ABSTRACT

KONIREDDYGARI, SREEKANTH. Measuring and Improving TLB Performance for Linux GUI Applications. (Under the direction of Dr. Edward Gehringer and Dr. William Cohen).

Modern GUI applications rely on a large number of dynamically linked shared libraries to reduce the applications' memory footprint and to avoid recompilation when newer versions of the shared libraries become available. However, each shared library needs separate memory pages, and the pages are laid out in virtual memory in essentially random fashion. This leads to increased conflict misses in the translation lookaside buffer (TLB) and diminishes the application's performance. Previous work has measured TLB performance using standard benchmarks, which consist of applications with statically-linked libraries. Statically-linked programs tend to use fewer libraries, and don't require separate pages for each library; hence they place much less stress on the TLB.

We added a TLB simulator to *Valgrind*, a Linux binary instrumentation tool, and used *Dogtail*, a testing framework for GUI applications, to measure TLB behavior for several dynamically linked GUI applications: the document-viewing application Evince, the word processor *Abiword*, the image viewer *Gthumb*, and the calculator *Gcalctool*. Analysis of TLB miss data from the *Valgrind* instrumentation showed that some shared library pages repeatedly conflicted with pages from other shared libraries for TLB entries. Moving just a single highly-contended shared library to a different place in the virtual address space reduced TLB conflict misses up to 14.7%. Average reductions across applications were greater in larger TLBs, with the highest average reduction, 7.5%, in the 512-entry TLB, the largest studied. We then investigated a more comprehensive relocation strategy based on

call-graph information. Results demonstrated greater improvement in miss ratios for instruction TLBs, the average reduction being 9.6%.

Measuring and Improving TLB Performance for Linux GUI Applications

by
Sreekanth Konireddygari

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2008

APPROVED BY:

_____	_____
William Cohen	Matthias Stallmann

_____
Edward Gehringer
Chair of Advisory Committee

BIOGRAPHY

Sreekanth Konireddygari is currently a graduate student in the Department of Computer Science, North Carolina State University. He earned his Bachelor's degree in Electronics and Communication Engineering from Karnatak University, India. Sreekanth is currently employed with IBM Corporation, RTP, NC, where he develops cross-platform software for the Systems and Technology Group.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisors Dr. Edward Gehringer and Dr. William Cohen for giving me an opportunity to pursue this study, and for their guidance throughout the whole journey of this thesis work. I would like to thank Dr. Matthias Stallmann for agreeing to serve on the committee, and for his guidance on graph-partitioning techniques.

I would like to extend my sincere appreciation to my manager at work, Ed Rabenda, for his encouragement and support all along.

I am grateful to my wife and my family for their inspiration. Without their unflagging support, this study would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

A translation lookaside buffer (TLB) is present on all modern microprocessors to maintain a cache of recently used virtual-to-physical address translations. The *reach*, or *coverage*, of a TLB is the amount of physical memory that the processor can access without suffering an expensive TLB miss, and is computed as the product of number of entries contained in the TLB by the amount of memory mapped by each entry. Due to space and lookup time constraints, the coverage of the TLB has not scaled as rapidly as the virtual-memory address spaces of application programs. This has become a significant performance bottleneck affecting the overall performance of the system.

The run-time instruction and data reference patterns of an application can have a significant bearing on the TLB performance. Like traditional caches, TLBs can suffer from lack of temporal and spatial locality. For example, for a graphics application accessing pixels in a column, each pixel might be on a separate page of memory; thus, accessing the column would cause frequent TLB misses. Likewise, graphical user interfaces (GUIs) often use dozens of shared libraries. Each shared library is a separate unit; separate units are linked into separate pages. Thus, functions in separate libraries that call each other will always be on separate pages, increasing the number of pages in the working set and increasing the probability of TLB misses.

TLB performance can be an issue for GUI applications. We modeled typical TLB configurations while simulating representative GUI applications. Some sets in the set-associative TLB suffered *much* higher contention than other TLB sets. The high variation in

TLB misses per set suggested that performance could be improved by relocating (changing the virtual address of) the shared library with the page causing the high number of misses. Moving shared libraries with the highly contended pages yielded modest improvements in the TLB miss rates for the applications examined. Call-graph analysis of the applications used in the study also revealed that function calls among pages mapped into same TLB sets showed higher levels of evictions. The contention observed within these sets suggested that TLB performance can be improved by distributing frequently called pages among different sets. We studied two software-oriented shared-library relocation methodologies to improve TLB performance on existing CPU architectures.

1.1 Related Work

Most previous TLB studies used SPEC benchmarks with little consideration of how many shared libraries they used, or how the libraries were laid out in the process's virtual address space. These studies primarily used non-GUI applications with smaller application working sets. Chen et al. [1] studied the performance of various TLB configurations with applications such as *gcc*, *magic*, and *ld*, which had an average text segment size of about 200KB (50 pages, 4KB each). This is significantly smaller than the working-set size of most modern GUI applications. In this study, all applications have instruction regions of at least 81MB (20,380 pages, 4KB each). This is representative of GUI application programs in modern computing environments, and is at least an order of magnitude larger than the reach of the largest TLB configuration simulated. Nor did Chen et al [1] consider the

demands placed on the TLB by the instruction stream. While the instruction stream might have a negligible impact on the TLB in smaller applications, the instruction TLB could become a performance bottleneck for the execution of applications with large working-set size, such as *Abiword*, one of the benchmarks used in this study.

Talluri et al [11] propose increasing the size of the TLB entries to improve the TLB performance of superpages with less operating system support. However, the increased TLB area can lead to increased access times. Kandiraju et al [5] also propose significant changes to TLB hardware to implement prefetching techniques. As the CPUs are becoming increasingly complex, the space constraints may neither allow for such an increase in TLB sizes nor for a complex TLB circuitry.

On-the-fly superpage construction approach was proposed by Romer et al [9]. The methodology describes methods to detect when and where a superpage should be constructed based on TLB miss behavior gathered at runtime. However, the implementation of the page-promotion policies will require significant changes to the operating system.

2 Simulations and Benchmarks

To better understand the contention within TLB sets, a TLB simulator was developed. Since TLB performance improvements cannot be computed using formulas, simulations were conducted with benchmarks that represent commonly-performed tasks.

2.1 TLB Simulator

A TLB simulator was developed and integrated into the *Cachegrind* component of the *Valgrind* suite [7], a Linux binary instrumentation tool. This simulator is capable of running GUI programs on Linux without modifying the executable. Several simplifying assumptions were made in the simulator; they are discussed in the Section 2.2. Section 2.3 describes the simulator internals.

2.2 Simulator Design Assumptions

The simulator was designed to execute in a single-process environment; TLB misses that might occur due to process context switches are not considered. In most IA-32 CPU implementations, the TLB is flushed whenever there is a context switch (Moshe Bar [12]), which results in additional compulsory TLB misses. These compulsory misses will not increase the number of other (capacity and conflict) misses. Our hypothesis is that if TLB-set contention is detected and reduced when the process is being executed, overall TLB performance for all processes is also improved. Moreover, the rising number of processor cores in current processors tends to diminish the number of processes handled by a single

core, and hence the frequency of context switches. In addition, TLB flushes due to context switches can be avoided by extending the index of the TLB to include the process ID or the domain ID (Ulrich Drepper [12]).

TLBs often have multiple levels. A miss in one level is passed on to the next-level TLB. We did not simulate multi-level TLBs, since misses at one level that are satisfied in a lower-level TLB cost a lot less than a miss that goes all the way to the page table in main memory. Thus, the cost is dominated by the global TLB miss rate. We assume that the TLBs we simulated are last-level TLBs before the page table. These misses are relatively expensive due to a chain of read accesses needed to fill the new TLB entry.

2.3 Design of the Simulator

Valgrind uses dynamic binary instrumentation, which eliminates the need to modify or recompile the applications used in our simulation studies. The core of the *Valgrind* suite provides a synthetic CPU on which instrumented instructions are executed. The simulated TLB configuration is given as a command-line argument to the simulator. The TLB simulator is invoked by the *Valgrind* core whenever the simulated application references its instructions or data. The *Valgrind* core supplies the following information to the TLB simulator:

- Type of access (instruction or data)
- Virtual address of the referenced instruction or data

LRU replacement is simulated. During the simulation, the TLB simulator maintains various counters to keep track of the total number of TLB accesses, hits and misses. The simulator

dynamically tracks and records eviction patterns in each set, and dumps this data to a file for further analysis. These eviction-history tables allow us to determine which sets experience heavy contention. The TLB simulator translates the virtual address to a physical address, which is forwarded to the cache simulator of the *Valgrind* suite.

2.4 Simulation Environment

The simulations were done on a workstation with a 32-bit Intel Core Duo processor hosting a Linux Fedora core 6 operating system.

