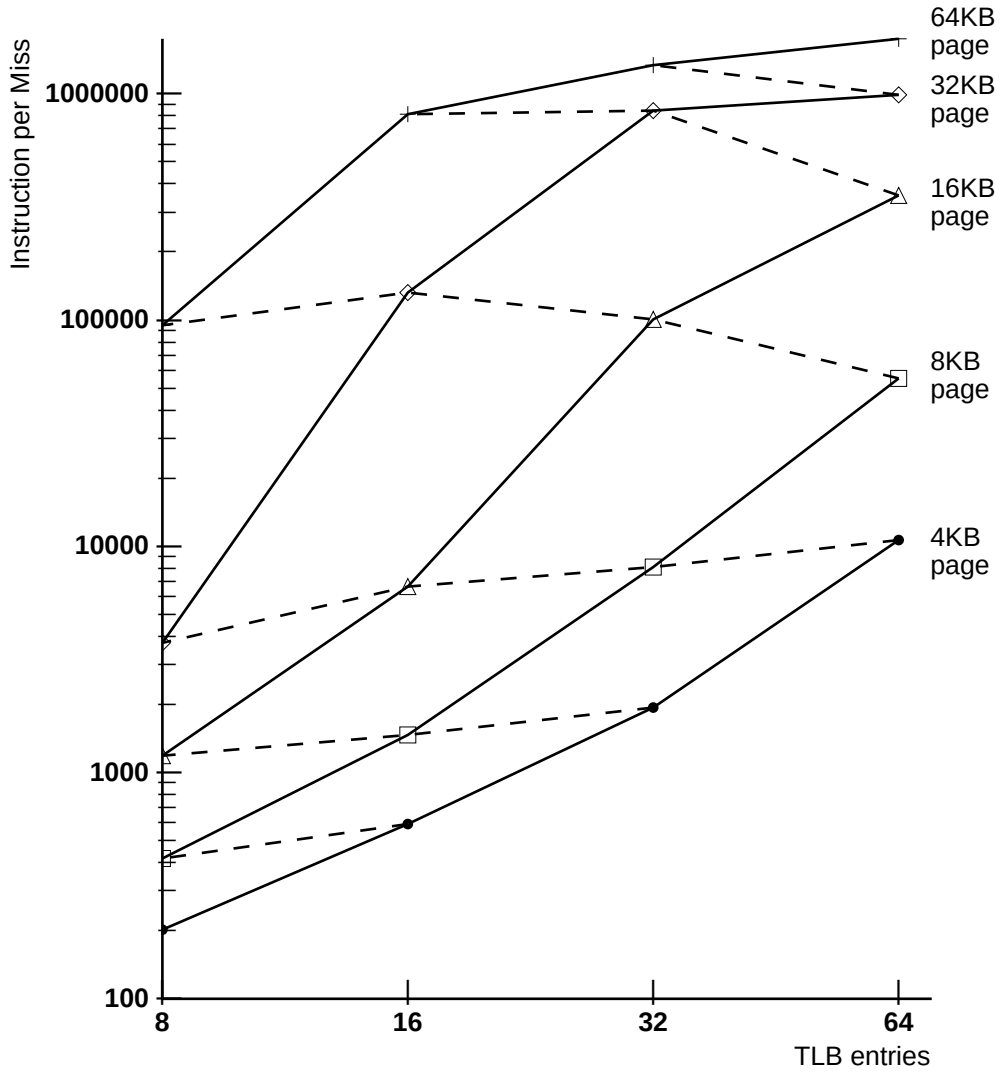

Figure 6: *Gcc* **Instruction TLB Behavior**
Random Replacement, Fully Associative

Table 2 gives miss rates for selected benchmarks for a number of instruction-only TLB configurations. The benchmarks not included have very low miss rates.

Our experiments suggest that an instruction TLB with 16 entries and 16K byte pages is adequate for most application workloads. One shortcoming of these experiments is that they do not consider the effects of operating system instruction references. It is known that system code generally exhibits worse instruction locality than applications. When designing an architecture to support a specific operating system, this should be a consideration.

