MODIFIED SET ASSOCIATIVE TLB

By

ABDURASHID ABDURAHMAN

Bachelor of Science
Xinjiang University
Urumqi, Xinjiang
People's Republic of China
1991

Master of Science
Oklahoma State University
Stillwater, Oklahoma
2001

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2001

MODIFIED SET ASSOCIATIVE TLB

Thesis Approved:

Thesis Adviser

Dean of the Graduate College

ACKNOWLEDGEMENTS

I sincerely thank my adviser Dr. Nohpill Park, for his guidance, help, encouragement and continuous support in finishing this thesis. Special thanks are extended to my committee members Dr. G. E. Hedrick, Dr. Blayne Mayfield for their advice, cooperation and suggestions for the completion of this thesis. I would like to express my sincere gratitude to Dr. G. E. Hedrick for his precious help in organizing and proofreading this thesis.

So many people have helped me during the completion of this thesis. It is impossible to acknowledge them all personally. I extend my special thanks to Dr. Mansur Samadzadeh for his help for finding the needed input trace files.

I thank God for giving me intelligence and courage. My special gratitude is extended to my parents and brother, sisters for their continuous support for my education.

Finally, I would like to dedicate this thesis to my lovely wife Aisha for her priceless sacrifices during the last four years of my higher education at OSU.

TABLE OF CONENTS

Chapter Page

1. INTRODUCTION 1
   1.1 Background 1
   1.2 Motivation 1
   1.3 Terminology 3
2. LITERATURE REVIEWS 4
3. PRELIMINARIES 7
   3.1 Virtual Memory 7
   3.2 Paging and Page Table 7
   3.3 Locality of Memory References 9
   3.4 Translation Lookaside Buffer (TLB) 9
   3.5 Page Placement Policies 11
      3.5.1 Direct Mapped 11
      3.5.2 Fully Associative 11
      3.5.3 Set Associative 12
   3.6 Page Replacement Policies 12
      3.6.1 Random Replacement Policy 12
      3.6.2 LRU Replacement Policy 13
      3.6.3 LFU Replacement Policy 13

Chapter Page

3.6.4 MFU Replacement Policy 13

3.6.5 Optimal Replacement Policy 14

4. MODIFIED SET ASSSOCIATIVE TLB 15

4.1 Basic Idea 15

4.2 Mapping Functions Choosing Criteria 15

4.3 Overhead due to Multiple Mapping Function 16

4.4 Mapping Functions for the Modified Set Associative TLB 17

4.5 Which Replacement Policy for the Modified TLB ? 26

5. SIMULATION RESULTS AND DISCUSSIONS 28

6. CONCLUSION 44

REFERENCES 46

APPENDIX

Source Code for the Simulator of Modified 2-way Set Associative TLB 48

LIST OF TABLES

Table Page

1. Valid Page Numbers in a Traditional 2-way TLB After One Read 28
2. Valid Page Numbers in a Modified 2-way TLB After One Read 28
3. Miss Rates for a Modified, a Conventional 2-way Set Associative TLB and a Fully Associative TLB When the Page Size is 512 30
4. Miss Rates for a Modified, a Conventional 2-way Set Associative TLB and a Fully Associative TLB When the Page Size is 1024 30
5. Miss Rates for a Modified, a Conventional 4-way Set Associative TLB When the Page Size is 512 33
6. Miss Rates for a Modified, a Conventional 4-way Set Associative TLB When the Page Size is 1024 34
7. Miss Rates for a Modified, a Conventional 8-way Set Associative TLB When the Page Size is 128 36
8. Miss Rates for a Modified, a Conventional 8-way Set Associative TLB When the Page Size is 256 36
9. Miss Rates for a Modified, a Conventional 8-way Set Associative TLB When the Page Size is 512 37
10. Miss Rates for a Modified 2-way Set Associative TLB When the Page Size is 128 41

Table	Page

11. Miss Rates for a Conventional 8-way Set Associative TLB When the Page Size

is 128 41

LIST OF FIGURES

Figure Page

1. Paging hardware 8

2. TLB acts as a cache 10

3. An implementation of a modified 2-way TLB 20

4. Miss rate comparison for the program, li, when the page size is 512 31

5. Miss rates for the program, spice2g6, when the page size is 512 31

6. Miss rates for the program, nasa, when the page size is 1024 32

7. Miss rate comparison for the program, fpppp, when the page size is 1024 33

8. Miss rates for the program, espresso, when the page size is 512 34

9. Miss rates for the program, li, when the page size is 1024 35

10. Miss rate comparison for the program, li, when the page size is 128 38

11. Miss rates for the program, spice2g6, when the page size is 128 38

12. Miss rate comparison for the program, doduc, when the page size is 256 39

13. Miss rate comparison for the program, gcc, when the page size is 512 39

14. Miss rate comparison for the program, fpppp, when the page size is 512 40

15. Miss rate comparison for the program, li, when the page size is 1024 40

16. Miss rate comparison for the program, nasa, when the page size is 128 42

17. Miss rate comparison for the program, spice2g6, when the page size is 128 42

NOMENCLATURE

CPI	clock cycles per instruction
CPU	central processing unit
LFU	least frequently used
LRU	least recently used
MFU	most frequently used
MMU	memory management unit
NNR	not-used, not-written random
OR	bit wise inclusive or operation
TLB	translation lookaside buffer
XOR	bit wise exclusive operation
$+$	OR
$\oplus$	XOR

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

Virtual memory is supported in almost all modern computer systems [10]. In 1959, Kilburn et al. [8] introduced the concept of a one-level store*, known now as virtual memory, to allow larger programs than available physical memory. Since then, a number of new mechanisms have been created to utilize the advantages of virtual memory to benefit the computer systems. One of these techniques is paging. In a paging scheme, physical memory is broken into fixed sized blocks called frames, logical memory is also broken into blocks of the same size called pages. Every address generated by the CPU is divided into two parts: page number and page offset. We locate pages by using a full table called a page table that contains the base address of each page in physical memory. Since the page tables are stored in main memory, every memory access by a program can take at least twice as long: one memory access to obtain the physical address and another access to get the data. Since most of the references exhibit both temporal and spatial locality, modern machines include a special cache that keeps track of recently used translations. This special address translation cache is called Translation-Lookaside Buffer (TLB) [6].

1.2 MOTIVATION

* Today this is frequently called multi-lever store.

As the microprocessor improves its performance at a speed of 1.35 times per year, the memory performance must improve at a speed of several percent per year. The gap between CPU performance and main memory has been widening with higher performance CPUs creating performance bottlenecks for memory access instructions [7].

A recent study shows that the memory system may stall the CPU for over 50% of the execution time [12]. The many recent studies on memory system behavior and performance have concentrated almost exclusively on cache design [11,15]. Little attention has been given to TLB performance. Early studies have shown that TLB miss penalties consume 6% of all machine cycles [3] and 4% of execution time [4]. This effect is even larger in today's modern computers that have a larger memory size. Because in modern computers TLB can be in the critical path of memory access, good TLB performance is essential to good overall performance of a machine [12].

A TLB keeps the average translation cost low since instruction accesses generally exhibit repetitive memory reference behavior in keeping with the principle of spatial and temporal locality. The efficiency of the virtual memory mechanism is measured by miss rate and miss penalty [1]. Reducing TLB misses and miss penalties becomes increasingly important to overall performance of machines. As with cache misses, TLB misses can be classified into three categories, capacity miss, compulsory miss and conflict miss.
Chen et al. [2] have shown that TLB misses are dominated primarily by capacity misses, because the mapping size of the TLB is not big enough to map the entire working set of the program.

Technological and architectural trends have led to increased memory sizes, decreased Clock Cycles Per Instruction (CPI), and larger working set programs. Both factors cause more significant performance impact due to TLB misses. It is therefore highly desirable to improve TLB hit ratios in future systems.

1.3 TERMINOLOGY

For the sake of clarity, a few terms that are used in this paper are defined as follows.

- *associativity* is the number of blocks in each set.
- *hit* is the existence of the data requested by the processor in the upper level [6].
- *hit ratio* is the fraction of memory accesses found in the upper level [6].
- *LRU Stack* is a stack that maintains a list of address references ordered according to how recently they were accessed with the most recent at the top [13].
- *miss* is the nonexistence of the data request by the processor in the upper level [6].
- *miss penalty* is the time to replace a block in the upper level with the corresponding block from lower level, plus the time to deliver this block to the processor [6].
- *miss rate* is 1- *hit ratio*.
- *recency* is the depth of a reference in the *LRU Stack* [13].
- *TLB reach* is the amount of memory that can be accessed without causing a TLB fault [16].

CHAPTER 2

LITERATURE REVIEWS

Traditional approaches for increasing the TLB hit ratio; that is, decreasing the TLB miss rate, include using more TLB entries and/or bigger page sizes. Increasing the number of TLB entries is expensive and inefficient. Allowing bigger page sizes can result in poor memory utilization, due to fragmentation problems. Therefore, these techniques often are subject to significant costs in implementation [10].

Designers have used a wide variety of associativity in TLBs. Some systems use fully associative TLBs because a fully associative mapping has a lower miss rate. However with a fully associative mapping, choosing the entry to replace becomes tricky since implementing a hardware LRU scheme is expensive [6].

Using variable page sizes involves complexities in both hardware and software implementations, and may not be incorporated easily into many existing architectures [10].

Recently a number of manufacturers introduced split TLB designs in which the TLB is split into data and instruction translation. The unpredictable nature of process reference patterns makes the predicting the optimal split of the TLB into portions impossible without prior knowledge of the application. An incorrectly selected partition size can lead to thrashing within a partition with associated loss in performance.

To solve the above problem, Channon and Koch [1] introduced a re-configurable partitioned TLB. Because this approach involves the dynamic partitioning the hardware and an adaptive algorithm for managing the TLB, it is expensive both in hardware and software implementation.

Liu [10] introduced a Multiple-Page Translation Lookaside Table MPTLB in addition to the conventional TLB. When a requested virtual address misses both TLB and MPTLB the slower translation process is invoked, which translates k pages together into a new entry at the MPTLB. One potential usage of the MPTLB is to serve only for selected types of memory references (e.g., vector operands, or accesses to special data areas marked by software). Therefore its usage is limited to some specific applications.

Other methods to improve TLB performance are to use software pre-fetching and caching. Saulsbury et al. [13] introduced a pre-fetching technique based on the recency of references. This technique uses an LRU stack to measure the recency of memory references. Upon a TLB miss, it predicts the translation for the next miss based on the recency of the miss currently being handled. If the prediction is incorrect, then a full TLB reload occurs with associated loss in performance. However, these type of methods are useful for dynamically allocated data structures such as kernel data structures, and they also add extra work for the operating system. Also, software managed TLBs can exhibit a high miss penalty [5].

Swanson et al. [16] proposed a mechanism of increasing the TLB reach by introducing a notion of shadow pages. A secondary MMU and a secondary TLB are placed in the main memory, makes the mechanism expensive and complex. Also, shadow regions are not supported on extremely high end machines [16].

Lee et al. [9] proposed a dual TLB structure which consists of two conventional TLBs; a conventional small page (4KB size) TLB and a conventional large page (16KB size) TLB. Both TLBs are designed as fully associative. This structure is simpler than the one proposed by Swanson et al. [16]. However, when there is a miss in both TLBs, we must flush both TLBs which invokes a slower translation process.

Another well-known alternative is the set associative organization. Increasing the associativity usually decreases the miss rate [6]. Previous studies have showed that, for a cache size larger than 64KB, direct mapped caches exhibit hit ratios nearly as well as set-associative caches [6]. But TLB is relatively small cache in size. A typical TLB has entries between 32 and 4096 [6]. This paper introduces a modified set associative TLB design that gives lower miss rate than a conventional set associative TLB.

CHAPTER 3

PRELIMINARIES

3.1 VIRTUAL MEMORY

Many years ago, when people first encountered programs that had larger size than available memory size, they usually split the program into pieces called overlays [17]. The overlays were kept on disk and swapped in and out of memory by the operating system. Since the generation of overlays was very time consuming, and complex, to overcome this drawback a way known as virtual memory was soon developed [14].

The basic idea behind the virtual memory is only a portion of the program is kept in main memory, the rest is stored in secondary storage. When the system needs other portions of the program, any of the well-known general schemes can be used to swap the portions between main memory and second storage. Therefore, program size is no longer a constraint for programmers and multiprogramming becomes feasible.

In virtual memory systems, there are two types of addresses for programs: virtual addresses and physical addresses. Virtual addresses are produced by programs and from the virtual address space. Physical addresses are the actual physical address in memory.

3.2 PAGING AND PAGE TABLE

There are two common techniques used in virtual memory system: paging and segmentation. We do not discuss segmentation here because our study is related to

paging. In a paging scheme, physical memory is broken into fixed sized blocks called frames. Virtual memory, also known as logical memory, is also broken into blocks of the same size called pages. Every address generated by the CPU is divided into two parts: page number and page offset.

A Page table is a table that contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory management unit (MMU) [18]. When a virtual address is sent to the MMU, the MMU determines the page number (*p*) to which the virtual address belong, gets the corresponding frame number (*f*) according to the page table, translates the virtual address into physical address, and sends it to the bus. The paging model of memory is given in figure 1.