2.5 Workload

The following GUI-based Linux applications were used as benchmarks for this study: *Evince*, *Abiword*, *Gthumb* and *Gcalctool*. These applications were chosen because they possess the following characteristics:

- represent commonly-performed tasks
- depend on a number of dynamically-linked libraries
- have built-in accessibility features needed to automate the workload generation process

The GUI automation tool *dogtail* [3] was used to generate consistent workloads across multiple runs. Dogtail scripts were generated to automate the following user tasks for each of the applications:

- *Evince*: Open a 32KB PDF document and print it to a file in the current directory, and then close the application.

- *Abiword*: Type 15 sentences, format the paragraph to make it right aligned, save the file in the current folder, and then close the application.
- *Gthumb*: Open a folder that contains an image. The size of the image file is 8 KB. In browser mode, select the thumbnail image and view it in an image window, then close the application.
- *Gcalctool*: Enter the decimal number 45 and multiply this by the decimal number 3. Divide the result by the decimal number 7. Enter the decimal number 6, delete it using the backspace key, enter the decimal number 9, and subtract this from the result. Add .03 to the result. Compute the final result and quit the application.

2.5.1 Working Set

These applications mapped 82MB to 190MB of instruction pages and 31MB to 89MB of data pages (Table 2.1) into their address space, and touched at least 3179 instruction and data pages. This means that the address space was at least 57 times as large as the reach of the largest (512-entry) TLB, and the number of pages touched was at least six times its reach.

Table 2.1: Number of instruction and data (4KB) pages mapped and touched in the process address space of the benchmarks

	Pages Mapped		Pages Touched	
	Instruction	Data	Instruction	Data
Evince	20,380	9,507	1922	3933
Abiword	21,541	7,815	2110	3743
Gthumb	47,490	22,137	1804	3322
Gcalctool	19,630	20,659	1291	1888

2.6 Simulator TLB Configurations

The page size simulated for all TLB configurations is 4 KB, which is a standard page size supported by Linux on IA-32 systems. The following four TLB configurations, representing a broad spectrum of TLB configurations on CPUs from different CPU vendors, were simulated:

- Unified 512-entry, 4 KB page, 4-way set-associative TLB (2 MB reach),
- Unified 256-entry, 4 KB page, 4-way set-associative TLB (1 MB reach)
- Separate instruction and data TLBs, each 128-entry, 4 KB page, 4-way set associative (512 KB reach for instruction pages and 512 KB reach for data pages)
- Separate instruction and data TLBs, each 64-entry, 4 KB page, 4-way set associative (256 KB reach for instruction pages and 256 KB reach for data pages)

The TLB configurations simulated in this study closely relate to those on some of the existing CPU architectures as listed below:

AMD K8: L2, unified 512-entry, 4-way set associative

Intel P4: L1, I-TLB and D-TLB 64-entry

Pentium M: L1, I-TLB and D-TLB, 128-entry, 4-way

3 Methodolgy

We used two approaches to relocating shared libraries: (1) Relocation of a single library experiencing high contention and (2) relocation of many libraries, based on the application call graph. This chapter describes these two methodologies. The following section describes the technique used by Linux to layout the shared libraries in a process address space.

3.1 Layout of Shared Libraries in a Process's Virtual Address Space

When an application is being loaded for execution, the dynamic linker is responsible for resolving dependencies, and mapping and loading the shared libraries into the process address space. While doing so, it does not take into account the reference patterns of the application nor the performance implications for the TLB. Instead, the loader attempts to minimize the load time. This speedup is accomplished through prelinking, as discussed in the following section.

3.1.1 Prelink – A Method to Modify the Shared Library Layout

The *prelink* program on the Linux operating system is intended to save load time by performing relocations in advance. It resolves dependencies prior to loading the libraries. It scans all dependent libraries, assigns a unique slot in the virtual address space to each library, and relinks the shared library with that base address. The dynamic linker, during the load process, maps the prelinked library into the allocated virtual memory address range, unless the memory address range is already occupied. In addition, the prelink utility, with

the help of the dynamic linker, resolves all relocations in the object files against dependent libraries, and stores the relocations in shared library object files. In our study, we prelinked all applications prior to the simulations to keep the shared library layout consistent across different data-collection runs. This allowed us to modify the virtual-address slot of a single library whose pages were contending heavily with other pages in a TLB set in a particular TLB configuration. The prelink utility provides a "relink-only" (`-r`) option, which allowed the location of a shared library to be changed in the object files without affecting virtual-address slots of other shared libraries. Based on the process address map, an empty slot was chosen so that the page descriptor would fall in the desired TLB set. The shared library was moved there using the relink option. The simulations were rerun after relinking libraries for each of the TLB configurations.

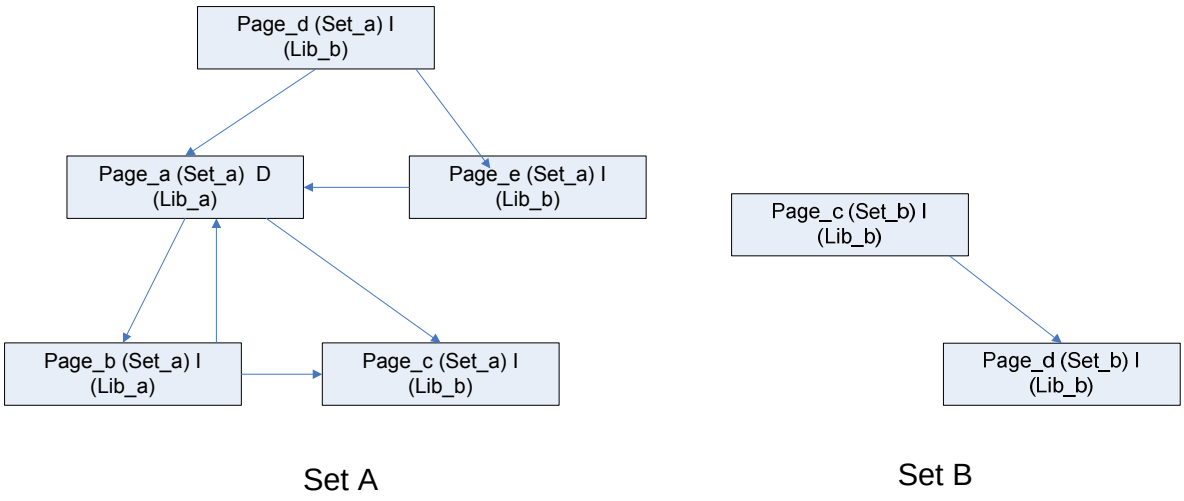
3.2 Single-library Relocation Based on TLB-set Eviction Patterns

The objective of this methodology is to move the most heavily-contending page from a set with most evictions to a set with fewer evictions in order to reduce TLB misses. Figure 3.1 illustrates a sample relocation of a page from Set A to Set B. The pages are denoted by circles, and the transitions in the figure denote the contention between the pages. Section 3.2.2 provides a more in-depth description of the graphs that were used in this study to analyze the contention between the pages in a set. The dotted lines in Figure 3.1 after relocation indicate the reduction in the number of evictions in Set A.

The single-library relocation study involves the following steps (see Figure 3.2): collection of simulation data from benchmarks with the default layout of the shared libraries, analysis

of the simulation data to detect the TLB sets with the highest number of conflict misses,

Before relocation:



After relocation:

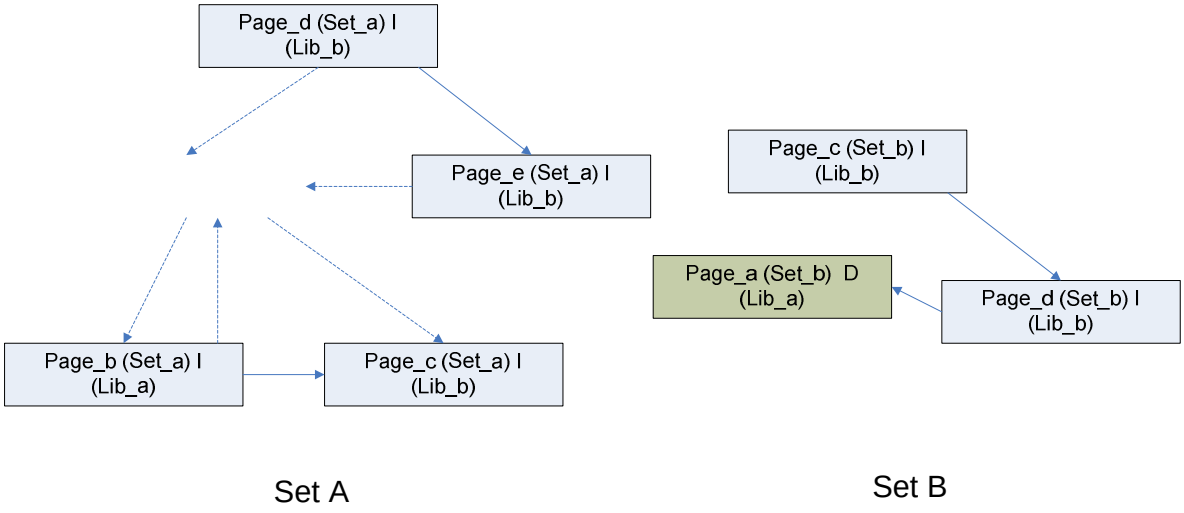


Figure 3.1: An example illustrating the single-library relocation methodology

modification of the layout of the shared library by relocating a single library, and manually, collection of the simulation data after the modification. The method described in Section 3.1.1 was used to modify the layout by changing the base virtual address of the shared library in the Linux operating system.