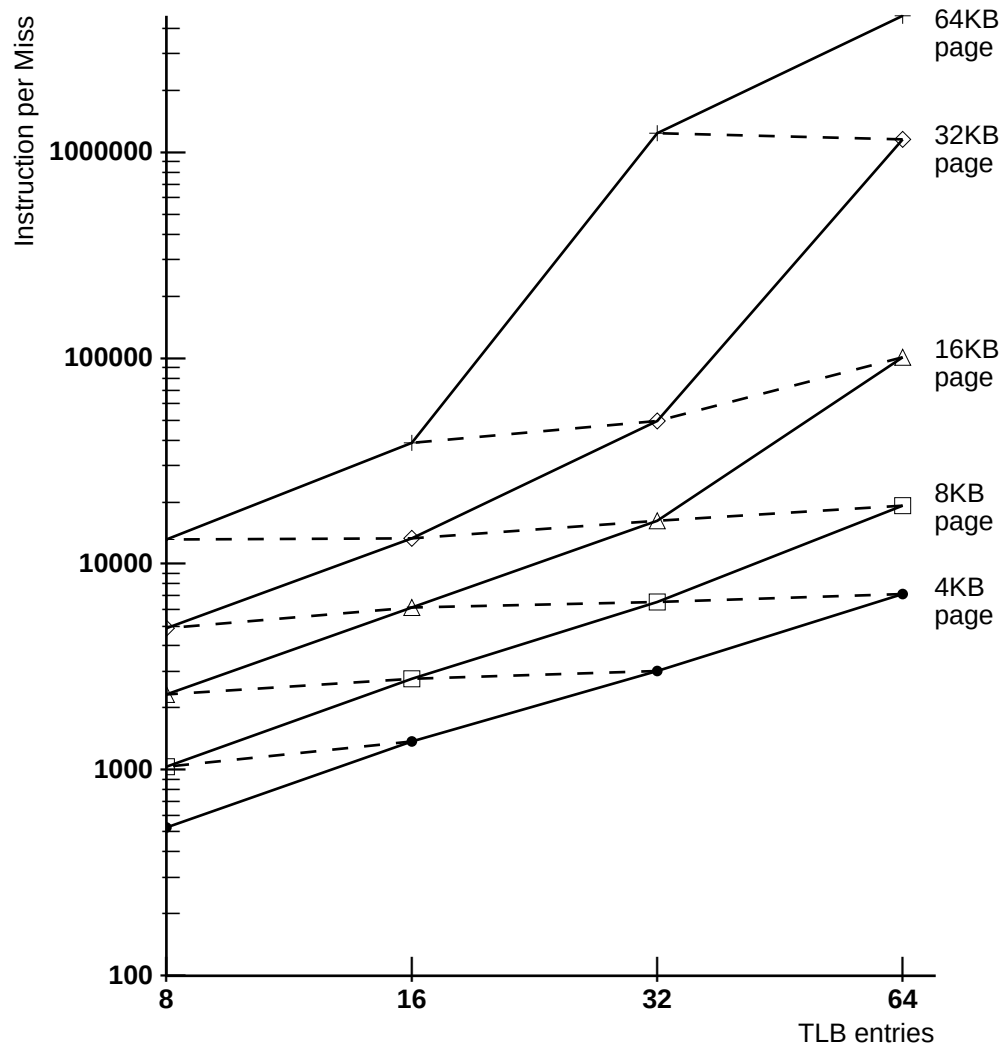

Figure 7: Multi-task instruction TLB behavior
Random Replacement, Fully Associative

	16x4KB	32x4KB	64x4KB	16x16KB	32x16KB
gcc1	**0.001688**	**0.000516**	0.000094	0.000151	0.000010
magic	**0.000275**	0.000059	0.000000	0.000041	0.000000
tree	**0.000309**	0.000000	0.000000	0.000000	0.000000
ccom	**0.001969**	0.000032	0.000003	0.000001	0.000001
mixA	**0.000728**	**0.000331**	**0.000140**	**0.000163**	0.000062

Table 2: Instruction TLB Miss Rates
Random Replacement, Fully Associative

4. Data and Shared TLB Results

Data-only TLBs and shared TLBs behave similarly. Most programs have well behaved instruction reference patterns, so in the shared case behavior is dominated by data references. Although the following discussion is in terms of shared TLBs, the analysis can be applied to the data-only case as well. Shared TLBs were simulated with sizes in powers of two from 32 to 256 entries, page sizes in powers of two from 4K to 64K bytes, and with LRU and Random replacement policies. All TLBs simulated were fully associative.