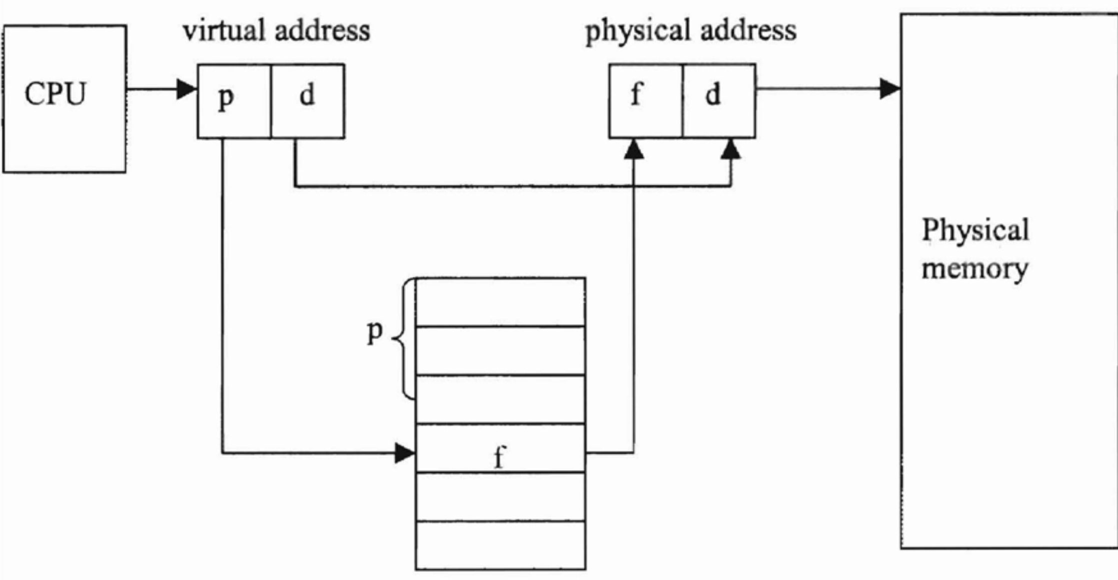


Figure 1. Paging Hardware

3.3 LOCALITY OF MEMORY REFERENCES

Locality is the property that references in a program tend to cluster into groups in time and/or space [18]. There are two types of locality: spatial and temporal.

Temporal locality is with respect to time clustering for a set of pages. That is, if a set of pages are referenced during a given time interval, it is likely that they will be referenced again soon [7].

Spatial locality occurs when two successive references have adjacent virtual spaces. In other words, if a word w is referenced at time t, then words in the range of $w-i$ to $w+i$ for some small i are likely to be referenced at time $t+1$ [7].
Most of the programs exhibit good spatial and/or temporal locality.

3.4 TRANSLATION LOOKASIDE BUFFER (TLB)

Since the page tables are stored in main memory, every memory access by a program can take at least twice as long: one memory access to obtain the physical address and another access to get the data. Also, because most of the references exhibit both temporal and spatial locality, the key to improving access performance is to rely on the locality of the references in the page table. When a translation for a virtual page number is used, it will probably be needed again in the near future. Accordingly, modern machines include a special cache that keeps track of recently used translations. This special address translation cache is called translation-lookaside buffer (TLB).

A TLB is a cache that holds only page table mappings. Thus, each tag entry in the TLB holds a portion of the virtual page number, and each data entry of the TLB holds a physical page number. On each reference, we look up the virtual page number in the TLB. If there is a hit, the physical page number is used to form the address. If there is a miss, then we look up the page table. Because the TLB has many fewer entries than the number of pages in main memory, TLB misses are much more frequent than true page faults. This is another reason why the improving TLB hit ratio is important. Figure 2 shows how the TLB acts as a cache for the page table references.

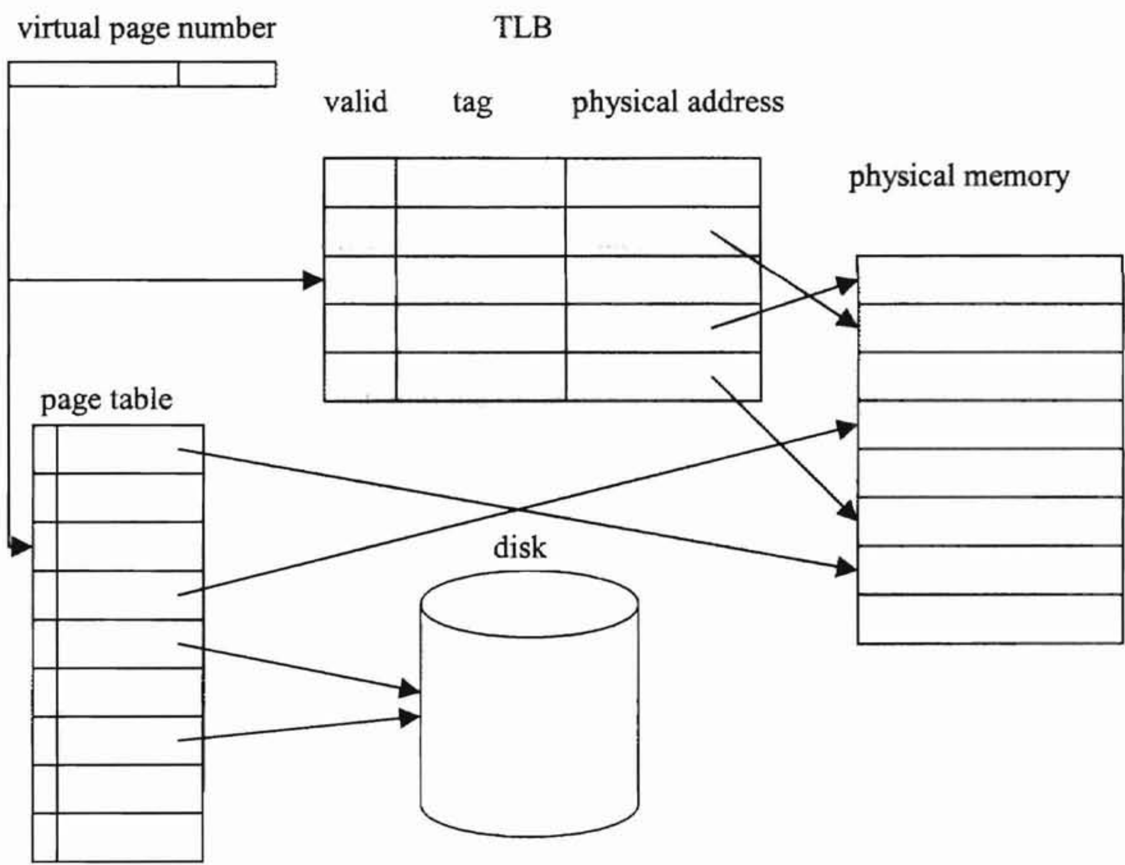


Figure 2. TLB acts as a cache

3.5 PAGE PLACEMENT POLICIES

Since TLB is a cache used for fast translation of page numbers, conventional cache block placement policies apply. Most commonly used block placement policies are direct mapped, fully associative and set-associative schemes.

3.5.1 Direct Mapped

In a direct mapped placement scheme, TLB is one set of entries, and a page can go exactly one entry in the TLB. Assume the number of entries in the TLB is n, then a page goes to the *entry* = *page number* MOD n. A direct mapped scheme can be considered as a one-way set associative scheme.

The direct mapped scheme is very common among cache implementations, since it is easy to implement. However, it performs poorly compared to other schemes in terms of hit ratio for caches that the cache size is less than 64KB. The hit ratio may drop sharply if many addressed blocks have to map into the same block frame [7]. For this reason, direct mapped caches tend to use a larger cache size with more block frames to avoid contention.

3.5.2 Fully Associative

In a fully associative scheme, the TLB is again one set of entries as in a direct mapped scheme, but a page can go any one of the entries in the TLB. When a page is referenced, we must search all entries in parallel to decide whether it is a miss or a hit. These

comparators significantly increase the hardware cost, effectively making fully associative placement practical only for small caches.

3.5.3 Set Associative

The middle range of design between direct mapped and fully associative is called set-associative. In a set-associative TLB, there are fixed number of entries in each block; a set-associative TLB with n blocks is called n-way set associative TLB. An n-way set-associative TLB consists of a number of sets, each of which consists of n blocks.

The advantage of increasing the degree of associativity is that it usually decreases the miss rate [6]. The improvement in miss rate comes from reducing misses that compete for the same location. But as discussed in section 2, when the cache size is large (larger than 64KB), a direct mapped scheme performs as well as a fully associative and a set-associative scheme.

3.6 PAGE REPLACEMENT POLICIES

If there is a miss on a page reference, then we must decide which entry should be replaced from the TLB. Following are some general replacement policies.

3.6.1 Random Replacement Policy

In a random replacement policy, the victim page is chosen randomly. Therefore, it may not utilize locality of references. However, since random replacement policy is easy to implement, it is commonly used in small size caches and in simulations.

3.6.2 LRU Replacement Policy

In a LRU (Least Recently Used) replacement policy, the victim page to be replaced is the one that has not been used for the longest time [14]. In general, LRU replacement policy exhibits lower miss rate than random replacement policy. But, it is quite expensive to implement, since there is a time stamp associated with each reference. There is a modification of LRU replacement policy that works almost as well, but less expensive to implement. This modification is described later in this thesis.

3.6.3 LFU Replacement Policy

LFU (Least Frequently Used) replacement policy is an approximation to LRU. Instead of having a time tag for each reference, each page is attached with a frequency counter. The page to be replaced is the one with lowest frequency count. This policy suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again. Since it was used heavily, it has a large count and remains in cache even though it is no longer needed [14].

3.6.4 MFU Replacement Policy

The MFU (Most Frequently Used) page replacement is based on the argument that the page with the smallest count was probably just brought in and yet to be used [14]. MFU has same characteristics as LFU.

3.6.5 Optimal Replacement Policy

Optimal replacement policy simply replace the page that will be not be used for the longest period of time [14]. An optimal page replacement has the lowest page fault rate of all policies. Unfortunately, it is difficult to implement, because it requires future knowledge of the reference string. It is used mainly for comparison studies.

CHAPTER 4

MODIFIED SET ASSCOCIATIVE TLB

4.1 BASIC IDEA

In a set-associative TLB, there are a fixed number of entries in each block; a set-associative TLB with n blocks is called n-way set associative TLB. An n-way set-associative TLB consists of a number of sets, each of which consists of n blocks. In an n-way set associative TLB, a page number p that mapped onto a set S_i (i = 0, 1, ... , #of sets), can be placed in the entry $f(p)$ in one of the block B_j ($j = 0,\ldots,n$) , where f is the mapping function. When n+1 pages contend for the same entry in a set S_i, one of the pages should be replaced from the TLB, because $f(p_1) = f(p_2) = \ldots = f(p_n) = f(p_{n+1})$, and there are only n blocks available in the set. The modified set associative TLB uses different mapping functions for different blocks so that if two pages contend for the same entry in block i, they have a low probability of being mapped onto the same entry in block j. For example, in a 2-way set-associative TLB, a page p can be mapped onto entry $f_0(p)$ in block 0 or onto entry $f_1(p)$ in block 1, so that for two distinct pages p_1 and p_2, $f_0(p_1) = f_0(p_2)$ but $f_1(p_1) \neq f_1(p_2)$.

4.2 MAPPING FUNCTIONS CHOOSING CRITERIA

In the previous section, we gave the idea of scattering data within blocks, by using different mapping functions for different blocks. In order to achieve above expectation, the mapping functions should have some special properties.

1. Assumption

For each entry in the TLB, the numbers of pages that may be mapped onto this entry are equal.

2. Inter-Block Dispersion

Mapping functions should have the inter-block dispersion property. That is, the set of pages that can be mapped onto an entry of block *i* will be equally distributed over the entries in other blocks.

3. Intra-Block Dispersion

Since many applications exhibit spatial locality, mapping functions should avoid mapping neighboring pages into the same entries in any block. In other words, the mapping function f_i should limit the number of conflicts when mapping a memory block within block *i*.

4.3 OVERHEAD DUE TO MULTIPLE MAPPING FUNCTIONS

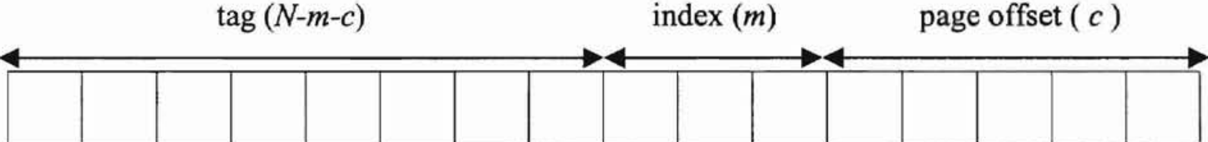
Since multiple functions are used in address translation, some extra delay is introduced as compared to using a single mapping function in conventional set associative TLB. But as long as we perform address computation in a noncritical stage of the pipeline and do not lengthen the pipeline cycle, the effect on overall performance can be negligible. Nowadays, in most of the new generation microprocessors, the address translation stage is not the critical stage of the pipeline. Surely, this works on non-pipelined machines as well as pipelined machines. Therefore, the mapping function should be simple to implement in hardware, and should introduce few extra gates and delays.

4.4 MAPPING FUNCTIONS FOR THE MODIFIED SET ASSOCIATIVE TLB

Assume the virtual memory address generated by the CPU is N bits, the page size is 2^c bytes, and the number of entries in each block is 2^m.

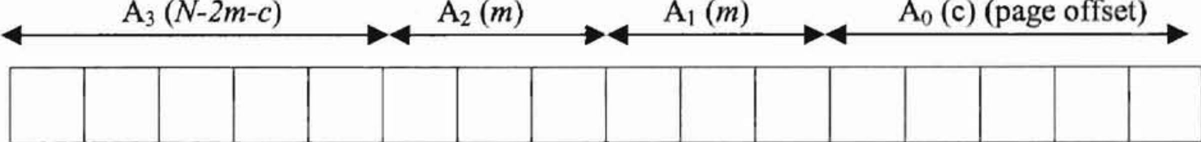
In a conventional n-way set associative TLB, the N bit memory address is divided into three parts: page offset (lowest c bits), entry index (mid m bits), and tag (highest N-m-c bits).