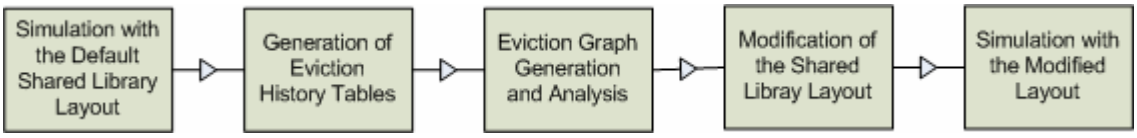


Figure 3.2: Steps involved in the single-library relocation strategy

3.2.1 TLB-Set Contention Analysis with Page Eviction History

To analyze the eviction patterns and study TLB-set contention among libraries, the TLB simulator generates an eviction-history table that contains the following information in tabular format:

- referenced page (victimizer) that is causing a TLB access, and is being loaded into one of the TLB sets,
- the page that is being evicted (victim) from the TLB,
- the number of times the victmizer page evicted the victim page,
- the types of the victimizer and victim page (instruction or data),
- the TLB set number,
- the names of the files to which each victimizer and victim page belongs,
- the name of the function or the data address that was being referenced in the victimizer page that causes the eviction

- number of times data or instructions referenced in the victimizer page

3.2.2 Eviction Graph Generation and Analysis

To study the contention for TLB sets by pages from shared libraries, eviction graphs were generated from the eviction history tables. Each set is represented by an eviction graph, and each node within the graph represents a page. A sample eviction graph used for contention analysis is illustrated in Figure 3.3. The edges represent transitions showing which page is replaced in the TLB. The transition from page *A* to page *B* indicates that (the victim) page *A* is being replaced by (the victimizer) page *B*. A label on the transition denotes the number of times the victim page has been evicted by the victimizer page. Nodes with less than 500 transitions (incoming edges) are not generated in the eviction graph for the 512-entry and 256-entry TLB configurations; for 128-entry and 64-entry TLB simulations, the corresponding thresholds are 1000 and 3000 evictions page, respectively. The sample eviction graph in Figure 3.3 illustrates the contention between three pages: page *A*, page *B* and page *C*. *A* is replaced by *B* 100 times because of the reference to the routine *func*() on *B*. Similarly, *B* and *C* replace each other 2500 times. After doing the analysis, we relocate the base virtual address of the library, so that that the most highly-contending page from this library gets mapped into a different set.

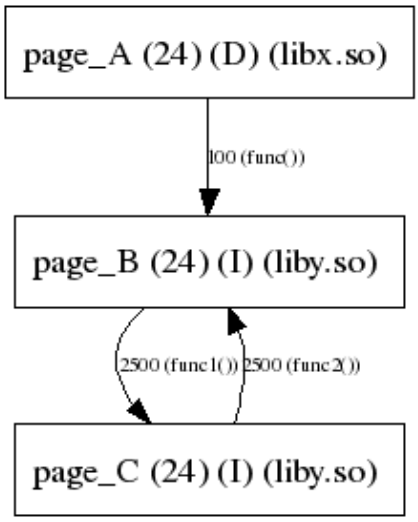


Figure 3.3: A sample eviction graph illustrating contention in set 24

3.3 Relocation Based on Application Call Graph

Another approach to reducing TLB contention involves relocation of shared libraries based on the call-graph information. Call-graph information was collected by the instrumentation tool *Callgrind,* a sub-component of the tool *Valgrind*. The objective of this strategy is to distribute the frequently called (referenced) pages among different sets. This tends to improve TLB performance by minimizing conflicts within TLB sets. The algorithm keeps track of the (dynamic) number of calls to pages in each set of the TLB. The most frequently called page of a library is assigned to the set having the least total calls thus far. This results in the library being assigned to a specific location in the virtual address space. Linked lists representing callee-caller relationships are constructed to lay out the libraries in the process's address space as explained in the algorithm below.

3.3.1 TLB-Set Allocation and Library Relocation Algorithm

The *TLB-set allocation and relocation algorithm* uses the callgraph data captured by running the application with the default layout of the shared libraries. The call-graph output generated by *Callgrind* lists the caller page for each of the callee (virtual) pages. In the sample call-graph output shown in Figure 3.4, the last entry denoted with a "*" is the callee page (28C6) and the entries above it (26AB, 2EE7, C9F etc.) denoted by "<" are its caller pages. The number to the left of "<" is the number of instruction references, and to its immediate right is the caller page number. The number in the parentheses is the number of times the function in the caller page called the function in the callee page.

```
86, 141 < 2EEA (2101x) [/usr/lib/libORBIT-2.so.0.1.0]
105,493 < 2EF0 (2573x) [/usr/lib/libORBIT-2.so.0.1.0]
340,628 < 2E28 (8308x) [/usr/lib/libbonobo-2.so.0.0.0]
784 < E (16x) [/usr/lib/gtk-2.0/2.10.0/engines/libclearlooks.so]
111,233 < AF4 (4734x) [/lib/libc-2.5.so]
11,016 < C9F (347x) [/usr/lib/libxml2.so.2.6.27]
86,141 < 2EE7 (2101x) [/usr/lib/libORBIT-2.so.0.1.0]
720,903 < 26AB (17583x) [/lib/libgobject-2.0.so.0.1200.9]
114,845,705 * 28C6 [/lib/libpthread-2.5.so]
```

Figure 3.4: Sample output of the *Valgrind* simulator listing TLB page interactions

The algorithm consists of two phases: initialization, and distribution of callee pages among sets. The initialization phase does the following:

- The call-graph data file generated by *Callgrind* is processed, and a linked list of all callee pages is constructed.
- For each callee page, a linked list of caller pages is constructed. The caller and the callee pages are represented by data structures with the following: page address, name of the library that owns the page, and reference count (number of times the page has been referenced).

- The callee page list is sorted in decreasing order of reference counts; this places the callee page with most references at the top of the list.
- An *n*-entry pseudo-TLB, represented by a linked list, is constructed, where *n* is the number of sets. Each set is represented by a data structure which contains the following: set number, list of callee pages mapped into this set, and reference count (the sum of reference counts of all callee pages mapped into the set).
- Each set in the pseudo-TLB comprises a list of library pages mapped into that set. Each library page is represented by a data structure which consists of the following: name of the shared library mapped into that set, a list of callee pages of the shared library mapped into that set.

The callee-page distribution phase of the algorithm consists of the following steps:

- The page at the front of the callee page list (the page with highest reference count) is picked for relocation.
- If the library that owns the callee page is already relocated, the next callee page from the list is picked.
- The TLB set with the lowest reference count is picked from the list that represents the pseudo-TLB.
- To minimize TLB replacements, the callee page is placed in a set other than the set that its callers lie in. So for each candidate set, the callee-page's caller list is traversed to check whether any of the calling pages lie in the candidate set. If it is found that the sum total of calls from all pages in the candidate set exceeds a certain threshold, then the algorithm proceeds to pick the TLB set with the next

lowest reference count. This check is performed iteratively until a set is picked from the pseudo-TLB, and the callee page is mapped into that set. The callee page is added to the library page list of the pseudo-TLB set to indicate that the page has been mapped into that set.

- Based on the selected TLB set, a new starting address for the callee page's library is calculated.
- If the library whose page has just been relocated has any other pages, they are mapped to virtual addresses contiguous with the just-relocated page. The reference counts of the sets into which the callee pages are mapped are updated.
- These steps are repeated starting with the next callee page in the sorted callee-page list. The process continues until all callee pages have been relocated.

4 Results

This chapter presents results from simulations before and after shared-library relocation, using the two methodologies explained in chapter 3.

4.1 Results from Single-library Relocation with Eviction Pattern Methodology

The graphs in Figure 4.1, Figure 4.2, and Figure 4.3 illustrate the distribution of misses across sets for *Evince* with unified 512-entry TLB, *Abiword* with unified 256-entry TLB and *Gthumb* with 64-entry split instruction TLB simulation runs, respectively, using the default layout of the shared libraries in their address spaces.