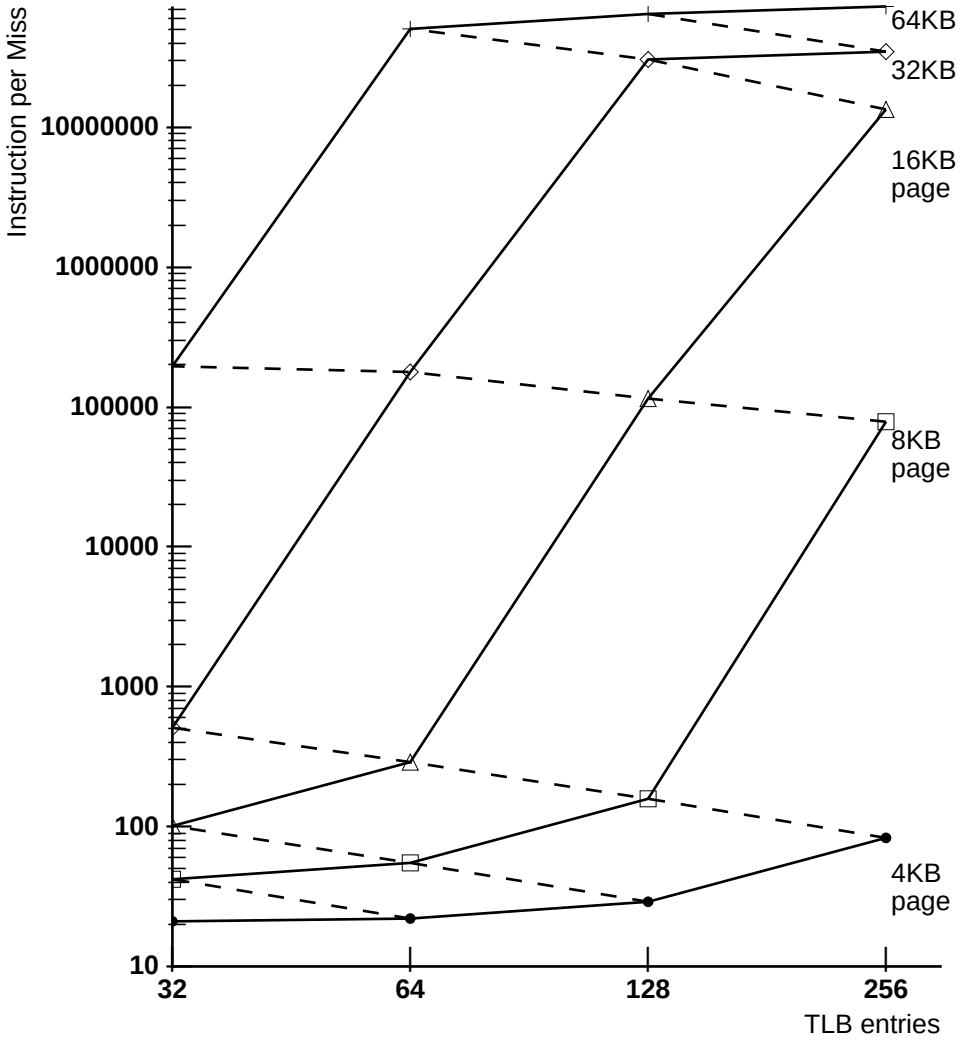

Figure 8: *Mat300* **Shared TLB Behavior**

Figure 8 shows the shared TLB behavior for *mat300*, which was found to be the SPECmark with the worst TLB performance. Note that the TLBs in this figure are a factor of four larger than those considered with instruction TLBs, ranging from 32 to 256 entries as opposed to 8 to 64 entries. With a miss penalty of 100 cycles and a 64 entry TLB with 4K byte pages, *mat300* spends 5 CPI on the TLB. *Mat300* does matrix operations on three matrices with a total size of

approximately 2.5 megabytes. The contents of these three matrices are accessed in regular patterns, sometimes sequentially and sometimes stepping by columns. This explains the poor behavior when the TLB maps less than 2.5 megabytes, and rapid improvement as that barrier is reached and surpassed. Observe that lines connecting TLBs with the same mapping size always slope down, consistent with the observation that *mat300* accesses its data contiguously.

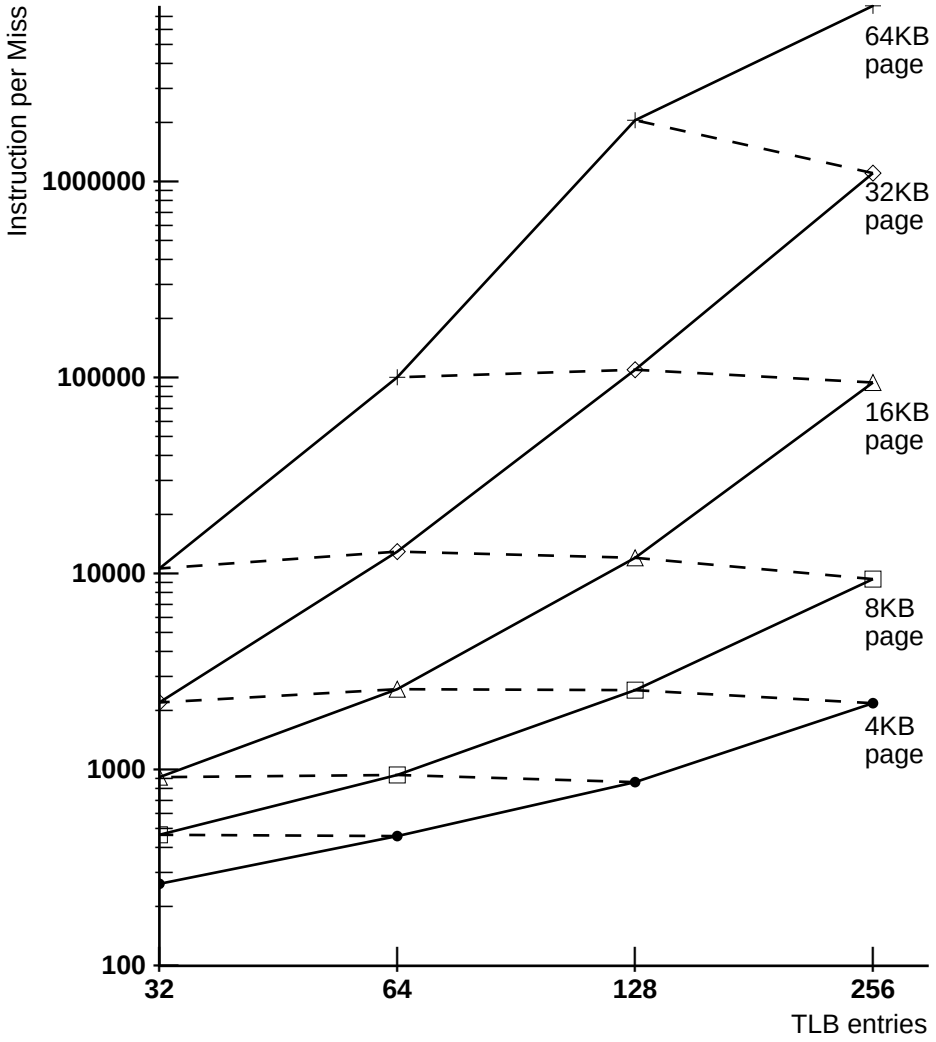

Figure 9: *Tree* **Shared TLB Behavior**
Random Replacement Fully Associative

Figure 9 shows the TLB behavior for *tree* running with a 10 megabyte heap. Memory is allocated from 5 megabytes of the heap, while the other 5 megabytes is reserved for garbage collection. As expected, performance improves steadly through a mapping size of 5 megabytes (16k pages × 256 entries), after which the rate of improvement begins to diminish, shown by the downward sloping lines connecting configurations with equal mapping sizes. Note that below the 5 megabyte boundary, the lines connecting TLBs with the same mapping size are nearly

horizontal. This indicates that TLB performance for this benchmark depends exclusively on the mapping size. Other TLB parameters are unimportant.