For example, let $N = 16$, $c = 5$ and $m = 3$.



A page p can be placed in an entry determined by the index bits on any one of the n blocks. That means a set of pages with the same index bits has only one entry associated with it. When n+1 pages contend for that entry, one of them must be rejected.

A Modified TLB scheme avoids this situation by using different indexes for different blocks. Architecture of a modified TLB is very simple. An N bit memory address is divided into four parts as shown below.



The index is determined by A_1 and A_2, and by using a different function for each block, multiple indices are generated for a set of pages that have same index in a conventional set-associative TLB.

Here, we present some preliminary work about memory addresses. Given an N-bit memory address, let $q = N\text{-}2m\text{-}c$ then

$A_3 = a_{N-1}2^{q-1} + a_{N-2}2^{q-2} + \ldots + a_{N-q}2^0$,

$A_2 = a_{N-q-1}2^{m-1} + a_{N-q-2}2^{m-2} + \ldots + a_{N-q-m}2^0$,

$A_1 = a_{N-q-m-1}2^{m-1} + a_{N-q-m-2}2^{m-2} + \ldots + a_{N-q-2m}2^0$,

$A_0 = a_{N-q-2m-1}2^{c-1} + a_{N-q-2m-2}2^{c-2} + \ldots + a_0 2^0$.

Now we have

$A_3 2^{2m+c} + A_2 2^{m+c} + A_1 2^c + A_0 = (a_{N-1}2^{q-1} + a_{N-2}2^{q-2} + \ldots + a_{N-q}2^0)2^{2m+c} + (a_{N-q-1}2^{m-1} + a_{N-q-2}2^{m-2} + \ldots + a_{N-q-m}2^0)2^{m+c} + (a_{N-q-m-1}2^{m-1} + a_{N-q-m-2}2^{m-2} + \ldots + a_{N-q-2m}2^0)2^c + a_{N-q-2m-1}2^{c-1} + a_{N-q-2m-2}2^{c-2} + \ldots + a_0 2^0 = a_{N-1}2^{q-1+2m+c} + a_{N-2}2^{q-2+2m+c} + \ldots + a_{N-q}2^{2m+c} + a_{N-q-1}2^{2m+c-1} + \ldots + a_0 2^0 = a_{N-1}2^{N-1} + a_{N-2}2^{N-2} + \ldots + a_{N-q}2^{2m+c} + a_{N-q-1}2^{2m+c-1} + \ldots + a_1 2 + a_0$.

That is, any memory address can be written in the form of $A_3 2^{2m+c} + A_2 2^{m+c} + A_1 2^c + A_0$.

Now, let us consider a 2-way set associative TLB with 2^m entries in each block. Let us assume that each page size is 2^c bytes. Then a virtual memory address A in the form of $A = A_3 2^{2m+c} + A_2 2^{m+c} + A_1 2^c + A_0$ may be mapped onto the entry $f_0(A) = A_1 \oplus A_2$ on block 0, or onto the entry $f_1(A) = g(A_1) \oplus A_2$ on block 1, where g is one-bit circular shift, and $\oplus$ is exclusive OR on each bit. g can be a right circular shift or a left circular shift. For example, let the binary representation of A_1 be 1011 then $g(A_1)$ = 1101 or 0111.

More generally, in an n-way set associative case, a page with the address $A = A_3 2^{2m} + A_2 2^m + A_1$ may be mapped:

onto the entry $f_0(A) = A_1 \oplus A_2$ on block 0 or

onto the entry $f_1(A) = g(A_1) \oplus A_2$ on block 1 or

onto the entry $f_2(A) = g^2(A_1) \oplus A_2$ on block 2 or

....

onto the entry $f_{n-1}(A) = g^{n-1}(A_1) \oplus A_2$ on block n-1.

As discussed in the previous section, the mapping function should be simple to implement in hardware, and should introduce few extra gates and delays. The exclusive OR (XOR) operation is used instead of inclusive OR to avoid the situation where all bits of A_1 are 1. When all bits of A_1 are 1, regardless the value of A_2, $A_1 + A_2 = g(A_1) + A_2 = g^2(A_1) + A_2 = \ldots g^{n-1}(A_1) + A_2$ thus even if multiple functions are used, all pages still be mapped to the same entry, and we cannot reach the goal of scattering the data among entries. Here + denotes the inclusive OR operation. A hardware implementation of a modified set associative TLB and a conventional TLB is almost the same, only fewer XOR gates are introduced.

Figure 3 shows the implementation of a sample modified 2-way TLB.

For the sake of simplicity, here we assume a 16-bit virtual address, the number of entries in each block is 8, and page size is 64 bytes.

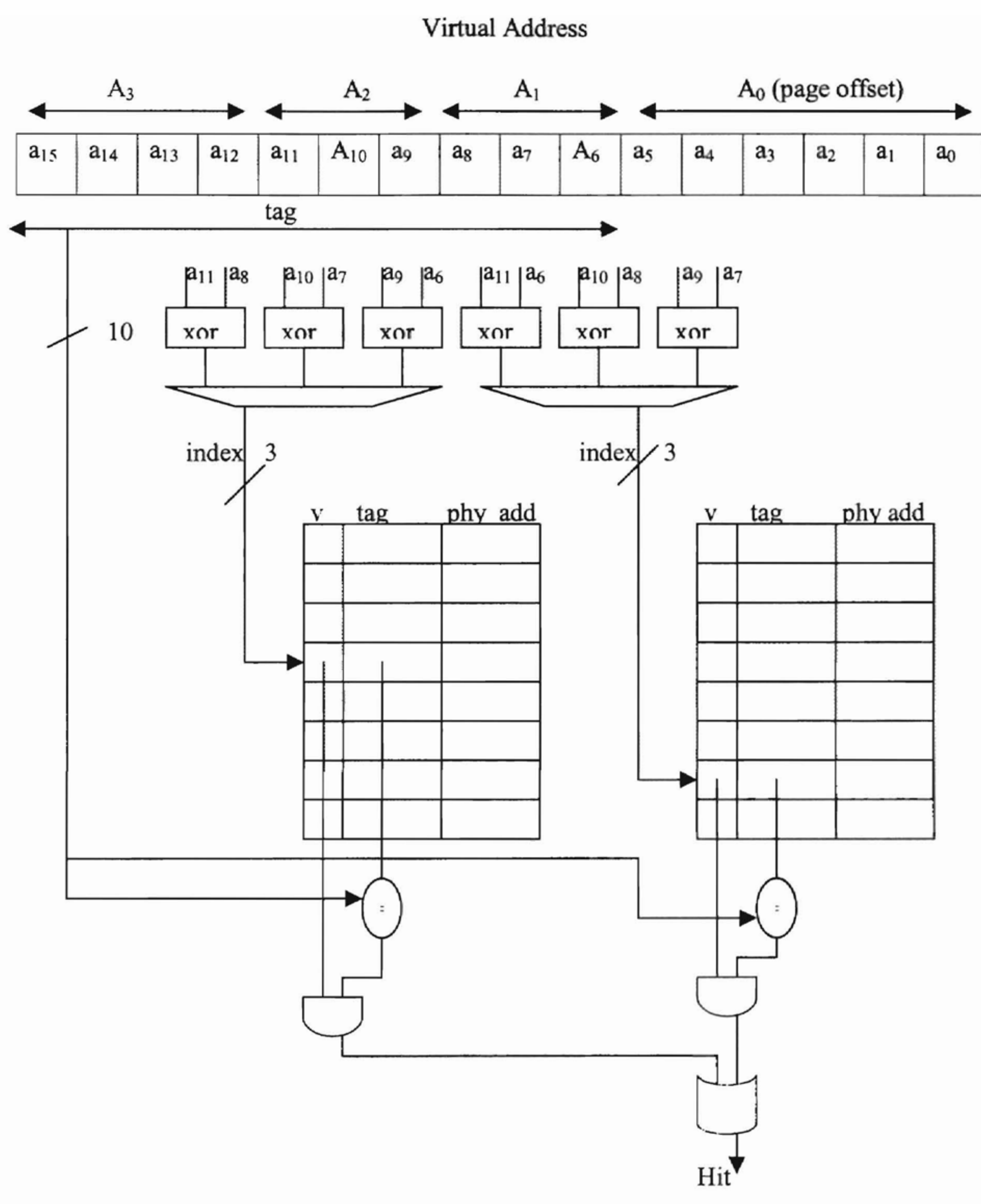


Figure 3. An implementation of a modified 2-way TLB

Claim 1. Mapping functions described above satisfy the Inter-Block Dispersion property. In other words, the set of distinct pages that can be mapped onto an entry of block i have a low probability of being mapped to the same entry in block j, where $0 \le i < j \le n$.

In mathematical terms, let $P = \{P_0, P_1, \ldots, P_{k-1}\}$ be a set of k distinct pages to be mapped onto an entry on block i, i.e. $f_i(P_0) = f_i(P_1) = \ldots = f_i(P_{k-1})$ for block i, where k is the number of entries in the block. Then the probability of these pages to be mapped into the same entry in block j is $2/(k+1)$, in other words $p[f_j(P_0) = f_j(P_1) = \ldots = f_j(P_{k-1})] = 2/(k+1)$.

Proof: Before proving Claim 1, let us first find the probability $p[A \oplus B = C \oplus D]$, where A, B, C, and D are any m bit numbers such that $A \neq C$, $A \neq D$, $B \neq C$, and $B \neq D$. Since A, B, C, D are m bit numbers, the range of A, B, C, D is $\{0, k\text{-}1\}$, where $k = 2^m$.

Let |XOR| be the total number of distinct bit-wise XOR operations on k distinct numbers between 0 and k-1. $A \oplus B$ and $B \oplus A$ are considered same XOR operation. Start with 0, there are k distinct bit-wise XOR operations, namely $0 \oplus 0$, $0 \oplus 1$, …,$0 \oplus k\text{-}2$,$0 \oplus k\text{-}1$. For 1, there are k-1 XOR operations, $1 \oplus 1$,$1 \oplus 2$,$1 \oplus 3$, … ,$1 \oplus k\text{-}1$. Since $1 \oplus 0$ is already been included in the set of XOR operations for 0. Similarly, there are k-2 distinct bit-wise XOR operations for 2. Therefore, $|XOR| = k + (k\text{-}1) + (k\text{-}2) + \ldots + 2 + 1 = k(k+1)/2$. Thus $A \oplus B$ and $C \oplus D$ are one of these $k(k+1)/2$ XOR operations. Now we need to find how many of these $k(k+1)/2$ numbers are equal.

Since $A \oplus A = 0$ for any number A, there are total k zeros among these $k(k+1)/2$ numbers. Thus the result of XORing two numbers being zero has highest probability among k

numbers between 0 and k-1 and its probability is $\frac{k}{\frac{k(k+1)}{2}} = \frac{2}{k+1}$. Therefore the probability $p[A \oplus B = C \oplus D] = 2/(k+1)$.

Now let us prove Claim 1.

For $\forall\ P_s, P_t \in P$, let $P_s = A_3 2^{2m} + A_2 2^m + A_1$ and $P_t = B_3 2^{2m} + B_2 2^m + B_1$, where A_2, A_1, B_2, B_1 are m bit numbers. In order prove the claim, it suffices to prove that for any P_s and P_t in P, if $f_i(P_s) = f_i(P_t)$ then $p[f_j(P_s) = f_j(P_t)] = 2/(k+1)$ for any $0 \le i < j \le n$, where n is the number of blocks in each set of the TLB.

By the definition of the mapping function, we have

$$f_j(P_s) = g^j(A_1) \oplus A_2 \text{ and } f_j(P_t) = g^j(B_1) \oplus B_2.$$

There are several cases to consider.

Case1. $A_1 = B_1$, $A_2 = B_2$.

In this case, $f_i(P_s) = f_i(P_t)$ and $f_j(P_s) = f_j(P_t)$ all the time. But we can avoid this situation simply by choosing the number of set in the TLB not equal to 2^r, where $m \le r \le 2m$, so that $P_s = A_3 2^{2m} + A_2 2^m + A_1$ and $P_t = B_3 2^{2m} + B_2 2^m + B_1$ are mapped to different sets and they don not contend for the same entry in the same block.

Case 2. $\begin{cases} A_1 = B_1 \\ A_2 \neq B_2 \end{cases}$ or $\begin{cases} A_1 \neq B_1 \\ A_2 = B_2 \end{cases}$

Assume $A_1 = B_1$ and $A_2 \neq B_2$, then $f_i(P_s) = g^i(A_1) \oplus A_2 = C \oplus A_2$ and $f_i(P_t) = g^i(B_1) \oplus B_2 = C \oplus B_2$ for some C. Let $a_{m-1}\ a_{m-2} \ldots\ a_1\ a_0$, $b_{m-1}\ b_{m-2} \ldots b_1\ b_0$, $c_{m-1}\ c_{m-2} \ldots\ c_1\ c_0$ be the bit

representation of A_2, B_2 and C respectively. Because A_2 is not equal to B_2, they differ at least at one bit position, call it the jth position. Then there are 4 subcases.

Subcase 1. $c_j = 0$, $a_j = 0$, $b_j = 1$, but $c_j \oplus a_j = 0$ and $c_j \oplus b_j = 1$.

Subcase 2. $c_j = 0$, $a_j = 1$, $b_j = 0$, but $c_j \oplus a_j = 1$ and $c_j \oplus b_j = 0$.