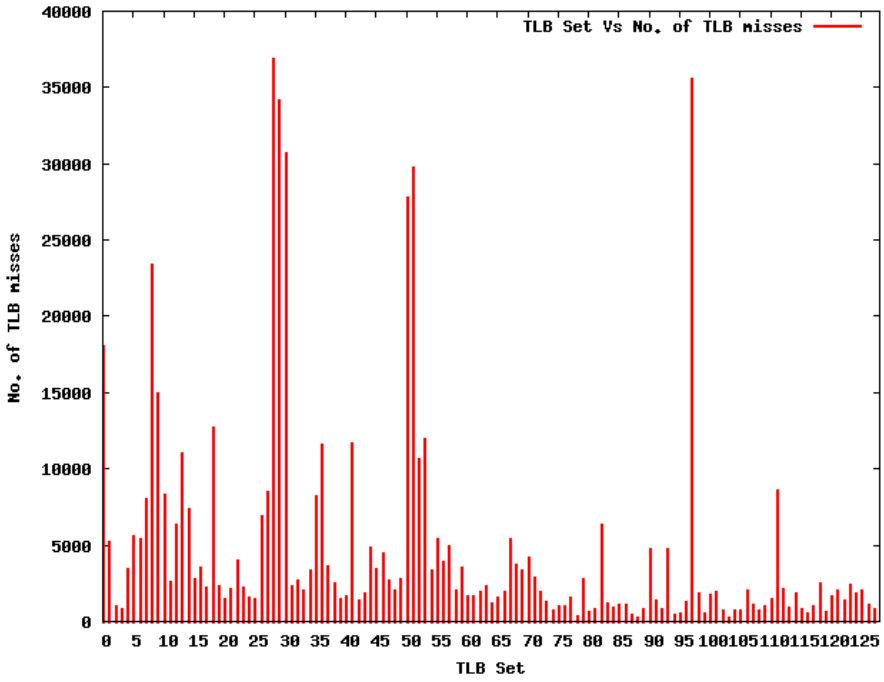


Figure 4.1: Distribution of misses among TLB sets from *Evince* with unified 512-entry, 4-way TLB simulation before the shared library layout modification

The peaks in the graph show that the contention among a few sets is significantly higher than for other sets. This suggests that if the pages with repeated conflicts in high-contention sets are moved to other sets with relatively fewer conflicts, a reduction in the total number of conflict misses might be achieved.

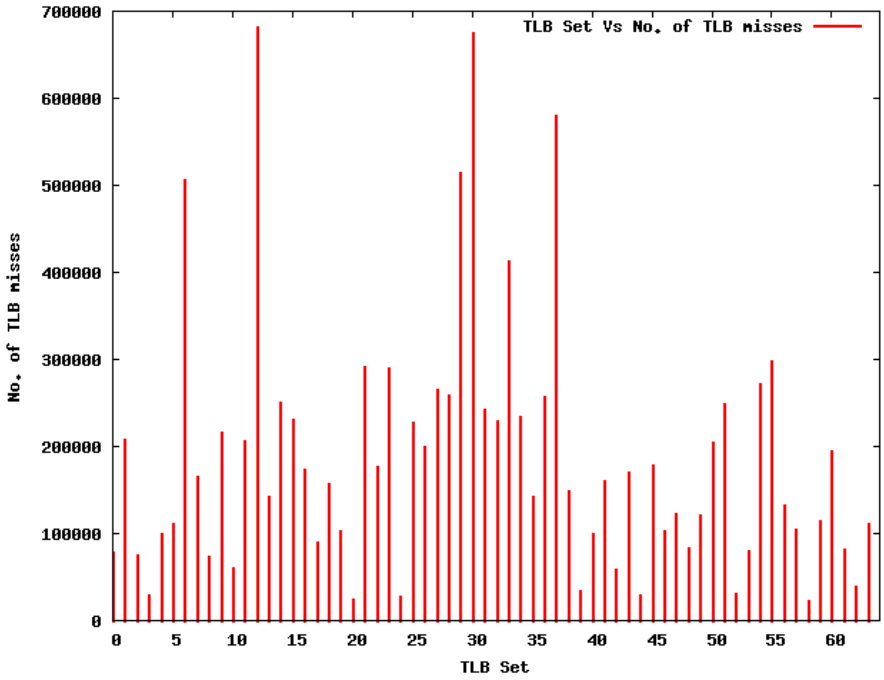


Figure 4.2: Distribution of misses among TLB sets from *Abiword* with unified 256-entry, 4-way TLB simulation before the shared library layout modification

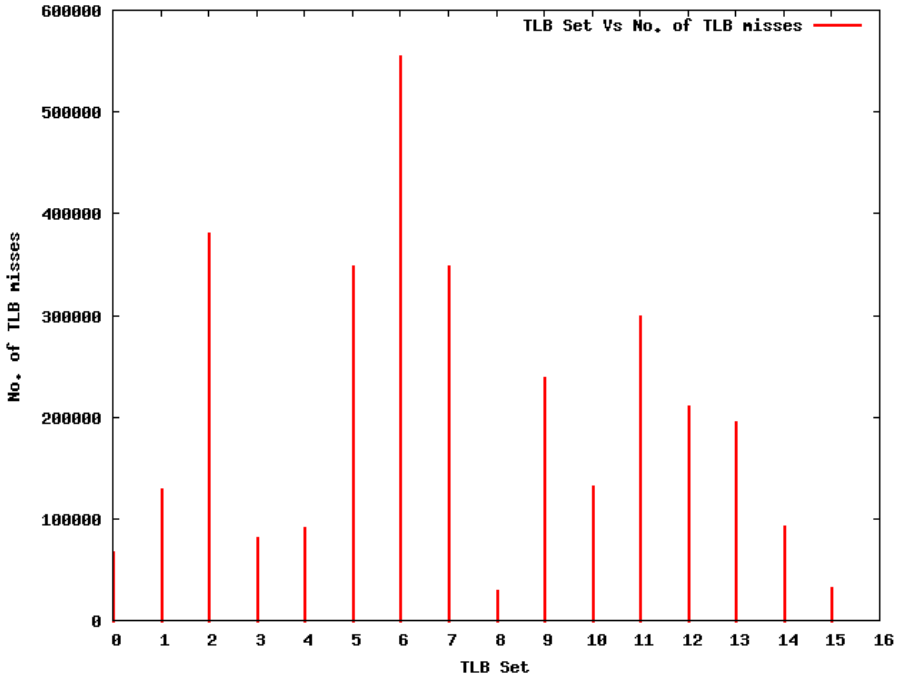


Figure 4.3: Distribution of misses among TLB sets from *Gthumb* with 64-entry, 4-way split data TLB simulation before the shared library layout modification

The eviction graph shown in Figure 4.4 illustrates the contention in set 97 (the set with the maximum number of misses) among pages from different libraries for *Evince* in the 512-entry TLB simulation run. It is clear from the graph that page 0x2E61 from the shared library *libbonobo-2.so.0.0.0* is involved with many TLB evictions in the high-contention set 97; the page is either being brought into the set, frequently replacing other pages; or is being evicted frequently by other pages, as indicated by the edges. The virtual base address of this shared library was modified so that the page is mapped into a different TLB set. The library was moved to virtual address 0x12C000, which was previously unoccupied in the process address space. This change in virtual address ensures that the page with maximum transitions is not mapped into the high-contention TLB set.

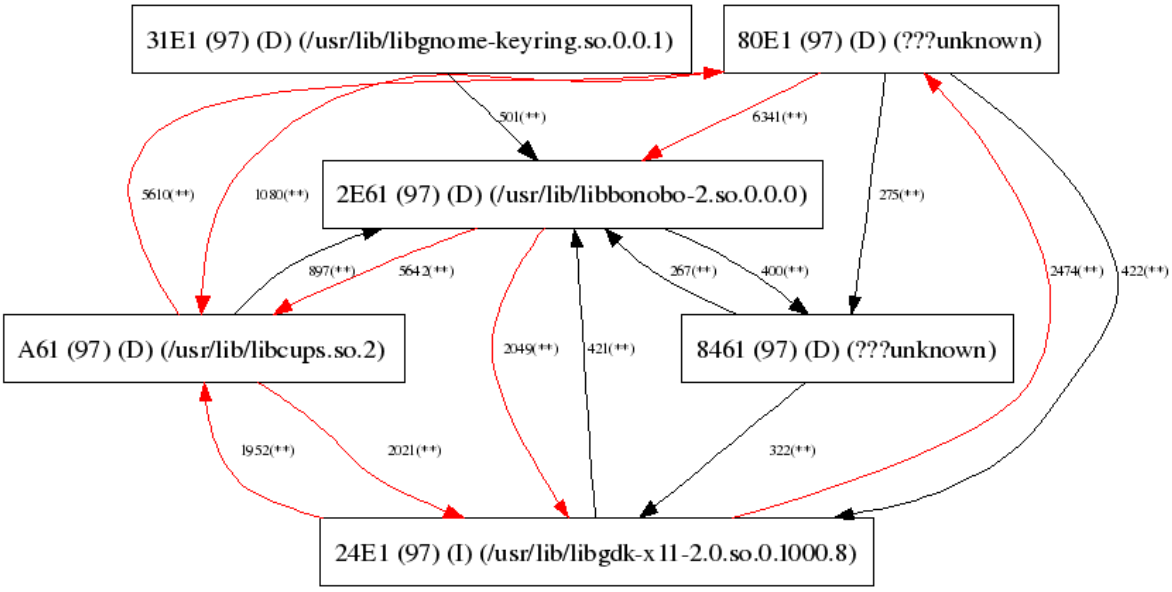


Figure 4.4: Eviction graph for Evince with unified 512-entry, 4-way TLB simulation