As a last illustration of shared TLB behavior, Figure 10 shows a plot for the multi-task mix. Several conclusions are immediate. First, as for *tree*, the mapping size of the TLBs is the dominant factor in performance. Only after a mapping size of eight megabytes do the dashed lines stop looking horizontal. Also, most of the plot is fairly compact. Data points become less compact beyond a mapping size of eight megabytes, as the working set size is approached. Lastly, for all TLBs with a mapping size of one megabyte or less, assuming a 100 cycle miss penalty, at least 0.05 CPI is lost to the TLB, a significant performance penalty. This suggests that if a machine with a TLB is to execute such a workload efficiently, the TLB must have a significantly larger mapping size.

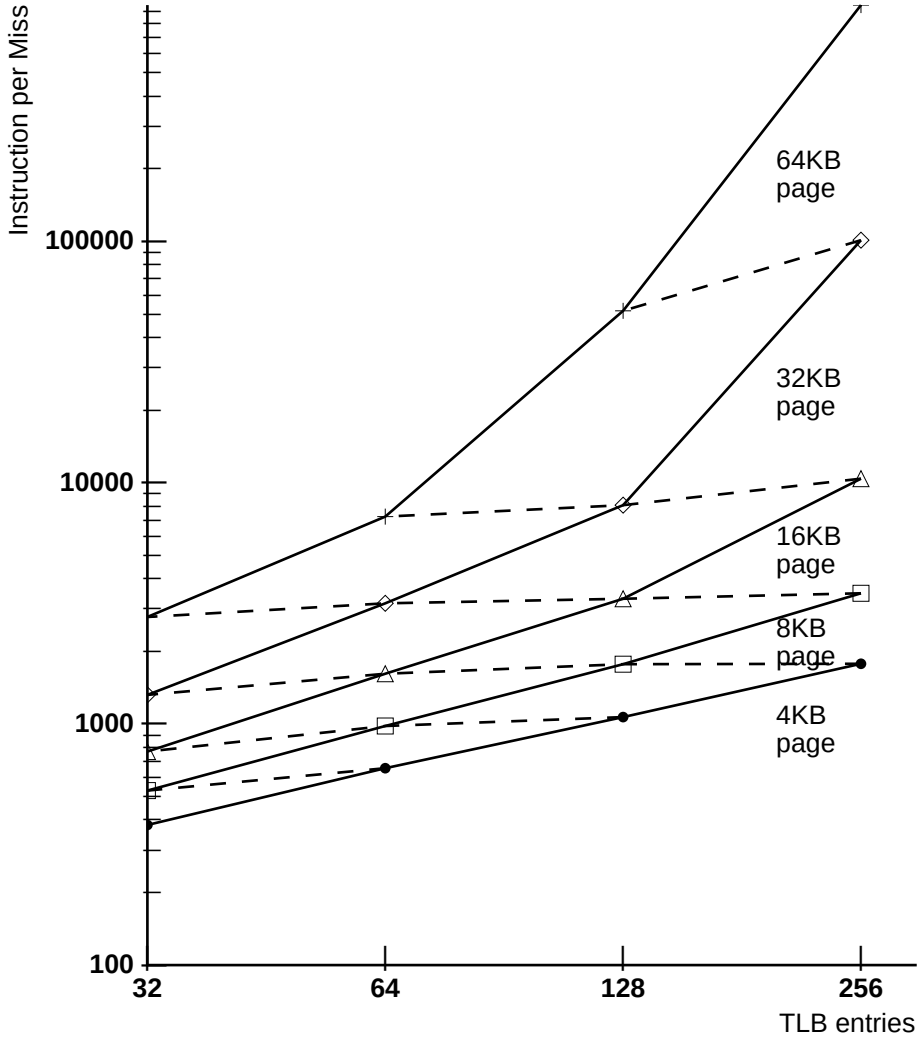

Figure 10: Multiprocess Mix TLB Behavior
Random Replacement, Fully Associative

Table 3 shows TLB miss rates for the ten SPECmarks, plus several other workloads of interest. This table is highlighted to indicate where the TLB penalty is more than 0.01 CPI. With the exceptions of *nasa7* and *mat300*, both of which are oriented towards scientific/vector machines, the SPECmarks perform better than this with a 64x16k TLB, suggesting that such a configuration provides adequate TLB performance. The smaller configurations don't perform as well. This suggests that sometime in the near future, TLBs with larger mapping sizes will be needed, especially for machines running numeric programs.

Table 4 gives miss rates for a number of data-only TLB configurations.