Subcase 3. $c_j = 1$, $a_j = 0$, $b_j = 1$, but $c_j \oplus a_j = 1$ and $c_j \oplus b_j = 0$.

Subcase 4. $c_j = 1$, $a_j = 1$, $b_j = 0$, but $c_j \oplus a_j = 0$ and $c_j \oplus b_j = 1$.

Therefore, in any subcase $f_i(P_s) \neq f_i(P_t)$, in other words if $A_1 = B_1$ and $A_2 \neq B_2$, then $f_i(P_s)$ can not be equal to $f_i(P_t)$.

Same argument when $A_1 \neq B_1$ and $A_2 = B_2$.

Case 3. $g^i(A_1) = B_2$ and $g^i(B_1) = A_2$.

We need to consider

$$f_j(P_s) = g^j(A_1) \oplus A_2 = g^{j-i}(g^i(A_1)) \oplus A_2 = g^{j-i}(B_2) \oplus A_2 \text{ and}$$

$$f_j(P_t) = g^j(B_1) \oplus A_2 = g^{j-i}(g^i(A_1)) \oplus A_2 = g^{j-i}(A_2) \oplus B_2.$$

Since $A_2 \neq B_2$, above equations can be written as

$$f_j(P_s) = g^{j-i}(B_2) \oplus A_2 = C \oplus A_2 \text{ and}$$

$$f_j(P_t) = g^{j-i}(A_2) \oplus B_2 = D \oplus B_2 \text{ for some } C \text{ and } D \text{ such that } C \neq D.$$

Thus, as we have shown at the beginning of the proof, $p[f_j(P_s) = f_j(P_t)] = 2/(k+1)$.

Case 4. None of the above.

In all other cases, $f_j(P_s)$ and $f_j(P_t)$ can be written as $f_j(P_s) = A \oplus B$ and $f_j(P_t) = C \oplus D$ for some A, B, C, D such that none of them are equal. Again we have shown at the beginning of the proof that $p[f_j(P_s) = f_j(P_t)] = 2/(k+1)$.

Claim 2. The mapping function satisfies the intra-block dispersion property. In other words, neighboring pages have low probability to be mapped onto the same entry on any block j.

In mathematical terms, let $P = \{P_{k/2}, P_{k/2+1}, ..., P_0, P_1, ..., P_{k/2-1})$ be a set of k adjacent pages. Then the probability of these pages being mapped onto the same entry in block j is $2/(k+1)$, where k is the number of entries in each block, i.e.

$$p[f_j(P_{k/2}) = f_j(P_{k/2+1}) = \ldots = f_j(P_{k/2-1})] = 2/(k+1).$$

Proof: For $\forall\ P_s, P_t \in P$, let $P_s = A_3 2^{2m} + A_2 2^m + A_1 + s$ and $P_t = A_3 2^{2m} + A_2 2^m + A_1 + t$, where $s, t \in \{k/2, k/2\text{-}1\}$, $k = 2^m$ and A_2 and A_1 are m bit numbers.

Again it suffices to prove that for any P_s and P_t in P, $p[f_j(P_s) = f_j(P_t)] = 2/(k+1)$ for any $0 \leq j \leq n$, where n is the associativity of the TLB.

There are four cases to consider.

Case 1. $\begin{cases} A_1 + s > 0 \\ A_1 + t > 0. \end{cases}$

There are two subcases.

Sub case 1. $\begin{cases} A_1 + s \geq 2^m \\ A_1 + t > 2^m \end{cases}$ or $\begin{cases} A_1 + s > 2^m \\ A_1 + t \geq 2^m \end{cases}$

In either case, P_s and P_t can be written as

$P_s = A_3 2^{2m} + (A_2 + 1)2^m + B$ for some B and

$P_t = A_3 2^{2m} + (A_2 + 1)2^m + C$ for some C.

Now we have, $f_j(P_s) = g^j(B) \oplus (A_2 + 1)$ and

$$f_j(P_t) = g^j(C) \oplus (A_2+1)\,.$$

As showed in the proof of Claim 1, $f_j(P_s) \neq f_j(P_t)$ in this case.

Subcase 2. $\begin{cases} A_1 + s \geq 2^m \\ A_1 + t < 2^m \end{cases}$ or $\begin{cases} A_1 + s < 2^m \\ A_1 + t \geq 2^m. \end{cases}$

If $A_1 + s \geq 2^m$ and $A_1 + t < 2^m$, then

$P_s = A_3 2^{2m} + (A_2+1)2^m + B$ for some B, and

$P_t = A_3 2^{2m} + A_2 2^m + A_1 + t\,.$

If $A_1 + s < 2^m$ and $A_1 + t \geq 2^m$, then

$P_s = A_3 2^{2m} + A_2 2^m + A_1 + s$ and

$P_t = A_3 2^{2m} + (A_2+1)2^m + C$ for some C.

In either case $p[f_j(P_s) = f_j(P_t)] = 2/(k+1)$ as proved in Claim 1.

Case 2. $\begin{cases} A_1 + s < 0 \\ A_1 + t > 0 \end{cases}$ or $\begin{cases} A_1 + s > 0 \\ A_1 + t < 0 \end{cases}$

Assume $A_1 + s < 0$ and $A_1 + t > 0$. Again there are two subcases.

Subcase 1. $A_1 + t \geq 2^m$.

$P_s = A_3 2^{2m} + A_2 2^m - B = B_3 2^{2m} + B_2 2^m + B_1$ for some B, B_1, B_2, B_3 and

$P_t = A_3 2^{2m} + (A_2+1)2^m + C$ for some C.

Subcase 2. $A_1 + t < 2^m$.

$P_s = A_3 2^{2m} + A_2 2^m - B = B_3 2^{2m} + B_2 2^m + B_1$ for some B, B_1, B_2, B_3 and

$P_t = A_3 2^{2m} + A_2 2^m + C$ for some C, where $C = A_1 + t$.

Therefore in either of the subcases, $p[f_j(P_s) = f_j(P_t)] = 2/(k+1)$ by the proof of Claim 1.

The proof of the case when $A_1 + s > 0$ and $A_1 + t < 0$ follows the same argument.

Case 3. $\begin{cases} A_1 + s < 0 \\ A_1 + t < 0. \end{cases}$

In this case, $P_s = A_3 2^{2m} + A_2 2^m - B = B_3 2^{2m} + B_2 2^m + B_1$ for some B, B_1, B_2, B_3, and

$$P_t = A_3 2^{2m} + A_2 2^m - C = C_3 2^{2m} + C_2 2^m + C_1 \text{ for some } C, C_1, C_2, C_3.$$

Again $p[f_j(P_s) = f_j(P_t)] = 2/(k+1)$ by Claim 1.

Case 4. Either $A_1 + s = 0$ or $A_1 + t = 0$.

Consider $A_1 + s = 0$, then $P_s = A_3 2^{2m} + A_2 2^m$ and $P_t = A_3 2^{2m} + A_2 2^m + A_1 + t$. Regardless of the value of $A_1 + t$, $p[f_j(P_s) = f_j(P_t)] = 2/(k+1)$ by the proof of Claim 1, except the case that $A_1 + t < 0$ and $P_t = A_3 2^{2m} + A_2 2^m + A_1 + t = B_3 2^{2m} + B_2 2^m$ for some B_2 and B_3 in which case $f_j(P_s) \neq f_j(P_t)$ simply because $A_2 \neq B_2$.

Same argument holds for the case $A_1 + t = 0$.

4.5 WHICH PAGE REPLACEMENT POLICY FOR THE MODIFIED TLB?

In section 3.5, we discussed some of the most common page replacement policies. Generally LRU or Random replacement policies are used in set associative TLBs.

For the modified TLB scheme, we may use the revised random replacement policy called Not-used Not-replaced Random (NNR) Replacement Policy as discussed below.

Not-used Not-replaced Random (NNR) Replacement Policy:

We associate a reference bit for each entry. This reference bit is asserted when the entry is accessed. When there is a miss, the victim page is selected among *n* possible pages in the following priority order.

1) Randomly among the entries for which the reference bit is clear. In other words, a victim page is selected among the pages that never been accessed again since loaded.
2) Randomly among the entries for which the reference bit is set but the dirty bit is clear. That is, if all the pages have been accessed, then select among the pages that have not been replaced yet.
3) Randomly among the entries for which the reference bit and dirty bit are set.

Eventually all the reference bits will be set, therefore we reset the reference bit at some point.

CHAPTER 5

SIMULATION RESULTS AND DISCUSSIONS

First let us look at a simple example to illustrate how the modified TLB may improve hit ratio. For the sake of simplicity, let us consider a 2-way set associative TLB with 16 entries. A pseudo-random sequence of 16 page numbers within the range of $\{0,\ldots, 2^{20}-1\}$ was generated randomly. The following tables show the valid page numbers in the TLB after reading the sequence once. The number inside parentheses on *Entry* column is the number of pages that mapped to this entry. For example, 0(6) denotes 6 pages were mapped onto entry 0.

Entry / *Block*	0(6)	1(0)	2(4)	3(0)	4(1)	5(2)	6(1)	7(2)
Block0	594368		919746		697940	828477	532238	119839
Block1	973104		863442			1021125		973903

Table 1. Valid page numbers in a traditional 2-way TLB after one read.

Entry / *Block*	0(3)	1(1)	2(2)	3(1)	4(1)	5(2)	6(4)	7(2)
Block0	863442		919746			930024	973104	532238
Block1	697940	828477	926800	989074	119839	817768	1021125	511352

Table 2. Valid page numbers in a modified 2-way TLB after one read

From the above tables we can see that 6 pages mapped for entry 0 in a traditional 2-way associative TLB so that 4 of them must be rejected. In a modified 2-way TLB only 3 pages contented for entry 0. There are 10 valid pages on the conventional 2-way TLB, while there are 13 valid pages on the modified 2-way set associative TLB after one read.

To measure the TLB miss rate, the SPEC92 benchmark programs are used as workloads. The SPEC92 benchmark consists of floating point programs and integer programs which are various application programs. Seven SPEC92 benchmark programs are used in simulations. One million memory references, generated by a R3000 CPU, are collected at some midpoint for each benchmark program. These traces are downloaded from ftp://ftp.cs.newcastle.edu.au/pub/r3000_traces/ and originally provided by New Mexico State University Trace Database Parallel Architecture Research Laboratory.

TLBs with different entries and with different associativity are simulated for different page sizes. Increasing the page size decreases the number of distinct pages for a particular program. Since, these benchmark programs contain only one million memory references, we experimentally discovered that when the number of entries in the TLB is larger than 128 and the page size is larger than 512 bytes, the number of distinct pages for benchmark programs are very close to the number of entries in the TLB so that the misses are dominated by compulsory misses and the miss rates for a modified TLB and a conventional TLB differ little. Therefore, in order to exploit the advantage of a modified TLB fully, small page sizes are used for the TLB yielding a higher number of entries to increase the number of distinct pages. TLBs with entries from 16 to 512 are simulated,

with the page size from 128 bytes to 1024 bytes. Performances of conventional 2-way, 4-way and 8-way set associative TLBs versus modified 2-way, 4-way and 8-way set associative TLBs are compared in terms of miss rate. The NNR page replacement policy, described in section 6.5, is used throughout the simulations. In all the tables in this paper, the terms new n- way, old n-way stand for modified n-way set associative TLB and conventional n-way set associative TLB respectively.

program	miss % for new 2 – way			miss % for old 2-way			miss % for fully as sociative		
	16-entry	32-entry	64-entry	16-entry	32-entry	64-entry	16-entry	32-entry	64-entry
nasa	0.1	0.0713	0.0508	0.1398	0.0932	0.0601	0.1114	0.0773	0.0546
gcc	5.6968	3.5274	1.888	6.1173	3.9101	2.1448	5.5957	3.6941	1.9234
espresso	1.0981	0.1047	0.0358	3.4035	0.5384	0.1301	0.741	0.0753	0.0259
fpppp	3.1317	1.3598	0.4332	3.4984	1.9758	0.8764	2.7784	1.1104	0.3925
li	3.0045	1.077	0.2824	3.9566	2.1149	0.5715	2.7243	0.814	0.148
doduc	2.3429	0.8752	0.2221	2.6164	1.4322	0.5583	1.9298	0.8417	0.1391
spice2g6	3.2639	1.08	0.5737	4.2482	1.6525	0.8054	2.8804	1.1196	0.6317

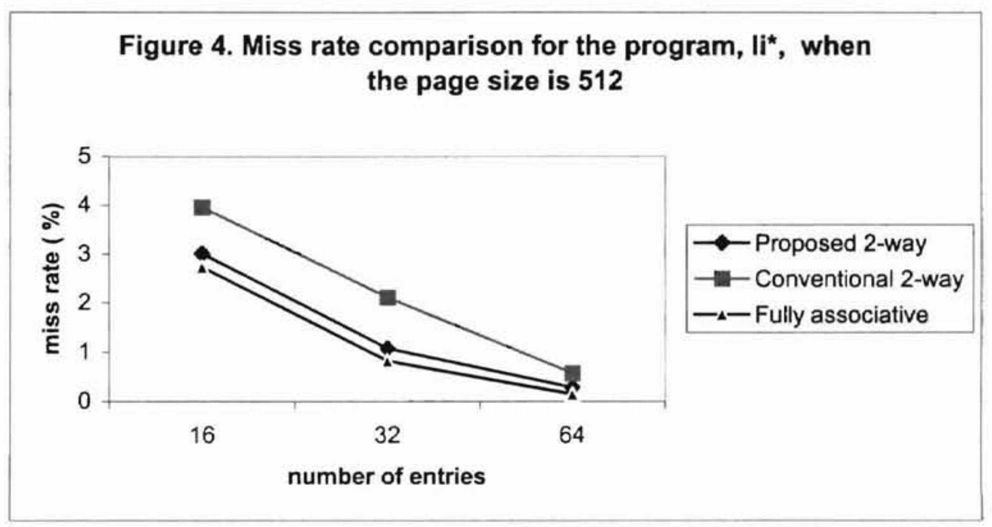
Table 3. Miss rates for a modified, a conventional 2-way set associative TLB and a fully associative TLB when the page size is 512.