Table 4.1 lists the libraries that have been modified for the 512-entry TLB simulations. The column “Page in high-contention set” lists the single page in each library causing the maximum number of evictions in the highly-contended set before the layout modification. The column “Page involvement in total misses” lists the percentage of the total number of misses for which the page (listed in the column “Page in high-contention set”) from the library is either fetched into the TLB or evicted from the TLB. The libraries listed in column “Library” of the table were picked for layout modification because their most highly-contending page is involved in about 2.6% (on average) of the total number of misses in the respective benchmark. The libraries that have been modified for the 128-entry instruction TLB simulation are listed in Table 4.2. Figure 4.5 displays the difference in the misses in each of the sets before and after the modification of the base virtual address of the

shared library *libbonobo-2.so.0.0.0* for *Evince* with 512-entry TLB simulation. The negative peaks show the reduction in the number of misses after the

Table 4.1: The default and the new base virtual addresses of the libraries for the 512-entry TLB simulation benchmarks

	Library	Page in high-contention set	Page involvement in total misses	Default Address	New Address
Evince	libbonobo	0x2E61	2.87%	0x2E05000	0x12C000
Abiword	libgdk	0x24B7	2.13%	0x24A3000	0x11C000
Gthumb	libORBit	0x2F1D	2.78%	0x2ECF000	0x2ED000

Table 4.2: The default and new base virtual addresses of the libraries for the split 128-entry instruction TLB simulation

	Library	Page in high-contention set	Page involvement in total misses	Default Address	New Address
Evince	libdbus	0x3032	3.83%	0x3012000	0x497000
Abiword	libgobject	0x2697	1.86%	0x2691000	0xF35000
Gthumb	libORBit	0x2EF6	3.81%	0x2ECF000	0x2ED000

modification of the layout. Set 97, which was one of the heavily contended sets in Figure 4.5, shows a negative peak indicating less contention after the modification. Figure 4.6 illustrates the differences in TLB misses across sets sorted in descending order. It can be observed that the shaded area below the axis is larger than the shaded area above which indicates that there is a net decrease in the number of TLB misses after relocation.

Figure 4.7 presents the coefficient of variance values for the misses across all sets for *Evince* with 512-entry TLB simulations before and the after the modification of the virtual base address of the shared library *libbonobo-2.so.0.0.0*. A lower coefficient of

variance indicates that replacements are more evenly spread among the sets, which means

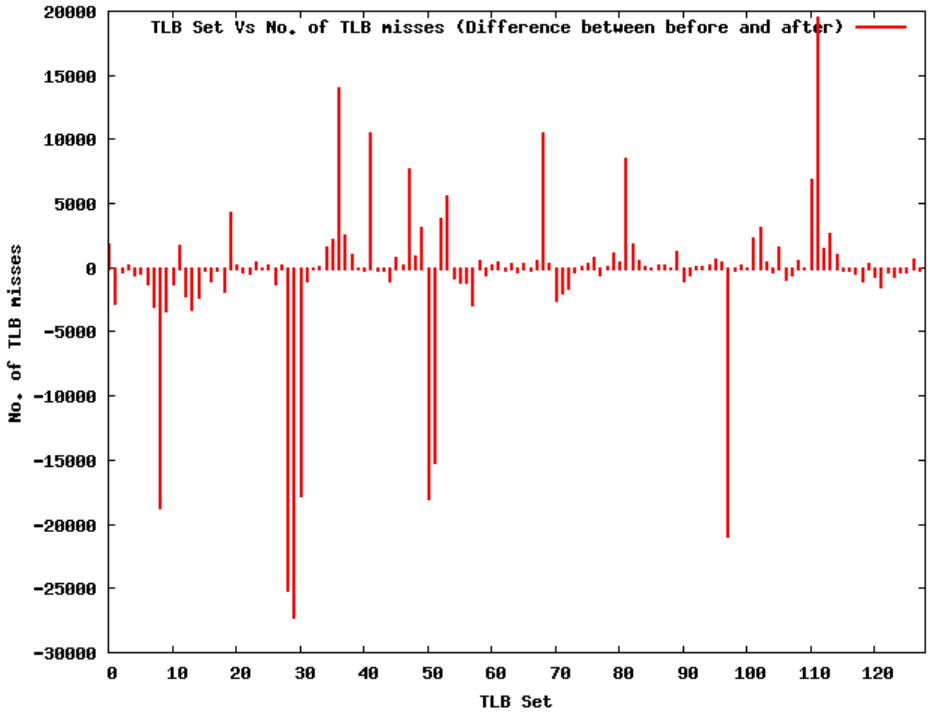


Figure 4.5: Difference in TLB misses across sets before and after the modification of the layout for Evince with unified 512-entry, 4-way TLB

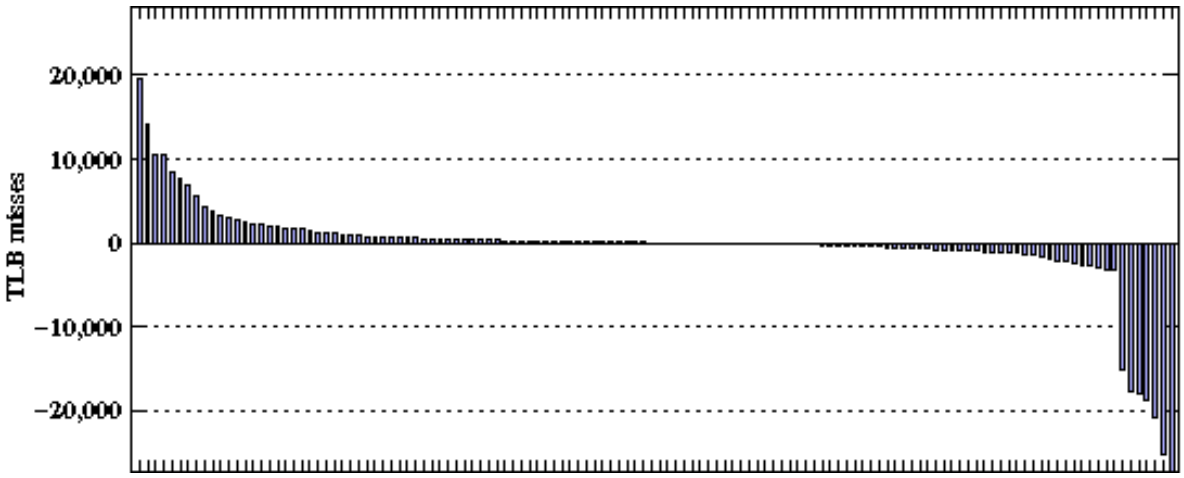


Figure 4.6: Differences (after and before relocation) in TLB misses across sets sorted in descending order for Evince 512-entry unified TLB. Each vertical bar represents a set.

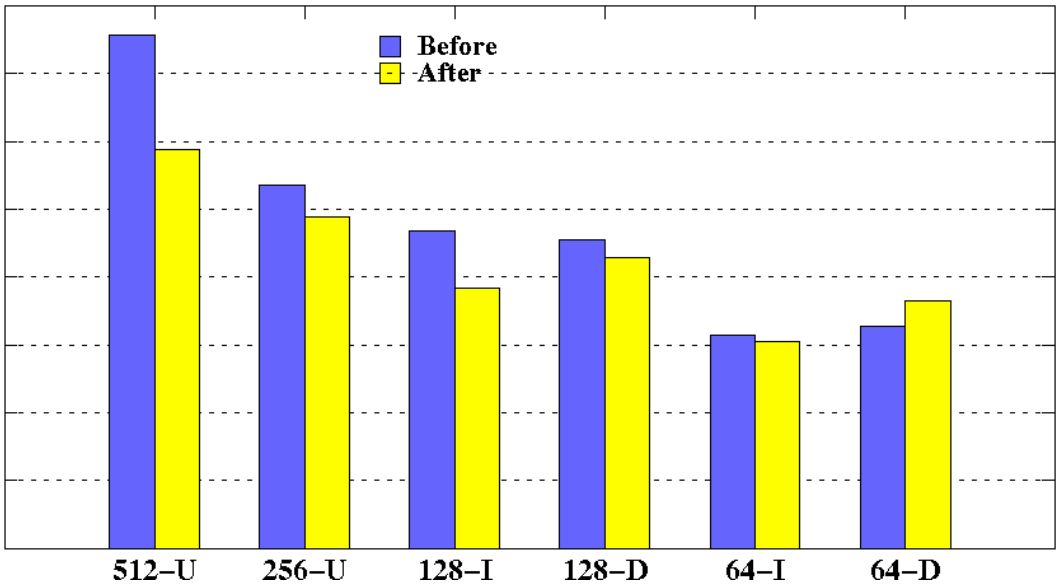


Figure 4.7: Coefficient of variance of misses across TLB sets for Evince before and after the base address modification

less thrashing in heavily contended TLB sets. This also suggests that the peaks (of misses) seen in some of the sets before the layout modification in Figure 4.1 have been reduced. Table 4.3, Table 4.4, Table 4.5, Table 4.6, Table 4.7 and Table 4.8 present the total number of TLB accesses and misses for the benchmarks before and after the shared library layout modification. It can be observed that misses are reduced after layout modification, except for a slight increase in the number of misses for *Abiword* with split 128-entry and 64-entry data TLB simulations. However, instruction-TLB misses for these two simulations have decreased. The corresponding miss rates for all simulated TLB configurations before and after the layout modification are presented in Figure 4.8, Figure 4.9, Figure 4.10, Figure 4.11, Figure 4.12, and Figure 4.13. These figures illustrate that the miss rates have decreased for all TLB configurations after the relocation except for Abiword with the 128-

entry and 64-entry D-TLB simulations as shown in Figure 4.11 and Figure 4.13.