	64x4KB	128x4KB	256x4KB	64x16KB	64x32KB
doduc	0.000014	0.000000	0.000000	0.000000	0.000000
eqntott	**0.000410**	**0.000162**	0.000002	0.000020	0.000000
espress	0.000004	0.000001	0.000001	0.000000	0.000000
fpppp	0.000001	0.000000	0.000000	0.000000	0.000000
gcc1	**0.001143**	**0.000244**	**0.000132**	0.000053	0.000013
li	0.000000	0.000000	0.000000	0.000000	0.000000
mat300	**0.049097**	**0.036378**	**0.012669**	**0.003461**	0.000006
nasa7	**0.016526**	**0.011955**	**0.005183**	**0.002024**	0.000000
spice	0.000052	0.000005	0.000000	0.000000	0.000000
tomcatv	**0.000151**	**0.000138**	0.000121	0.000033	0.000012
ccom	**0.000283**	0.000009	0.000006	0.000005	0.000002
sed	0.000017	0.000017	0.000015	0.000006	0.000005
tree	**0.004683**	**0.002749**	**0.001015**	**0.000224**	0.000069
magic	**0.000644**	**0.000139**	0.000003	**0.000106**	0.000086
mixA	**0.001527**	**0.000937**	**0.000563**	**0.000620**	**0.000316**

Table 3: Shared TLB Miss Rates
Random Replacement, Fully Associative

	64x4KB	128x4KB	256x4KB	64x16KB	128x16KB
doduc	0.000000	0.000000	0.000000	0.000000	0.000000
eqntott	**0.000355**	**0.000140**	0.000002	0.000016	0.000000
espress	0.000001	0.000001	0.000000	0.000000	0.000000
fpppp	0.000000	0.000000	0.000000	0.000000	0.000000
gcc1	**0.000152**	**0.000104**	0.000067	0.000022	0.000014
li	0.000000	0.000000	0.000000	0.000000	0.000000
mat300	**0.045666**	**0.034275**	**0.012050**	**0.003460**	0.000008
nasa7	**0.016521**	**0.011947**	**0.005178**	**0.002022**	0.000001
spice	0.000034	0.000000	0.000000	0.000000	0.000000
tomcatv	**0.000149**	**0.000137**	0.000120	0.000033	0.000021
ccom	0.000005	0.000004	0.000003	0.000001	0.000001
sed	0.000011	0.000010	0.000009	0.000005	0.000004
tree	**0.001804**	**0.001002**	**0.000420**	**0.000288**	0.000076
magic	**0.000270**	0.000038	0.000003	0.000059	0.000002
mixA	**0.001358**	**0.000866**	**0.000511**	**0.000546**	**0.000258**

Table 4: Data TLB Miss Rates
Random Replacement, Fully Associative

Figure 11 compares miss rates for shared and split TLBs. All TLBs use 64 entries and 4K byte pages. For all but *gcc* and the multi-task mix, the instruction miss rates are inconsequential. Note that the sum of the split instruction and data miss rates is generally less than the miss rate for the shared TLB. This difference represents competition for TLB entries between instruction

and data references. This comparison is not meant to suggest two specific implementation alternatives, as the split TLBs illustrated use twice the resources of the shared TLB.

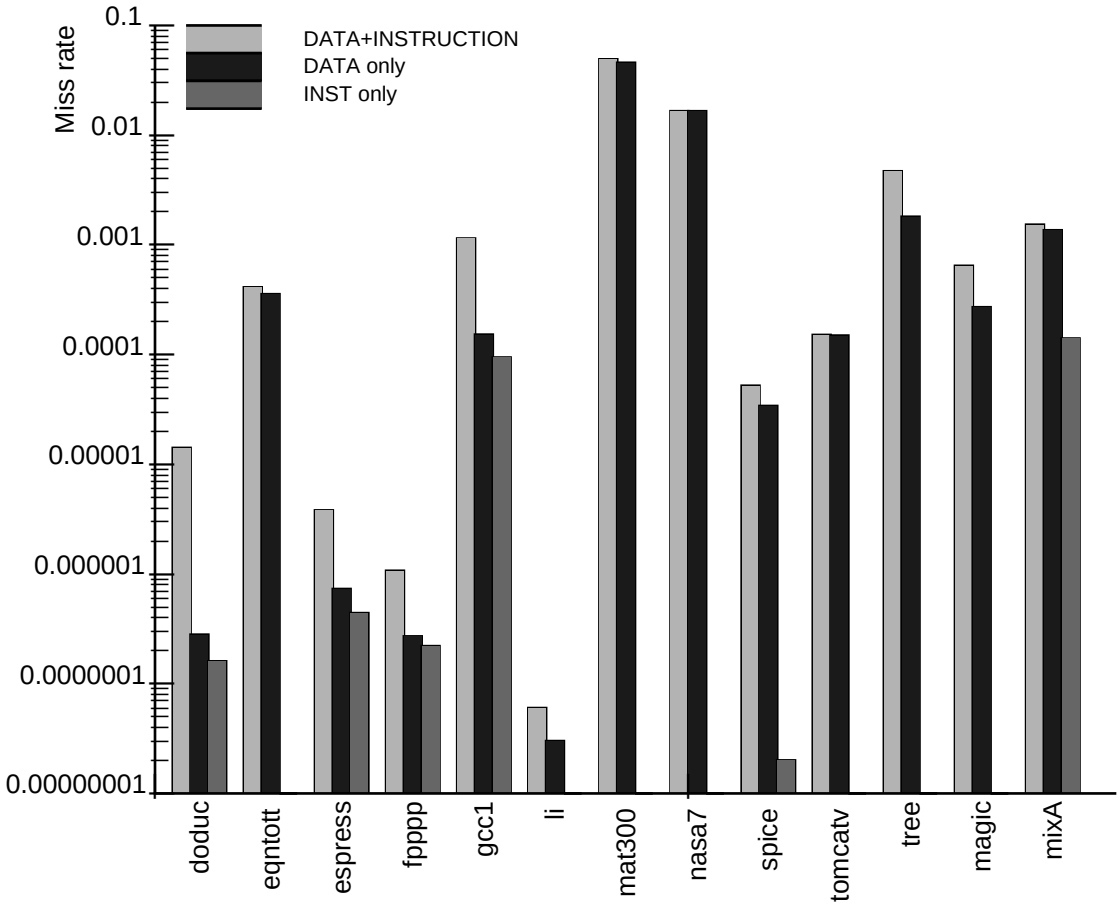

Figure 11: Shared vs. Split TLBs 64 entries, 4KB pages
Random Replacement, Fully Associative