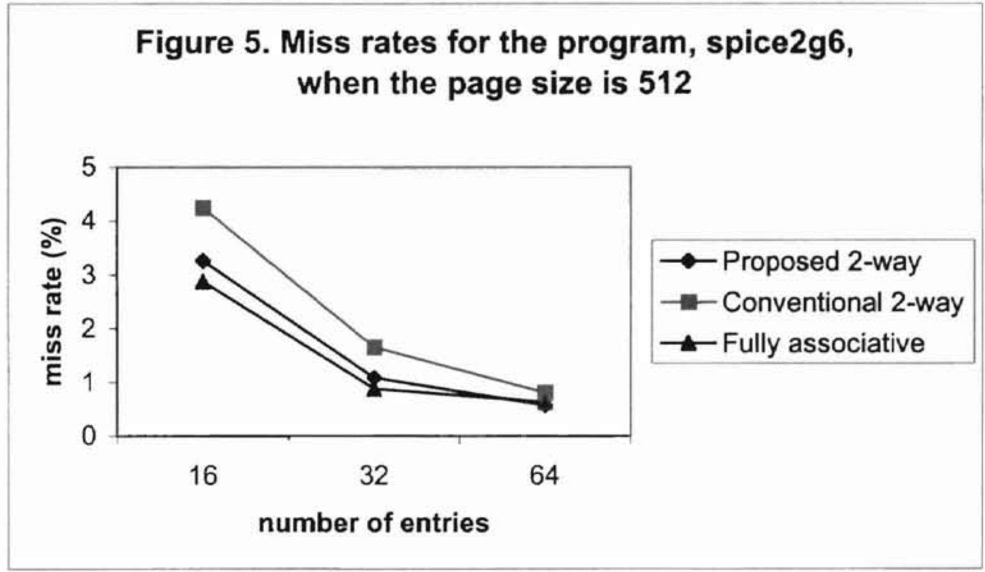
program	miss % for new 2 – way			miss % for old 2-way			miss % for fully as sociative		
	16-entry	32-entry	64-entry	16-entry	32-entry	64-entry	16-entry	32-entry	64-entry
nasa	0.0714	0.0424	0.0248	1.4179	0.6826	0.1309	0.0618	0.0347	0.0251
gcc	4.4353	2.1098	0.7123	4.4295	2.4546	1.3067	4.157	2.2894	0.5579
espresso	0.3564	0.0405	0.0142	0.8385	0.1521	0.0368	0.3446	0.0241	0.012
fpppp	2.0218	0.6736	0.1444	2.471	1.1786	0.638	1.6071	0.4777	0.123
li	2.7804	0.6025	0.17	3.6204	1.5161	1.0958	1.8523	0.3795	0.0407
doduc	1.5437	0.4071	0.106	1.9649	0.8559	0.227	1.2801	0.3305	0.0577
spice2g6	1.5928	0.5311	0.2144	2.6702	0.8281	0.2178	1.4397	0.608	0.1881

Table 4. Miss rates for a modified, a conventional 2-way set associative TLB and a fully associative TLB when the page size is 1024.

Tables 3 and 4 give miss rates for a conventional 2-way, a modified 2-way, and a fully associative TLBs, when the page size is 512 and 1024. As expected and as shown in the Tables 3 and 4, increasing the page size decreases the miss rate.

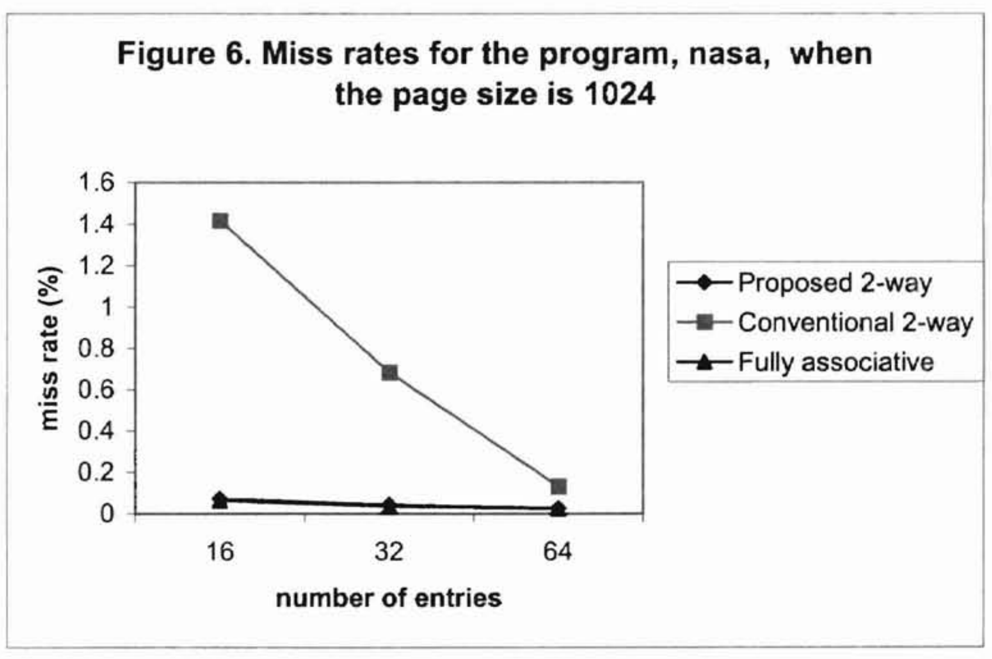
From the results in the tables, we can see a fully associative TLB performs better than a set associative TLB. Also, a modified 2-way TLB exhibits lower miss rate than a conventional 2-way and performs close to a fully associative TLB.





* A benchmark program from SPEC92 test suite.

Figures 4 and 5 show that a modified 2-way associative TLB exhibits a much lower miss rate than a conventional 2-way TLB. Numerical results from tables 3 and 4 give us a 23% – 50% lower miss rate for the modified 2-way set associative TLB than the conventional 2- way set associative TLB. Although, it seems from the graph that, a TLB with fewer entries performs better than one with more entries, but experimental numerical results show that better performances are gained when the TLB has more entries. For example, a modified 2-way set associative TLB exhibited a 24% lower miss rate when the number of entries were 16, 49% lower miss rate when the number of entries were 32, and 51% lower miss rate when the number of entries were 64. This explains the advantage of the idea of scattering the data among entries. Because when each block has more entries, more pages can be scattered within the block and to be put in the TLB so that achieve the goal of decreasing miss rate.



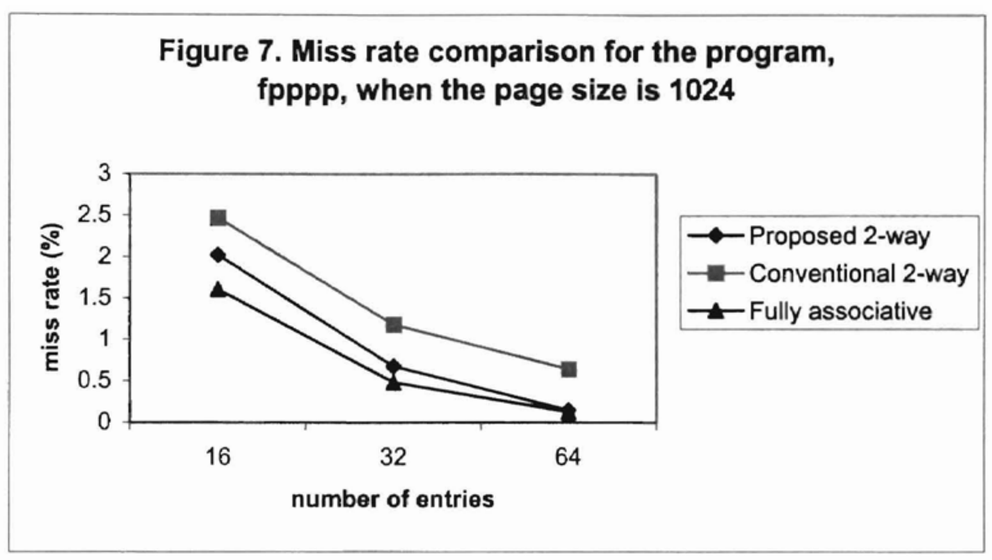


Figure 7 is based on the simulation results for program, fpppp, when the page size is 1024. Interestingly, for the benchmark program, nasa, a modified TLB gave better miss rate with fewer entries rather than more entries while it still exhibited better performance with more entries for the benchmark program, fpppp. Tables 3 and 4 indicate that a modified 2-way associative TLB gives an 18% - 95% lower miss rate than a conventional 2-way set associative TLB. Also, for program, nasa, a modified 2-way set associative TLB performed as well as a fully associative TLB.

Same of the simulation results for 4-way set associative TLBs are shown in Table 5.

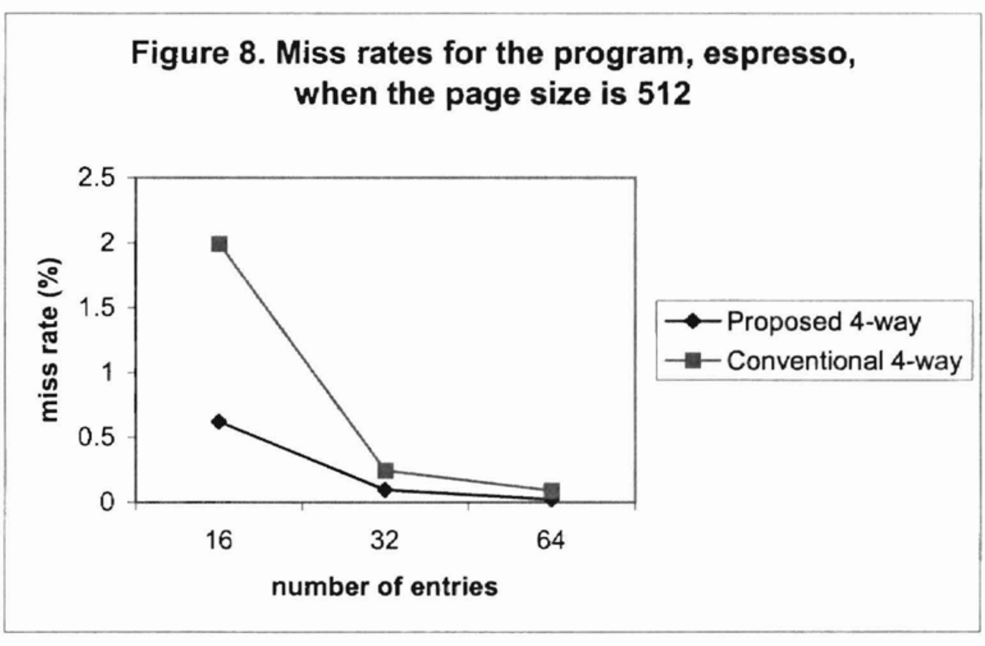
program	miss %	for new	4-way	miss %	for old	4-way
	16 entries	32 entries	64 entries	16 entries	32 entries	64 entries
nasa	0.0958	0.0648	0.0488	0.1238	0.0805	0.0569
gcc	5.6255	3.4761	1.6714	5.5131	3.7556	1.9638
tomcatv	0.2519	0.1901	0.1686	0.5119	0.2679	0.1926
espresso	0.6196	0.0943	0.0227	1.9931	0.2433	0.0904
fpppp	2.7434	0.9769	0.3391	3.0257	1.3295	0.4554
li	2.9764	0.6646	0.1411	3.0514	1.1079	0.3522
doduc	1.9715	0.7519	0.1447	2.0082	0.9436	0.2564
spice2g6	3.1892	0.9027	0.5538	3.2573	1.294	0.682

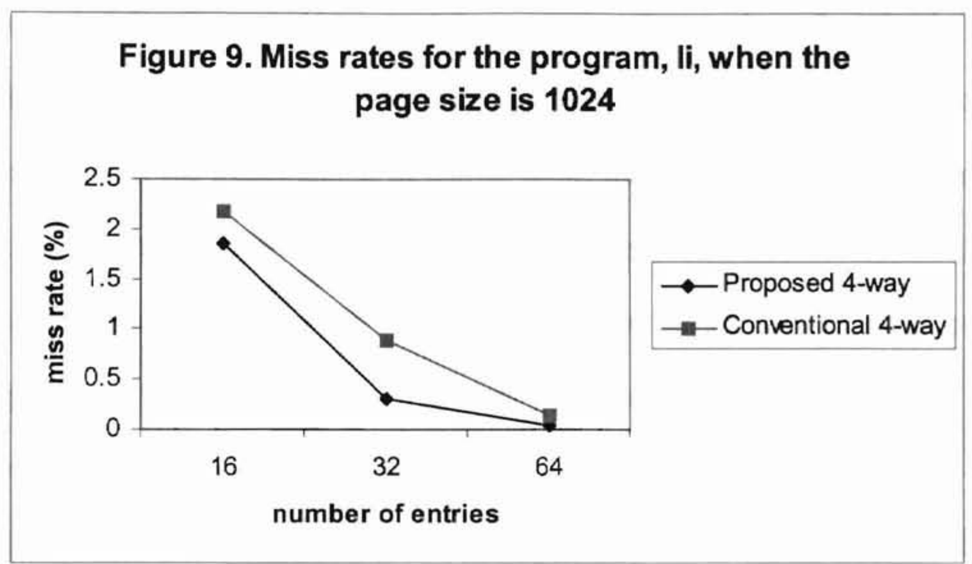
Table 5. Miss rates for a modified, a conventional 4-way set associative TLB when the page size is 512.

program	miss % for new 4 -way			miss % for old 4-way		
	16 entries	32 entries	64 entries	16 entries	32 entries	64 entries
nasa	0.0523	0.0322	0.0247	0.0678	0.0398	0.0248
gcc	4.0553	2.1317	0.5155	4.2476	2.3962	0.7927
tomcatv	0.1515	0.1176	0.1093	0.2221	0.1374	0.1196
espresso	0.3219	0.0206	0.0123	0.4033	0.0982	0.0138
fpppp	1.5857	0.4701	0.0971	1.8137	0.5354	0.14
li	1.8598	0.3049	0.0349	2.1703	0.8951	0.1509
doduc	1.2432	0.3127	0.0494	1.3581	0.4103	0.0915
spice2g6	1.4893	0.487	0.1629	2.5664	0.7295	0.2248

Table 6. Miss rates for a modified, a conventional 4-way set associative TLB when the page size is 1024.