Table 4.3: Performance figures for the unfied 512-entry, 4-way TLB

	TLB Accesses		TLB Misses	
	Before	After	Before	After
Evince	2,440,825,239	2,453,775,228	609,102	552,421
Abiword	6,998,081,655	6,990,273,929	3,109,707	2,852,774
GThumb	1,957,182,463	1,958,059	493,741	468,913

Table 4.4: Performance figures for the unified 256-entry, 4-way TLB

	TLB Accesses		TLB Misses	
	Before	After	Before	After
Evince	2,432,596,581	2,457,165,965	3,366,102	2,872,029
Abiword	6,977,572,797	6,957,123,472	11,966,085	11,337,441
GThumb	1,958,149,042	1,957,018,016	2,026,559	1,984,695

Table 4.5: Performance figures for 128-entry, 4-way I-TLB

	TLB Accesses		TLB Misses	
	Before	After	Before	After
Evince	1,566,255,316	1,575,622,195	1,167,461	1,088,987
Abiword	4,330,240,432	4,305,221,408	6,895,394	6,550,586
GThumb	1,253,874,020	1,255,493,913	791,188	787,931

Table 4.6: Performance figures for 128-entry, 4-way D-TLB

	TLB Accesses		TLB Misses	
	Before	After	Before	After
Evince	890,978,634	895,263,366	2,251,358	2,009,389
Abiword	2,646,306,273	2,629,314,357	4,907,404	4,964,352
GThumb	702,923,236	703,660,666	1,517,735	1,498,081

Table 4.7: Performance figures for 64-entry, 4-way I-TLB

	TLB Accesses		TLB Misses	
	Before	After	Before	After
Evince	1,562,216,950	1,577,635,742	4,336,542	4,328,009
Abiword	4,335,826,825	4,340,823,503	16,810,390	16,384,778
GThumb	1,253,613,714	1,255,616,965	3,343,632	2,884,001

Table 4.8: Performance figures for 64-entry, 4-way D-TLB

	TLB Accesses		TLB Misses	
	Before	After	Before	After
Evince	888,170,929	896,415,162	5,277,293	5,030,958
Abiword	2,649,817,917	2,652,221,532	11,496,889	12,364,948
GThumb	702,651,903	703,799,087	3,643,586	3,619,820

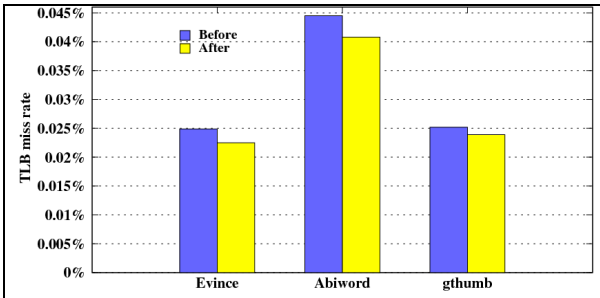


Figure 4.8: Miss rates for the unfied 512-entry TLB

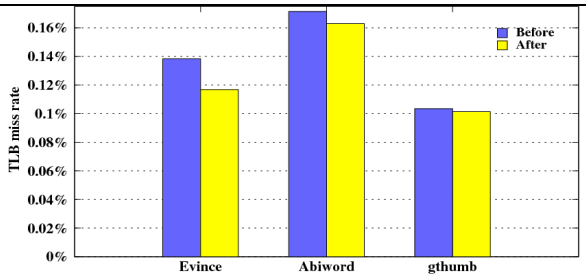


Figure 4.9: Miss rated for the unified 256-entry TLB

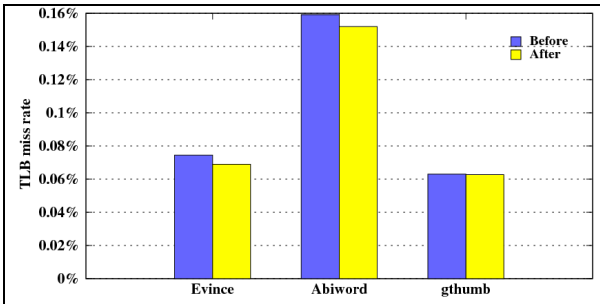


Figure 4.10: Miss rates for 128-entry I-TLB

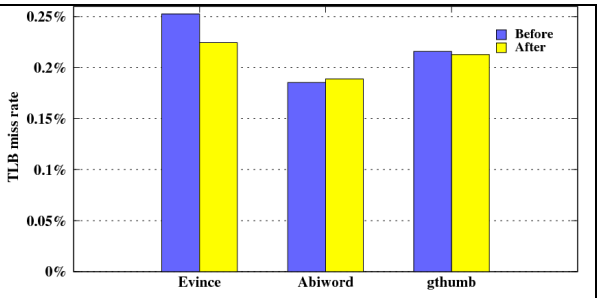


Figure 4.11: Miss rates for 128-entry D-TLB

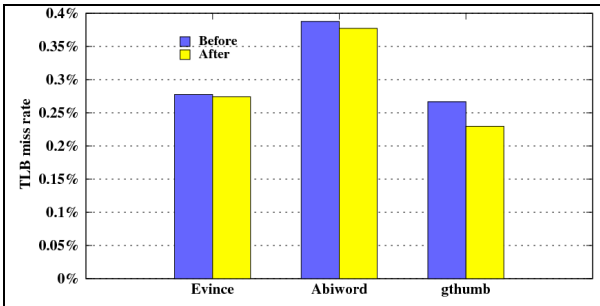


Figure 4.12: Miss rates for 64-entry I-TLB

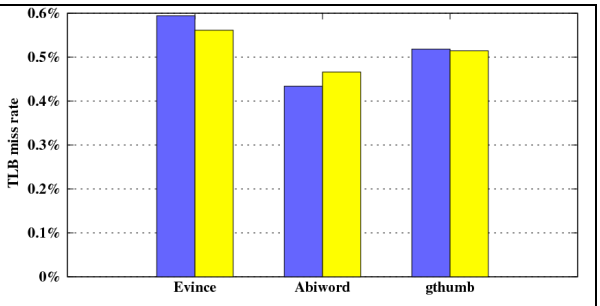


Figure 4.13: Miss rates for 64-entry D-TLB

4.2 Results from Call-Graph Methodology

Table 4.9, Table 4.10, Table 4.11, Table 4.12, Table 4.13 and Table 4.14 present TLB performance figures before and after shared library relocation using the call-graph technique of Section 3.3. Figure 4.14 illustrates the differences in TLB misses after and before the relocation sorted in descending order for Evince with 128-entry I-TLB. It can be observed that there are more number of sets with greater magnitude below the axis (negative) indicating a net decrease in the number of TLB misses. The miss ratios for the 128-entry and 64-entry split TLB simulations (in Figure 4.15 and Figure 4.17) indicate that a significant reduction in miss ratios was achieved for the instruction TLBs after the relocation. However, the data-TLB results for the split 128-entry and 64-entry TLBs are mixed, as shown in Figure 4.16 and Figure 4.18. Let's consider why this might be. Each library contains both instruction and data pages, which are moved as a single unit. Using the call graph allowed us to minimize conflicts between instruction pages, resulting in lower miss ratios for all the I-TLBs. The greatest reduction in average miss-ratio, 10.5%, was seen for the 64-entry I-TLB. The relocation of data pages may cause interference with

other page descriptors in the data TLB, or with instruction pages in a unified TLB. Thus, the unified TLBs also demonstrate mixed results, as shown in Figure 4.19 and Figure 4.20.