5. Variable Size TLB Entries

An interesting question for future work is how to make use of the variable size TLB entries that have appeared in recent architectures [5, 7]. Maps of the dynamic patterns of memory access are useful to understand this problem. Figure 12 shows the pattern of data memory accesses for *mat300*. Page address varies in the x dimension, from 0x10000000 on the left to 0x1021e000 on the right, a range of about 2.2 megabytes. Instruction count (i.e. time) varies along the y dimension, ranging from 0 at the top to 2.63 billion at the bottom. The darkness of each square corresponds to the number of accesses per 16K byte page during a 1000000 instruction interval.

The three matrices used by *mat300* are clearly visible from the usage patterns in the address space. The compactness and predictability of the *mat300* accesses show that the use of larger

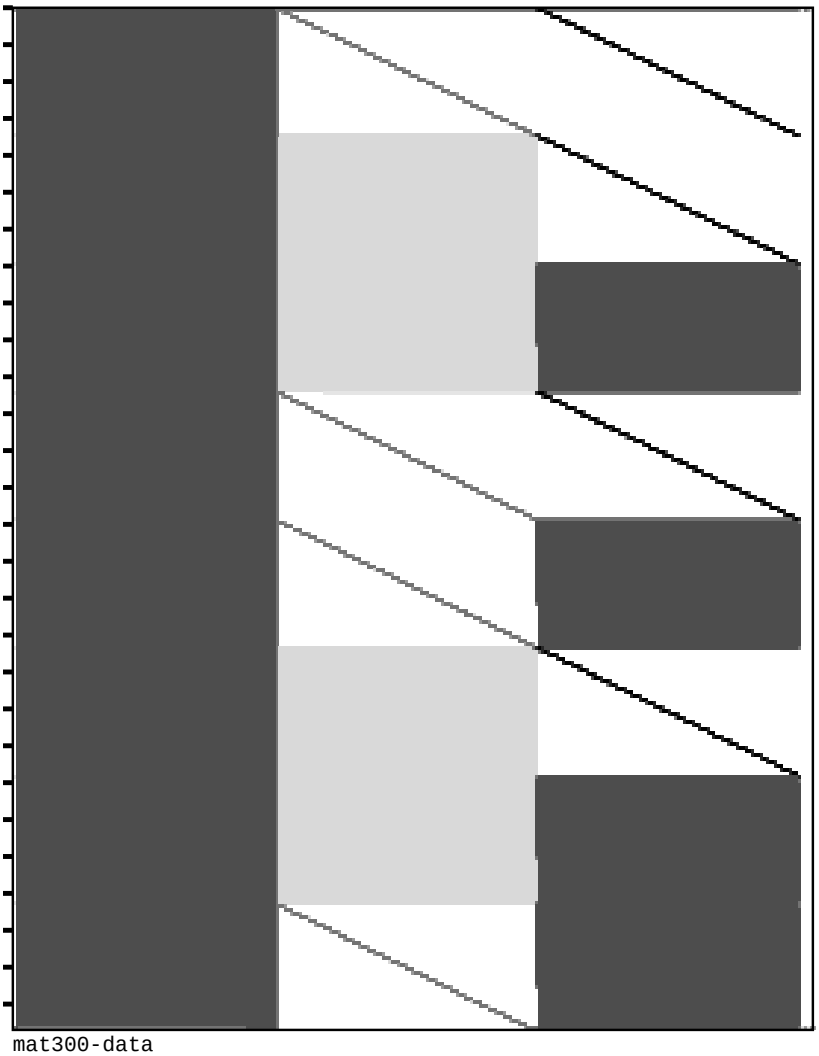

Figure 12: *mat300* **Data Memory Access Patterns**

pages could virtually eliminate TLB misses, provided that adequate memory resources were available.

Tree, the lisp benchmark, also shows interesting data reference patterns, illustrated in Figure 13. Note that a page size of 64K bytes was used. The address space represented in this figure is about 11 megabytes. The descending staircase pattern shows the behavior of the memory allocator as it walks across the heap. Solid vertical bands show where garbage collection has compacted the heap into frequently accessed regions. The pattern of memory references for *tree* is sparse relative to *mat300*. This, along with the size of the address space, suggests that lisp workloads such as *tree* are relatively poor candidates for variable size pages.