Increasing associativity generally decreases the miss rate [14]. This fact is exhibited in our simulation. It can be seen easily by comparing the results in the table 3 with those from table 5 and results from table 4 with those from table 6. Below are some graphic comparisons of the miss rates of a modified and a conventional 4-way set associative TLB.





On these two cases, a modified 4-way set associative TLB gave a 14% - 77% lower miss rate than a conventional 4-way set associative TLB. It showed better performance in terms of miss rate with more entries in each block. The miss rate reduction percentage for lisp interpreter on a 2-way and a 4-way set associative TLB when the page size is 1024 can be compared. For the 2-way case, the miss rate reduction percentages are 23%, 60% and 84% while the number of entries are16, 32, 64 respectively. For the 4-way case, the miss rate reduction percentages are 14%, 66% and 77%. This tells us that a 2-way set associative TLB yields more improvement than a 4-way set associative TLB. This is not unusual, since the total number of entries in the TLB are equal, increasing associativity decreases the number of entries in each block. Since a 2-way set associative TLB has twice as many entries in each block than a 4-way set associative TLB, more pages can be put in the TLB by scattering pages among the entries in a 2-way set associative TLB. This shows that scattering data among entries is a good idea to reduce the miss rate.

We can compare a conventional and a modified 8-way set associative TLB. As discussed at the beginning of this section, more entries and smaller page sizes are used for 8-way

associativity. Because if we used fewer entries, the number of entries in each block will be even less and we cannot exploit the advantage of scattering data among entries. For example, if the total number of entries is 16, then there are only 2 entries in each block; consequently there will be no room for putting more entries into the block. The reason for using small page sizes is to increase the number of distinct page numbers for the program. If the total number of distinct pages is close to total number of pages, then the miss rate will be dominated by compulsory misses, for which we can do little. It is hard to compare the miss rates for conventional TLB methods and modified TLB methods effectively.

program	miss % for new 8-way			miss % for old 8-way		
	128 entries	256 entries	512 entries	128 entries	256 entries	512 entries
nasa	0.1744	0.1567	0.1528	0.1861	0.1624	0.1546
gcc	2.8927	0.7109	0.2756	3.1453	1.0023	0.2993
espresso	0.076	0.0555	0.0456	0.091	0.0575	0.0464
fpppp	0.8957	0.3025	0.1859	1.1382	0.3695	0.1816
li	0.3435	0.1596	0.0901	0.4149	0.1748	0.0969
doduc	0.7437	0.2566	0.1922	0.848	0.2909	0.1915
spice2g6	1.3974	0.4744	0.0622	1.5266	0.578	0.1063

Table 7. Miss rates for a modified and a conventional 8-way set associative TLB when the page size is 128.

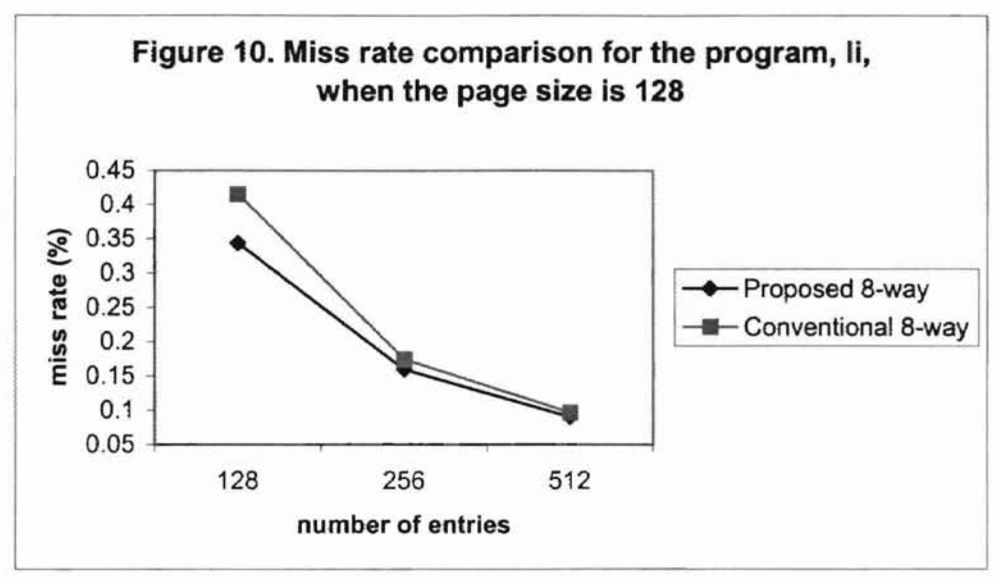
program	miss % for new 8-way			miss % for old 8-way		
	128 entries	256 entries	512 entries	128 entries	256 entries	512 entries
nasa	0.0845	0.0812	0.0797	0.0917	0.0824	0.0804
gcc	0.9592	0.2199	0.1022	1.2784	0.2784	0.109
espresso	0.0352	0.0269	0.0268	0.0415	0.0284	0.0268
fpppp	0.2796	0.1216	0.0753	0.373	0.1367	0.0754
li	0.1197	0.0544	0.0401	0.145	0.065	0.0404
doduc	0.1568	0.1135	0.0812	0.1907	0.1179	0.0821
spice2g6	0.592	0.0593	0.0341	0.6752	0.1159	0.0341

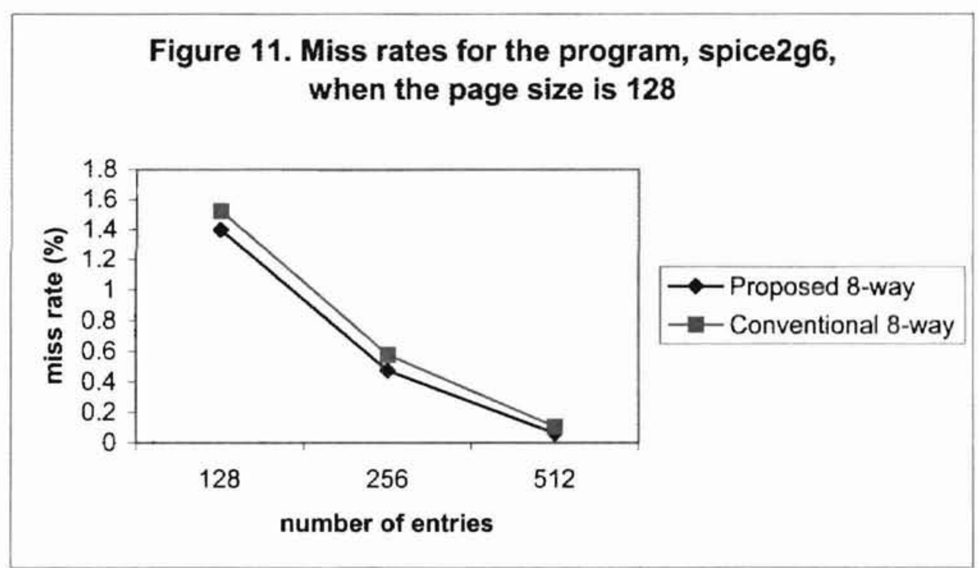
Table 8. Miss rates for a modified and a conventional 8-way set associative TLB when the page size is 256.

program	miss % for new 8-way			miss % for old 8-way		
	128 entries	256 entries	512 entries	128 entries	256 entries	512 entries
nasa	0.0431	0.0425	0.0423	0.0448	0.0427	0.0426
gcc	0.25	0.0677	0.0448	0.3268	0.0906	0.0501
espresso	0.0159	0.0155	0.0155	0.0171	0.0158	0.0158
fpppp	0.0899	0.0469	0.0365	0.1054	0.0482	0.037
li	0.0397	0.0228	0.0226	0.0531	0.024	0.0228
doduc	0.0678	0.0468	0.0384	0.0747	0.0489	0.0381
spice2g6	0.0909	0.0205	0.0201	0.1467	0.0215	0.0205

Table 9. Miss rates for a modified and a conventional 8-way set associative TLB when the page size is 512.

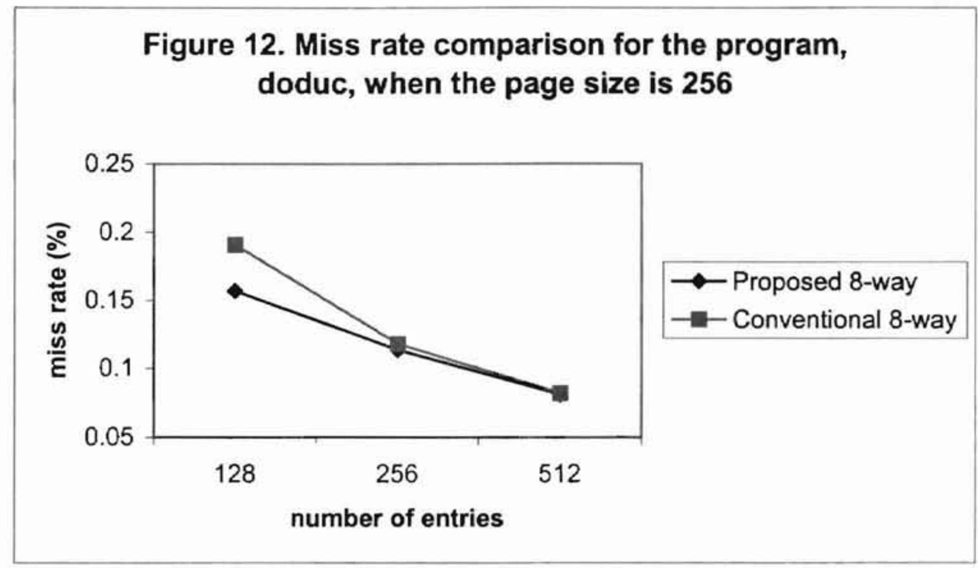
Tables 7, 8 and 9 show miss rates for a modified 8-way set associative TLB and a conventional 8 - way set associative TLB when the page size is 128, 256 and 512 respectively. From the tables one can observe that although a modified TLB exhibits lower miss rates than traditional TLB, as the number of entries increases and/or page size increases the difference between miss rates for a modified and a conventional TLB is not very significant. The author believes this is due to the following factors. Firstly, all benchmark programs have only one million memory references which is fewer than the actual memory references for a program. A typical midsize application program usually has over hundred million instructions. Secondly, it would have either a larger page size and/or more entries in the TLB. Increasing the page size decreases the total number of distinct pages for a program. If the number of entries in the TLB is near the number of distinct pages in the program, then most of the pages are kept in the TLB so that miss rate is dominated by the compulsory misses.

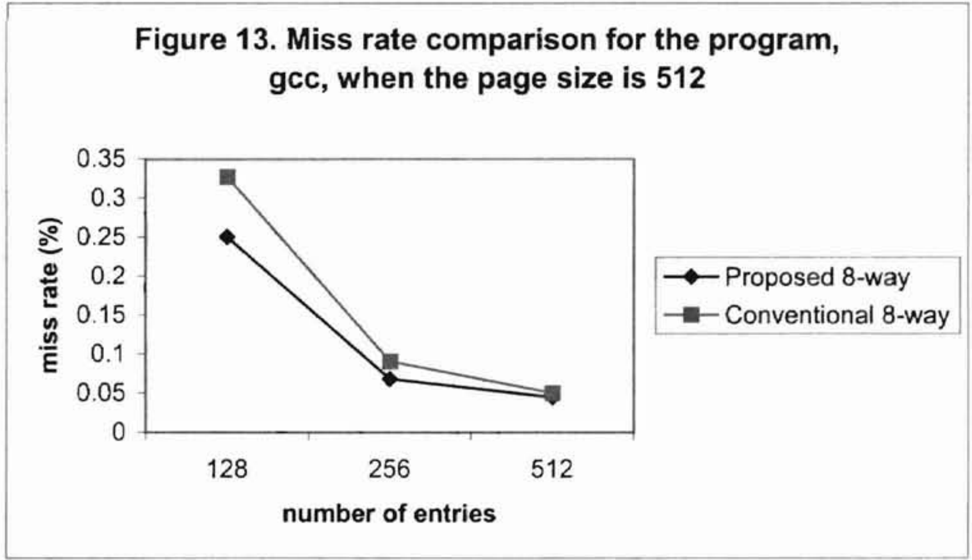




Figures 10 and 11 are based on simulation results of a conventional and a modified 8-way set associative TLB for benchmark programs, lisp interpreter and spice2g6, when the page size is 128. For lisp interpreter, a conventional TLB showed 17.2%, 8.7% and 7% larger miss rate than a modified TLB when the number of entries were 128, 256 and 512, respectively. For the benchmark program, spice2g6, a modified TLB gave 8.5%, 18% and 41% lower miss rate when compared to a conventional TLB. Although, the above

performance improvements in terms of miss rate are not as good as in a 2-way or a 4-way case, they are still improvements that cannot be neglected.