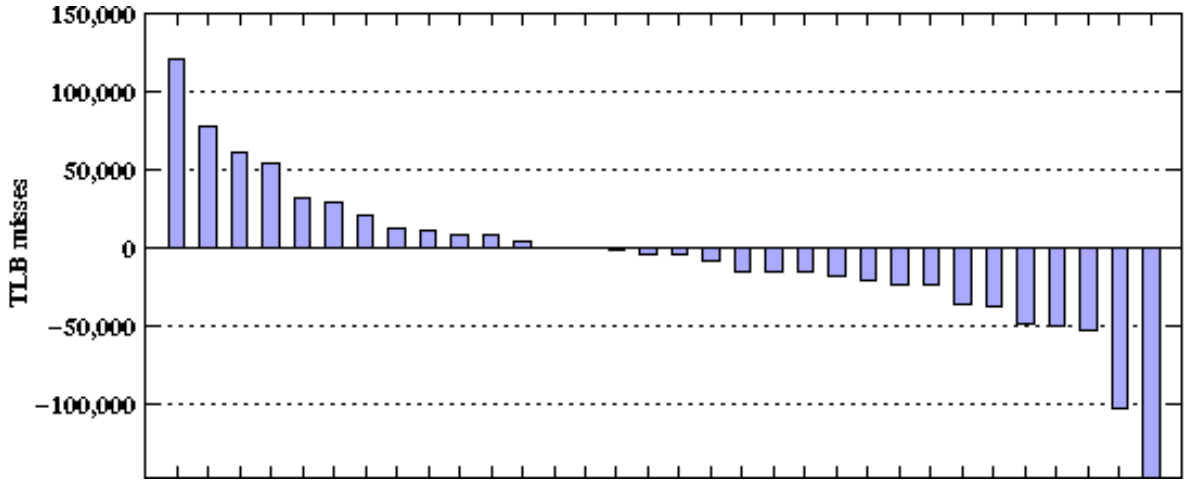


Figure 4.14: Differences (after and before relocation) in TLB misses across sets sorted in descending order for Evince with 128-entry I-TLB. Each vertical bar represents a set.

Analysis of eviction graphs from the unified TLBs indicated less contention between the instruction pages within sets. However, this was offset by increased contention between data and instruction pages, resulting in higher miss ratios for some sets. Some data pages are called *a*nonymous pages because they do not map back to files. The data pages that conflicted with the instruction pages were usually from anonymous regions of the shared library. The use of superpages (very large pages) for the anonymous data regions might be able to reduce conflicts.

Table 4.9: Performance figures for the unified 512-entry, 4-way TLB

	TLB Accesses		TLB Misses	
	Before	After	Before	After
Evince	2,440,825,239	2,493,092,468	468 609,102	675,382
Abiword	6,998,081,655	6,943,165,650	3,109,707	3,306,917
GThumb	1,957,182,463	1,957,757,267	493,741	488,084
Gcalctool	10,302,050,488	10,317,161,194	892,587	1,085,857

Table 4.10: Performance figures for the unified 256-entry, 4-way TLB

	TLB Accesses		TLB Misses	
	Before	After	Before	After
Evince	2,432,596,581	2,466,145,691	3,366,102	2,462,219
Abiword	6,977,572,797	6,927,738,289	11,966,085	12,440,600
GThumb	1,958,149,042	1,957,960,886	2,026,559	1,885,401
Gcalctool	10,291,653,021	10,310,440,397	8,183,581	10,821,483

Table 4.11: Performance figures for 128-entry, 4-way I-TLB

	TLB Accesses		TLB Misses	
	Before	After	Before	After
Evince	1,566,255,316	1,568,440,670	1,167,461	1,028,631
Abiword	4,330,240,432	4,265,969,510	6,895,394	6,440,247
GThumb	1,253,874,020	1,256,111,153	791,188	718,884
Gcalctool	6,434,091,484	6,477,231,581	5,081,368	4,665,743

Table 4.12: Performance figures for 128-entry, 4-way D-TLB

	TLB Accesses		TLB Misses	
	Before	After	Before	After
Evince	890,978,634	907,951,938	2,251,358	1,995,101
Abiword	2,646,306,273	2,607,129,388	4,907,404	5,180,330
GThumb	702,923,236	704,228,286	1,517,735	1,458,420
Gcalctool	3,849,406,916	3,826,300,213	5,018,408	4,980,255

Table 4.13: Performance figures for 64-entry, 4-way I-TLB

	TLB Accesses		TLB Misses	
	Before	After	Before	After
Evince	1,562,216,950	1,574,408,827	4,336,542	4,086,406
Abiword	4,335,826,825	4,306,619,272	16,810,390	15,845,033
GThumb	1,253,613,714	1,255,383,279	3,343,632	2,959,370
Gcalctool	6,448,666,909	6,456,974,567	23,963,801	19,513,587

Table 4.14: Performance figures for 64-entry, 4-way D-TLB

	TLB Accesses		TLB Misses	
	Before	After	Before	After
Evince	888,170,929	902,118,785	5,277,293	5,222,082
Abiword	2,649,817,917	2,631,807,307	11,496,889	12,566,841
GThumb	702,651,903	703,719,931	3,643,586	3,866,747
Gcalctool	3,856,304,970	3,860,823,018	18,592,604	16,367,273

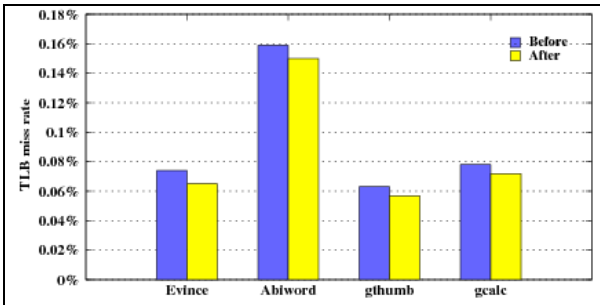


Figure 4.15: Miss rates for 128-entry I-TLB

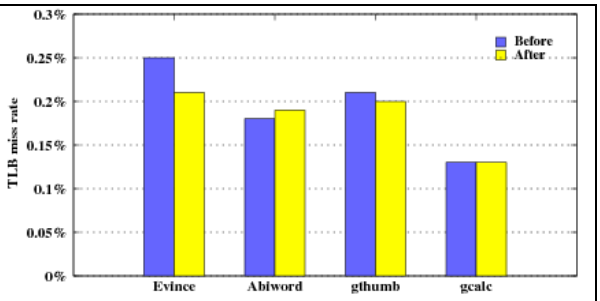


Figure 4.16: Miss rates for 128-entry D-TLB

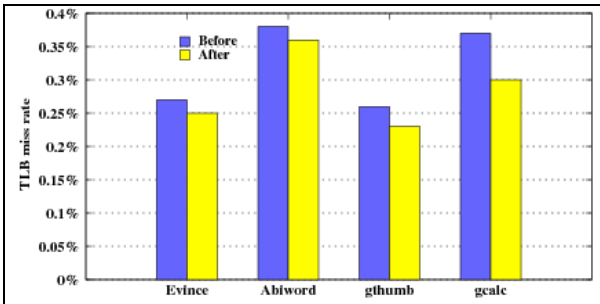


Figure 4.17: Miss rates for 64-entry I-TLB

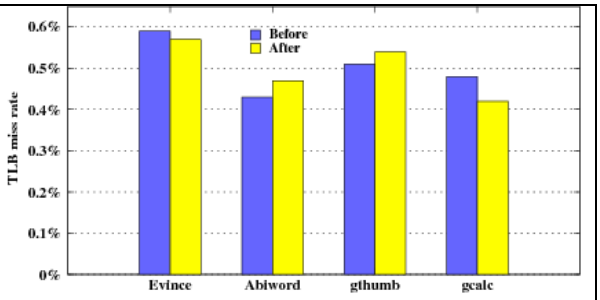


Figure 4.18: Miss rates for 64-entry D-TLB

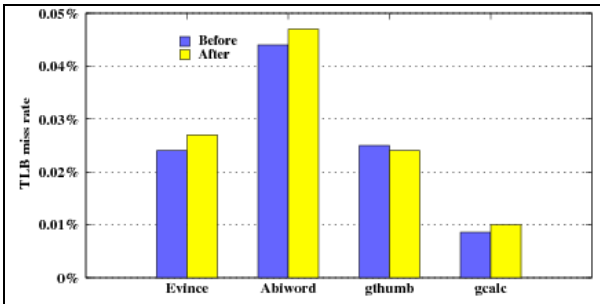


Figure 4.19: Miss rates for the unified 512-entry TLB

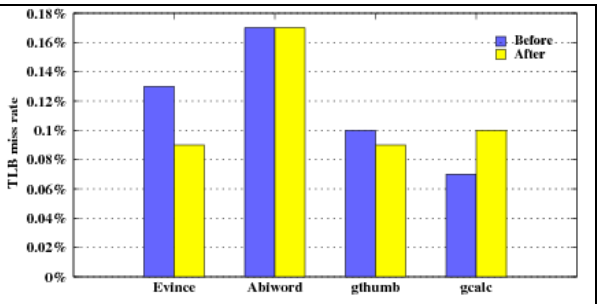


Figure 4.20: Miss rates for the unified 256-entry TLB

4.3 Different workloads

Analysis of eviction graphs from the unified TLBs indicated less contention between the instruction pages within sets. However, this was offset by increased contention between data and instruction pages, resulting in higher miss ratios for some sets. Some data pages are called *a*nonymous pages because they do not map back to files. The data pages that conflicted with the instruction pages were usually from anonymous regions of the shared library. The use of super pages (very large pages) for the anonymous data regions might be able to reduce conflicts.