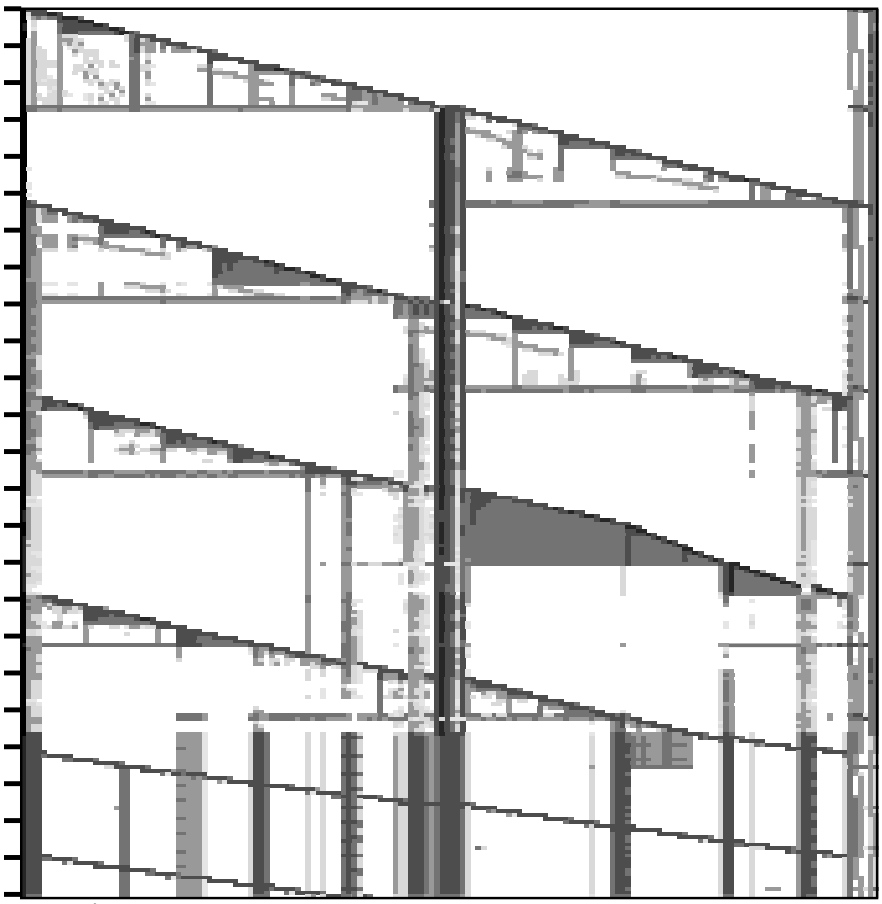

Figure 13: *tree* **Data Memory Access Patterns**

Interesting patterns of reference are the exception rather than the rule in memory access patterns. Most of the benchmarks concentrate on a small number of unclustered pages, resulting in a few dark vertical bars from the top to the bottom of the map, with occasional horizontal excursions.

Figure 14 shows a map of instruction references for *gcc*. Each point represents one or more references to a 4K byte page during an interval of 100000 instructions. The address space spanned in this figure is 684K bytes, the largest text segment of any of the SPECmarks. The number of different pages touched during a single 100000 instruction interval illustrates clearly why *gcc* places high demands on the TLB. If variable size memory pages were to be used to improve *gcc* performance, the only solution would be load the entire program text into a contiguous segment.

For instruction references, compilers might use feedback information on performance critical applications to locate active text contiguously, making the use of a single larger TLB entry a more attractive option. Such techniques are more difficult to apply to data references, as heap

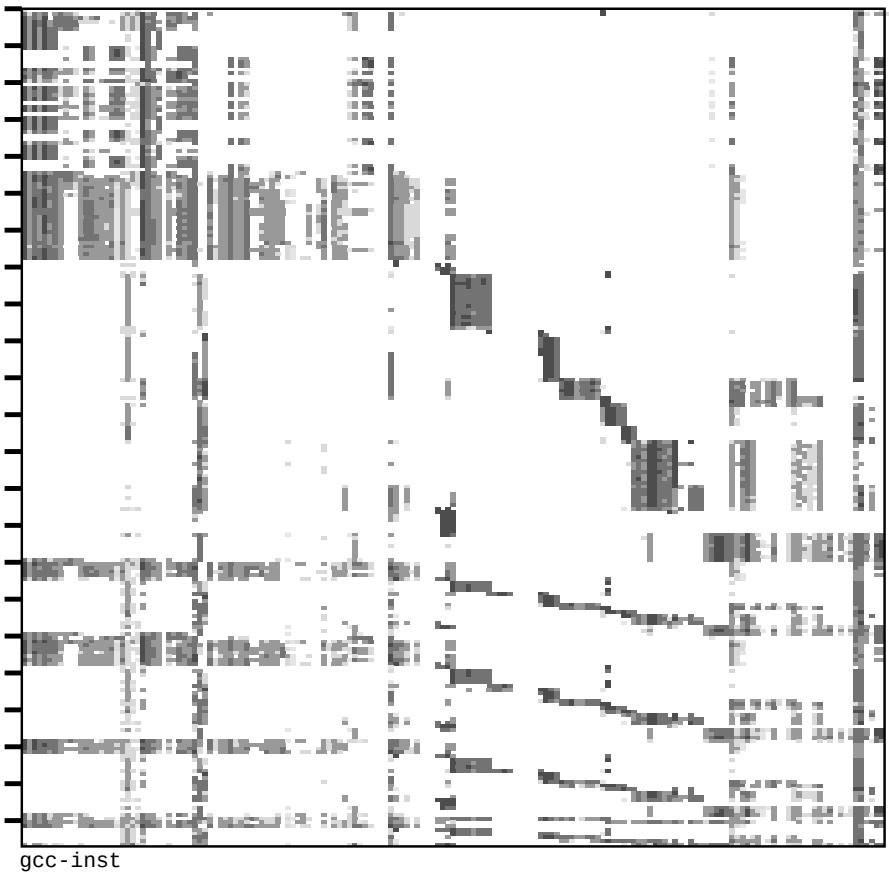

Figure 14: *gcc* **Data Memory Access Patterns**

allocated structures are allocated dynamically, and so their location is not under the control of the compiler. With the relocatable nature of lisp data, it might be possible to tune garbage collectors to improve the locality of reference. For uncollected memory allocation schemes, a tool using feedback information could make suggestions of how to order heap data allocation to improve contiguity of data.