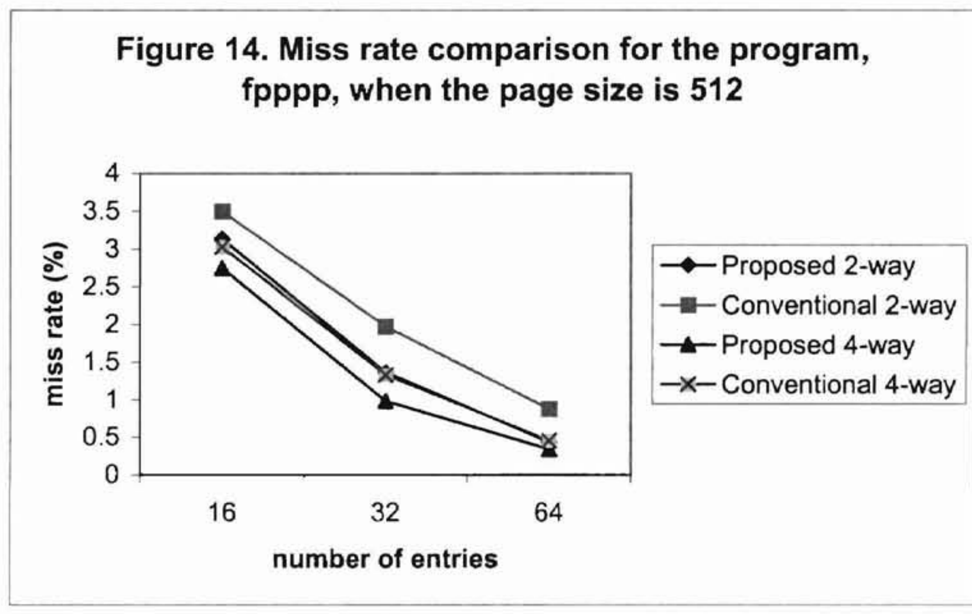


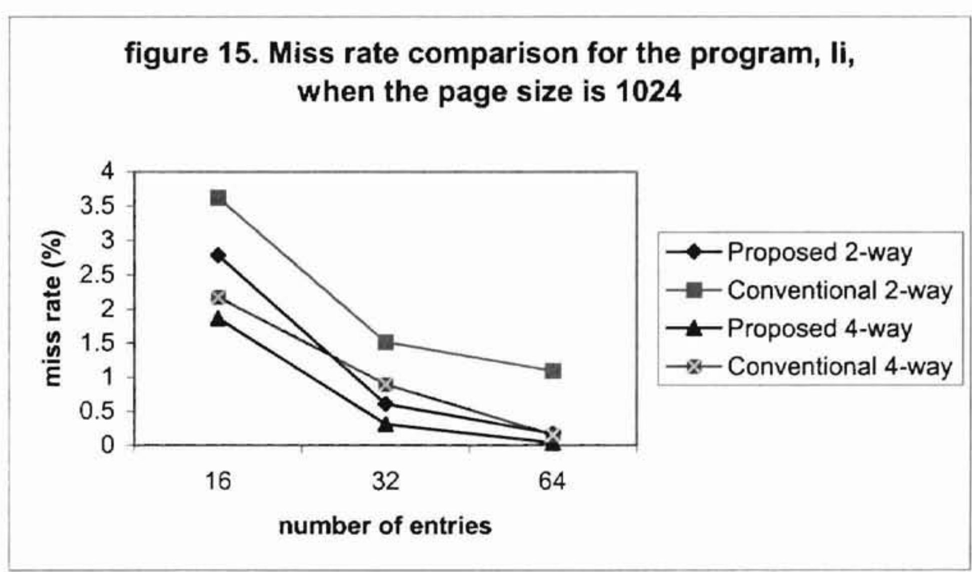


Figures 12 and 13 show the miss rate comparison for the benchmark programs, doduc and gcc, when the page size is 256 and 512. Figures 10, 11, 12 and 13 show that as the number of entries in the TLB increases, the miss rates for a conventional and a modified

TLB get closer. However, when the number of entries was 128, a modified TLB still exhibited 17% - 23 % lower miss rate than a conventional TLB.

We did not compare modified 2-way, 4-way set associative TLBs together with 8-way set associative TLB, because a different number of entries were used for 8-way set associativity.





These two figures show that a modified 2-way set associative TLB performs close to a traditional 4-way set associative TLB. Results from table 3 and table 4 also show that a modified 2-way set associative TLB performs close to a fully associative TLB. Thus, modified 2-way set associative TLB should be preferred to conventional 2-way set associative TLB.

Table 10 and 11 show the miss rates for modified 2-way and conventional 8-way set associative TLBs.

program	miss % for new 2 - way					
	16 entries	32 entries	64 entries	128 entries	256 entries	512 entries
nasa	0.5264	0.31	0.2462	0.1975	0.1676	0.1583
gcc	9.8303	7.4291	5.4942	3.2869	1.3801	0.3988
espresso	2.3826	1.1577	0.3502	0.1044	0.061	0.0501
fpppp	7.4141	4.6783	2.7642	1.1111	0.6217	0.2237
li	6.651	3.4508	1.421	0.5042	0.2301	0.1089
doduc	5.4091	3.8208	2.2575	0.9805	0.3776	0.2168
spice2g6	7.885	4.0074	2.4232	1.5423	0.5956	0.094

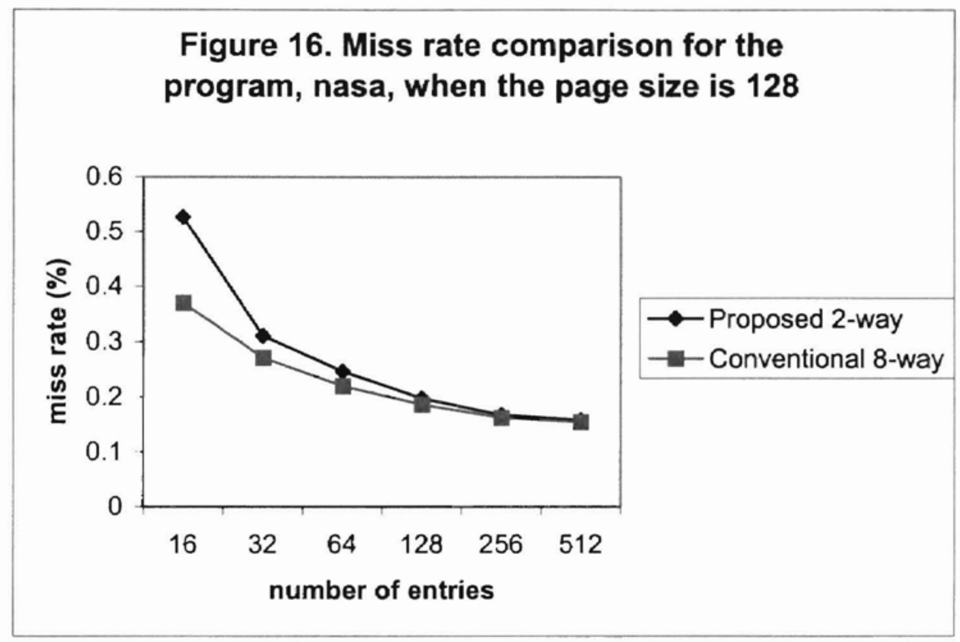
Table 10. Miss rates for a modified 2-way set associative TLB when the page size is 128.

program	miss % for old 8 - way					
	16 entries	32 entries	64 entries	128 entries	256 entries	512 entries
nasa	.3704	.271	.2195	.1861	.1624	.1546
gcc	9.552	7.4075	4.8028	3.1453	1.0023	.2993
espresso	1.8415	.9443	.194	.091	.0575	.0464
fpppp	6.7614	4.3016	2.3254	1.1382	.3695	.1816
li	6.0816	3.0429	1.279	.4149	.1748	.0969
doduc	5.0938	3.6226	2.098	.848	.2909	.1915
spice2g6	7.2773	3.4428	2.1034	1.5266	.578	.1063

Table 11. Miss rates for a conventional 8-way set associative TLB when the page size is 128.

From tables 10 and 11, one can observe that a modified 2-way set associative TLB does not perform better in terms of miss rate than a conventional 8-way set associative TLB.

However, when the number of entries were bigger or equal to 128, the modified 2-way set associative TLB exhibited as well as the conventional 8-way set associative TLB. This supports the idea of scattering data among entries.



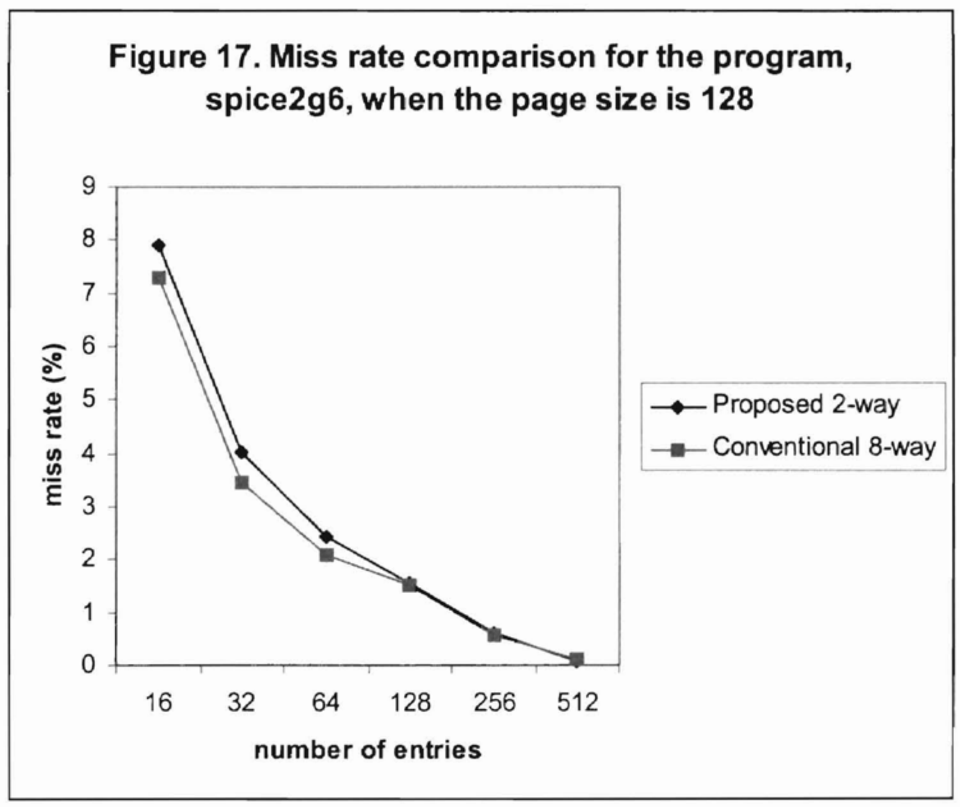


Figure 16 and 17 show that, as the number of entries increases, a modified 2-way set associative TLB performs in terms of miss rate very close to a conventional 8-way set associative TLB. Since increasing the associativity complicates the structure of the TLB, it is favorable to use a modified 2-way set associative TLB instead of a conventional 8-way TLB that has more than 128 entries.

CHAPTER 6

CONCLUSION

In modern computers a TLB can be in the critical path of memory access. Good TLB performance is essential to good overall performance of a machine [12]. Technological and architectural trends have led to increased memory sizes, decreased CPIs, and larger working set programs. Both factors cause more significant performance impact due to TLB misses. Therefore, it is highly desirable to improve TLB hit ratios in future systems.

Designers have used a wide variety of associativity in TLBs. Some systems use fully associative TLBs because a fully associative mapping exhibits a lower miss rate. However, with a fully associative mapping, choosing the entry to replace becomes tricky since implementing a hardware LRU scheme is expensive [6]. Therefore, some systems use set associative mapping. A multiple mapping function scheme is introduced to modify the conventional set associative TLB. It is shown that the modified set associative TLB has a good data scattering property among entries as the number of entries in each block increases. Simulation results also showed that an n-way modified set associative TLB gives lower miss rate than an n-way conventional set associative TLB. Also, performance gain is greater for smaller size TLBs than it is for larger size TLBs. Specifically, a modified 2-way set associative TLB performs in terms of miss rate much better than a conventional 2-way set associative TLB. It performs as well as a fully associative TLB or a 4-way set associative TLB. Simulation results also showed that a 2-way modified set associative TLB performs close to a conventional 8-way set associative

TLB when the number of entries are bigger or equal to 128. Though a modified set associative TLB introduces some extra overhead due to multiple mapping functions, its hardware implementation is almost the same as for a conventional set associative TLB, and since the address translation is not on the critical stage of the pipeline, this overhead is negligible. Therefore a modified *n*-way set associative TLB should be preferred to a conventional *n*-way set associative TLB.

REFERENCES

[1] Channon, D. and D. Koch. Performance Analysis of Re-configurable Partitioned TLBs. Proceedings of the 30th Hawaii International Conference on System Sciences, Volume 5, 1997.

[2] Chen, J. B., A. Borg, and N. P. Jouppi. A Simulation Based Study of TLB Performance. *WRL Research Report* 91/2, Palo Alto, CA, Digital Equipment Western Research Laboratory, May, 1992.