As we have seen, the call-graph methodology is based on changing the layout of instruction pages only. By contrast, the eviction pattern methodology takes into account conflicts between data and instruction pages into account. However, it does this by tracking the TLB evictions during a single run of the program. Thus the question arises, Will a relocation based on the performance of a single *run* of the system produce an improvement for a different run, e.g., with a different set of inputs? Fortunately, improvements made to benefit one run of an application also benefit other runs.

To evaluate the impact of the modified shared library across different workloads, we studied TLB behavior with a workload that was different from the one used in initial data-collection runs. The benchmark *Evince* with unified 512-entry TLB was re-simulated with the following workload:

- Open a PDF document
- Zoom in three times
- Zoom out one time
- Perform best-fit operation
- Save the file in the current folder
- Quit the application

Data was collected before and after layout modification. The library and the new load address of the shared library *libbonobo-2.so.0.0.0* was kept consistent with the initial workload. It can be observed from the results presented in Table 4.15 that there is a decline in the TLB miss ratio by 4.9%.

Table 4.15: Performance figures with a different workload for Evince with unified 512-entry, 4-way TLB

	TLB Accesses		TLB Misses	
	Before	After	Before	After
Evince	6,208,255,627	5,998,949,296	828,097	761,396

4.4 Performance

TLB misses have been shown to be relatively expensive. Values between 350 [4] and 2000 [2] cycles have been measured. For the estimates of the effect of TLB misses it will be assumed the cost of a TLB miss is 375 cycles (Ulrich Drepper [4]), the cost of two or three

accesses to main memory that cannot be performed in parallel. From Table 4.10, the number of TLB accesses before relocation for Evince with the unified 256-entry TLB are 2,432,596,581 and number of misses before relocation are 3,366,102. For a miss ratio of 0.00138 and 375 cycle TLB miss cost, the contribution to CPI for TLB misses before relocation is 0.51. For a miss ratio of 0.00099 (2,466,145,691 accesses and 2,462,219 misses) after relocation using the call-graph method, the contribution to CPI for TLB misses is 0.37. For a high-performance processor which has a CPI around 1 this reduction would be a significant performance improvement.

Table 4.16 presents TLB miss contributions to CPI before and after relocation using the call-graph method. Unsurprisingly, the largest TLB, 512 entries unified has the lowest CPI cost for TLB misses, a contribution of 0.17 or less to the CPI. For the smallest TLB split 64 entries for instructions and 64 entries for data the CPI contribution was 0.96 CPI or more. These experiments indicate there is a significant performance improvement in the TLB for the GUI applications. The area required for increasing the TLB size may be a better investment than using it for cache memory. Table 4.16 shows improvements for the instruction TLBs using the call-graph method, but increased data TLB misses for *Abiword* and *GThumb*. It also shows mixed results for the larger unified TLBs.

Table 4.16: Miss cost in terms of CPI before and after relocation of shared libraries with the call-graph method.

Note: The cost for each TLB miss is assumed to be 375 cycles. ‘B’ stands for miss cost before relocation and ‘A’ stands for miss cost after relocation.

	64i		64d		128i		128d		256u		512u	
	B	A	B	A	B	A	B	A	B	A	B	A
Evince	1.03	0.96	1.26	1.24	0.27	0.24	0.53	0.47	0.51	0.36	0.09	0.09
Abiword	1.44	1.37	0.99	1.09	0.59	0.56	0.42	0.45	0.63	0.63	0.16	0.17
GThumb	0.99	0.87	1.08	1.15	0.23	0.21	0.45	0.43	0.38	0.36	0.09	0.09
Gcalctool	1.38	1.13	1.08	0.95	0.29	0.27	0.29	0.28	0.29	0.39	0.03	0.03

5 Conclusions

This study investigated the impact of shared libraries on the TLB. Our benchmark results indicate that it is possible to improve on the random layout of shared libraries to produce a lower TLB miss rate. Simulations of various TLB configurations show reductions in average TLB miss rates of 7.5% for unified TLBs by moving one shared library per application based on page-eviction patterns. A different relocation strategy based on the call graph, rather than page evictions, which relocated multiple shared libraries demonstrated significant reductions in the miss ratios of up to 27.8%, with the highest average miss-rate reduction, 9.6%, seen for I-TLBs. Thus, the eviction-pattern strategy worked best for unified TLBs, because it takes into account conflicts between and data pages. The call-graph methodology, which considers only conflicts between instruction pages, works better for split TLBs, such as those found on most modern processors. (The AMD Athlons have split L1 TLBs and unified L2 TLBs.) However, a methodology based on eviction patterns will be more sensitive to changes in input than will a method based on call graphs. Our experiments show that the eviction-pattern strategy also produced improvements for different input patterns, but the improvements are smaller than for the inputs on which the relocation was based. In summary, it is possible to produce significant savings in expensive TLB misses by changing the layout of shared libraries in the virtual address space. In the future, these strategies could be automated by using performance-monitoring hardware to drive dynamic changes to the shared-library layout.

5.1 Future Work

The data collected for this study shows that there are significant variations in the number of misses encountered by individual sets in the TLB mechanism and reductions in the TLB miss rates are possible through changes in the virtual memory layout of the shared libaries. However, for production machines, it is not practical to use a heavy weight instrumentation such as *Valgrind* and make manual changes to the layout of the shared libraries. Many processors have performance-monitoring hardware. This hardware could be used to collect information about which memory locations are causing TLB misses while the program is running normally. The performance-monitoring hardware could also be used to collect dynamic call-graph information. This information could be used for automated layout similar to the techniques developed by Pettis and Hansen [8] for function layout. Manually relocating one library to reduce the number of conflicts is not practical for applications with many libraries. The use of call-graph data to relocate libraries is an attempt to automate the relocation process. In Linux distributions such as Fedora, the prelink command runs on a regular basis to handle any new libraries loaded on the machine. Changing the prelink mechanism to use the data collected from the performance monitoring hardware and could reduce TLB conflicts.

In the eviction graphs used for the single-library relocation strategy, the number of nodes and transitions are hard coded with a preset threshold. These graphs can be generated with a set percentage of the total weighted values of edges coming in and out of the nodes (page). Similarly, the threshold for the transitions could be made user-definable.

To minimize the conflicts among data pages as seen with call-graph relocation strategy, the anonymous data regions could be represented using superpages.

6 References

[1] J. Bradley Chen, Anita Borg, Norman P. Jouppi. A Simulation-based study of TLB Performance. In *Proc. of the 19th Annual Symposium on Computer Architecture*, pages 114-123, May 1992.

[2] David Cortesi, Origin 2000 and Onyx2 Performance Tuning and Optimization Guide. Document Number 007-3430-003, Silicon Graphics Inc.. 2001. http://techpubs.sgi.com/

[3] Len DiMaggio. Automated GUI testing with Dogtail. In *Red Hat Magazine*, June 2006. http://www.redhat.com/magazine/020jun06/features/dogtail/

[4] Ulrich Drepper. Memory part 2: CPU caches. In *Linux Weekly News*, October 2007. http://lwn.net/Articles/252125/

[5] Gokul B. Kandiraju and Anand Sivasubramaniam. Going the distance for TLB prefetching: An appliction-driven study. *In Proc. of the 29th annual international symposium on Computer architecture*, pages 195-206, May 2002.

[6] Yousef A. Khalidi, Madhusudhan Talluri, Micheal N. Nelson, Dock Williams. Virtual Memory Support for Multiple Page Sizes. In *Proc. of the 4th Workshop on Workstation Operating Systems*, pages 104-109, Oct. 1993.

[7] Nicholas Nethercote. Dynamic Binary Analysis and Instrumentation. PhD Dissertation, University of Cambridge, November 2004.

[8] Karl Pettis and Robert C. Hansen. Profile Guided Code Positioning In *Proc. of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16-27. June 1990.

[9] Theodore H. Romer,Wayne H. Ohlrich, Anna R. Karlin. Reducing TLB and Memory Overhead Using Online Superpage Promotion. In *Proc. of the 22nd annual international symposium on Computer architecture*, pages 176-187, 1995.

[10] Mark Swanson, Leigh Stoller, and John Carter. Increasing TLB reach using superpages backed by shadow memory. *In Proc. of the 25th annual international symposium on Computer architecture*, pages 204-213, July 1998.

[11] Madhusudhan Talluri and Mark D.Hill. Surpassing the TLB Performance of superpages with Less Operating System Support. In *Proc. of the 6th Symposium on Architectural Support for Programming languages and Operating Systems*, pages 171-182, Oct. 1994.

[12] Ulrich Drepper. The Cost of Virtualization. *ACM Queue*, Vol 5, No. 7. http://www.acmqueue.com/modules.php?name=Content&pa=showpage&pid=524

[13] Moshe Bar. The Linux Process Model. Linux Journal. March 2000. http://www.linuxjournal.com/article/3814