6. Conclusions

This study has investigated the performance of one and two-level instruction TLBs, data TLBs, and shared TLBs, as well as analyzing the potential performance implications of variable-sized pages. In contrast to previous studies, this work concentrated on fully-associative TLB organizations and split instruction and data reference streams.

For instruction TLBs, programs such as *gcc* and *li* that make many nested calls to small procedures are the hardest to satisfy. For most of the SPECmarks, 4K byte pages and a two entry micro-TLB (whose misses are serviced in several cycles by a shared TLB) perform reason-

ably well. For example, with a 3 cycle micro-TLB miss penalty (i.e., assuming that the reference hits in the 2nd-level TLB) all SPECmarks except *gcc* and *li* incur a CPI of less than 0.03 due to microTLB misses. *gcc* and *li* can achieve this level of performance with 4-entry micro-TLBs, but incur a CPI penalty of about 0.06 with a 2-entry micro-TLB. A FIFO replacement policy performs almost as well as LRU for micro-TLBs.

In single-level instruction, data, and shared TLBs, TLB performance is usually dominated by how much memory is mapped. Single-level fully-associative instruction TLBs (or the second level of a two-level organization) with more that 32 entries, 4K byte pages, and a 100 cycle miss penalties incur CPIs of under 0.1 even for *gcc*. Performance on other benchmarks and with larger TLBs is better. With the larger capacities and miss penalties of full size instruction TLBs, multi-tasking and system effects also become important.

A data or shared TLB mapping 256K bytes in 4K byte pages (i.e., 64 entries) with 100 cycle miss penalty incurs 0.1 CPI or less for all of the SPECmarks except *nasa7* and *mat300*. Both of these are scientific/vector oriented programs with large data sets. Furthermore, column access (i.e., non-unit stride) can result in successive data references to successive pages, disastrous for TLB performance unless the entire data set is mapped at the same time. *nasa7* and *mat300* incur a CPI of 1.7 and 4.9, respectively, for the TLB parameters given above. This is not reduced to under 0.1 CPI for *mat300* until the TLB can map 2 megabytes (e.g., 256 entry TLB with 8K byte pages). Work with more demanding workloads suggests that future TLBs must map significantly more memory.

One way to increase the amount of memory mapped without requiring an unreasonably large number of TLB entries is the use of variable-sized pages. Memory access plots suggest that the use of very large pages (e.g., 256K byte or greater) for the data space of *mat300* and *tree*, and the instruction space of *gcc* could vastly reduce the size of the TLB required for good performance while decreasing its miss rate.

One significant shortcoming of this TLB analysis is the inability to consider operating system effects. We are currently involved in completing a new tracing system which includes system traces, with the intention of performing a thorough exploration of operating system memory behavior on modern RISC processors.

7. Acknowledgements

We would like to thank David Wall for keeping the compiler aspects of the tracing project in tiptop shape. We would also like to thank Joel Bartlett for consultations on Scheme. Thanks to Brian Bershad and John Ousterhout for their useful comments on this paper.

8. References

[1] J. F. Bartlett.
SCHEME->C: A Portable Scheme-to-C Compiler.
WRL Research Report 89/1, Digital Equipment Western Research Laboratory, 1989.

[2] Anita Borg, R.E. Kessler, Georgia Lazana, and David Wall.
Long Address Traces from RISC Machines: Generation and Analysis.
WRL Research Report 89/14, Digital Equipment Western Research Laboratory, 1989.

[3] Douglas W. Clark and Joel S. Emer.
Performance of the VAX 11/780 Translation Buffer: Simulation and Measurement.
ACM Transactions on Computer Systems 3(1), February, 1985.

[4] Douglas W. Clark, Peter J. Bannon, and James B. Keller.
Measuring VAX 8800 Performance with a Histogram Hardware Monitor.
In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 176-185. June, 1988.

[5] Daniel Dobberpuhl, et. al.
A 200Mhz 64b Dual-Issue CMOS Microprocessor.
In *The 39th International Solid-State Circuits Conference*, pages 106-107. IEEE Computer Society Press, February, 1992.
See also slide supplement.

[6] Norman P. Jouppi.
Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.
In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 281-289. May, 1989.

[7] Sunil Mirapuri, Michael Woodacre, and Nader Vasseghi.
The MIPS R4000 Processor.
IEEE Micro 12(4):10-22, April, 1992.

[8] J. Ousterhout, G. Hamachi, R. Mayo, W. Scott, and G.S. Taylor.
The Magic VLSI Layout System.
IEEE Design and Test of Computers 2(1):19-30, February, 1985.

[9] Steven A. Przybylski.
Cache Design: A Performance-Directed Approach.
Morgan-Kaufmann, San Mateo, CA, 1990.

[10] Alan Jay Smith.
Cache Memories.
ACM Computer Surveys 14(3):473-530, September, 1982.

[11] David A. Wood, et. al.
An In-Cache Address Translation Mechanism.
In *The 13th Annual Symposium on Computer Architecture*, pages 358-365. IEEE Computer Society Press, June, 1986.

[12] David A. Wood.
The Design and Evaluation of In-Cache Address Translation.
PhD thesis, Department of Computer Science, UC Berkeley, March, 1991.
Report Number UCB/CSD 90/565.

Table of Contents

List of Figures

List of Tables