[3] Clark, D. W., P. J. Bannon, and J. B. Keller. Measuring VAX 8800 Performance with a Histogram Hardware Monitor. Proceedings of the 15th Annual International Symposium on Computer Architecture, June 1988.

[4] Clark, D. W., and J. S. Emer. "Performance of the VAX 11/780 Translation Buffer: Simulation and Measurement". ACM *Transactions on Computer Systems* Volume 3, Number 1, 1985.

[5] Denning, P. J.. Working Sets Past and Present. Communications of the ACM, Volume 6, Number 1, 1980.

[6] Hennessy, J. L., and D. A. Patterson. Computer Organization and Design. 2nd ed. San Francisco, CA, Morgan Kauffmann Publisher, Inc., 1999.

[7] Hwang, Kai. Advanced Computer Architecture: Parallelism, Scalability, Programmability. New York, McGraw-Hill, 1993.

[8] Kilburn, T., et al. One-level Storage System. IEEE *Transactions on Electronic Communications*, Volume 11, Number 2, 1962.

[9]. Lee, Jung-Hoon, et al. Dual TLB Structure for Supporting Two Page Sizes. Electronics Letters, Volume 36, Number 8, 2000

[10] Liu, Lishing. Multiple-Page Translation for TLB. Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors, Volume 3, Number 6, 1993.

[11] Przybylski, S. A.. Cache Design: A Performance-Directed Approach. San Mateo, CA, Morgan Kaufmann Publisher, Inc., 1990.

[12] Rosenblum, M., et al. The Impact of Architectural Trends on Operating System Performance. In Proceedings of 15th Symposium on Operating Systems Principles, pages 285-298, December 1995.

[13] Saulsbury, Ashley, et al. Recency-Based TLB Preloading. Proceedings of the 27th International Symposium on Computer Architecture, 2000.

[14] Silberschatz, A., and P. B. Galvin. Operating System Concepts. 5th ed. Reading, MA, Addison Wesley Longman Inc., 1997.

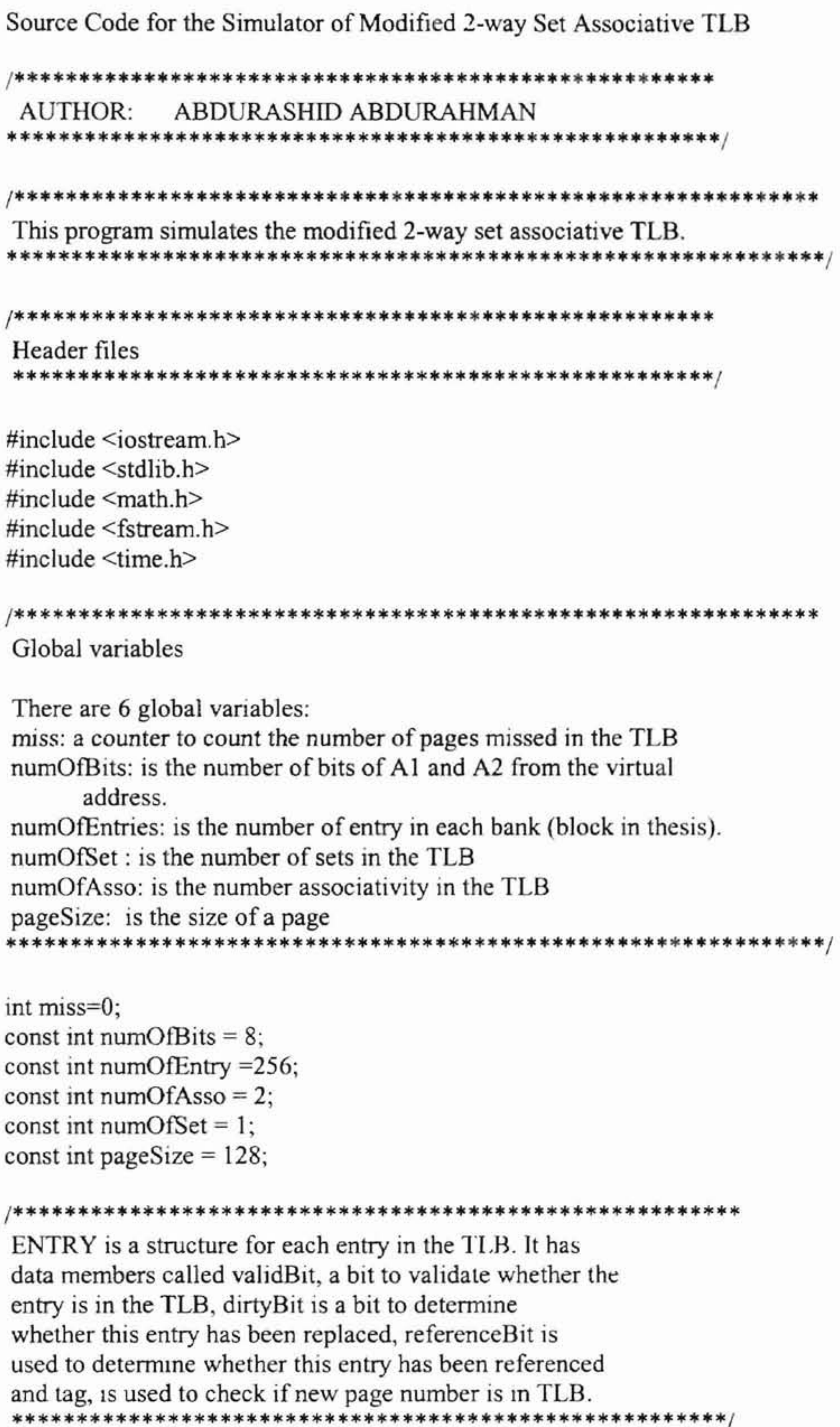
[15] Smith, A. J.. "Cache Memories". ACM *Computer Surveys* Volume 14, Number 3, 1982.

[16] Swanson, Mark, et al. Increasing TLB Reach Using Superpages Backed by Shadow Memory. Proceedings of the 25th Annual International Symposium on Computer Architecture, 1998.

[17] Tanenbaun, A. S.. Modern Operating Systems. Upper Saddle River, NJ, Prentice Hall, Inc., 1992.

[18] Thoreson, S. A., and A. N. Long. Locality: A Memory Hierarchy and Program Restructuring in a Dataflow Environment. *The Journal of Systems and Software* Volume 9, Number 4, May, 1989.

APPENDIX



Source Code for the Simulator of Modified 2-way Set Associative TLB

```
/*****************************************************
 AUTHOR:     ABDURASHID ABDURAHMAN
*****************************************************/

/************************************************************
 This program simulates the modified 2-way set associative TLB.
************************************************************/

/*****************************************************
 Header files
*****************************************************/

#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include <fstream.h>
#include <time.h>

/************************************************************
 Global variables

 There are 6 global variables:
 miss: a counter to count the number of pages missed in the TLB
 numOfBits: is the number of bits of A1 and A2 from the virtual
        address.
 numOfEntries: is the number of entry in each bank (block in thesis).
 numOfSet : is the number of sets in the TLB
 numOfAsso: is the number associativity in the TLB
 pageSize:  is the size of a page
************************************************************/

int miss=0;
const int numOfBits = 8;
const int numOfEntry =256;
const int numOfAsso = 2;
const int numOfSet = 1;
const int pageSize = 128;

/*****************************************************
 ENTRY is a structure for each entry in the TLB. It has
 data members called validBit, a bit to validate whether the
 entry is in the TLB, dirtyBit is a bit to determine
 whether this entry has been replaced, referenceBit is
 used to determine whether this entry has been referenced
 and tag, is used to check if new page number is in TLB.
*****************************************************/
```

```
struct ENTRY
{
        int validBit;
        int dirtyBit;
        int referenceBit;
        int tag;
};

/*************************************************************
 Class BANK has a private data member called Entry. It is an
 array of structure ENTRY. Total number of ENTRY in each bank
 is predefined by numOfEntry.

 Methods in the class BANK:

 BANK(): constructor

 checkPage (int, int): This method checks if the page referenced
 in the first argument is in the TLB or not. Second argument is
 entry number to be checked. If the entry to be checked is
 empty it returns 0. If the page in the entry not equal to the
 page to be checked it returns 1. If the page referenced is a hit
 then it returns 2.

 putPage (int, int): This method puts the page in the first argument
 into the entry in the second argument.

 replacePage (int, int): This method replaces the page in the first
 argument with the page in the entry at the second argument.

 getVictim (int, int, int): This method decide which entry to be
 replaced in case there is a conflict miss.

 countValid (int, int): This method counts the valid page numbers
 inside at some point.

 reset (): This method resets the reference bits in each entry for
 this bank.

*************************************************************/

class BANK
{
private:

        ENTRY Entry[numOfEntry];

public:

        BANK(); //constructor
```

```
        int checkPage (int, int);
        void putPage (int, int);
        void replacePage (int,int);
        int getVictim (int, int);
        int countValid();
        void reset();
};

/***************************************************************
 Class SET has a data member called Bank. It is an array of
 structure BANK. Array size is the number of associativity.

 Methods in SET class:

 checkPage (int, int, int): Calls checkPage method for the bank
 given int the second argument.

 putPage (int, int, int) : Also calls putPage method for the bank
 in the second argument.

 replacePage (int, int, int): Same as above calls replacePage method
 for the bank in the second argument.

 getVictim (int, int, int): Again calls getVictim method for the bank
 in the second argument.

 countValid (): Counts the valid entries in the Set by calling
 countValid() method for each bank in the Set.

 reset(): Resets the reference bit of each entry in this set.

****************************************************************/


class SET
{
private:

        BANK Bank[numOfAsso];

public:

        SET (); //constructor
        int checkPage (int, int, int);
        void putPage (int, int, int);
        void replacePage (int, int, int);
        void getVictim (int, int&, int&);
        int countValid();
        void reset();
};
```

```
/*****************************************************************
 This is the class TLB. It has data member Set which is an array
 of SET. Size of the array is predefined by number of sets.
 Our TLB has number of sets, each set has number of banks determined
 by the associativity of the TLB and each bank has number of entries
 which is predefined.

 TLB class has only four methods:

 TLB() : constructor

 accessPage (int) : This method calculates the entries corresponding
 to the page in the argument. Then calls checkPage methods for the
 corresponding Set and decide whether the referenced page is hit or
 miss. After that I calls related methods in the set.

 countValid() : Calls countValid() method in the set to count the
 valid pages in the TLB.
*****************************************************************/

class TLB
{
private:

        SET Set[numOfSet];

public:

        TLB(); //constructor
        void accessPage (int);
        int countValid();
        void reset();
};


/*****************************************************************
 This function converts an integer to its binary representation
*****************************************************************/

void  intToBin(int num, int binary[])
{
  int i=0;
  int n = numOfBits;
  while(num>0)
  {
    if(num%2==1)
      binary[n-1-i]=1;
    else
      binary[n-1-i]=0;
    num=num/2;
    i++;
```

```
  }//end of while(num>0)

  for(int j=0;j<n-i;j++) //this loop fills with zeros.
     binary[j]=0;
}

/****************************************************************
 This function converts a binary number to an equivalent integer
 ****************************************************************/

int binToInt(int bin[])
{
  int num=0;
  //this loop calculates the integer value of the binary string
  for(int i=0;i<numOfBits;i++)
  {
     num = num+ static_cast<int>(bin[i]*pow(2,numOfBits-i-1));
  }
  return num;
}

/****************************************************************
 This function returns an integer value after bit by bit XORing
 the two arguments. This is actually the first mapping function
 used to calculate the entry for a page.
 ****************************************************************/
int XOR (int num1, int num2)
{
        int bit1[numOfBits]; //bit1 is the bit representation of num1
        int bit2[numOfBits]; //bit2 is the bit representation of num2
        int bit3[numOfBits]; //bit3 is the bit representation of the
                      //result of XORing num1 and num2

        intToBin(num1, bit1); //converts num1 to binary
        intToBin(num2, bit2); //converts num2 to binary

        //this loop performs the XOR operation bit by bit
        for (int i=0; i<numOfBits;i++)
        {
                if(bit1[i]==0 && bit2[i]==0)
                        bit3[i] = 0;
                if(bit1[i]==0 && bit2[i]==1)
                        bit3[i] = 1;
                if(bit1[i]==1 && bit2[i]==0)
                        bit3[i] = 1;
                if(bit1[i]==1 && bit2[i]==1)
                        bit3[i] = 0;
        }

        return binToInt(bit3); //converts result to integer
}
```

```
/*******************************************************************
 This is the second mapping function to calculate entry for a page.
 First it shifts first arguments circularly to the right, then
 XOR it with second argument.
*******************************************************************/

int XOR2(int num1, int num2)
{
  int bit1[numOfBits]; //bit1 is the bit representation of num1
        int temp;

        intToBin(num1, bit1); //converts to binary

        //implementation of circular right shift
        temp = bit1[numOfBits-1]: //store the last bit

        for (int j=numOfBits-1; j>0;j--) //shift
        {
                bit1[j] = bit1[j-1];
        }
        bit1[0] = temp;

        num1 = binToInt(bit1); //convert back to integer

        return XOR(num1, num2); //perform XOR after shift
}

/*******************************************************************
 Following is the constructor and method implementation for BANK
*******************************************************************/

/*******************************************************************
 Constructor for class BANK. It initializes each data member (ENTRY)
 of the instance of BANK.
*******************************************************************/

BANK::BANK()
{
        for (int i=0; i<numOfEntry;i++)
        {
                Entry[i].validBit = 0;
                Entry[i].dirtyBit = 0;
                Entry[i].referenceBit = 0;
                Entry[i].tag = -1; // since zero can be a valid tag address,
                        // -1 is used for tag initialization
        }
}
```

```
/***************************************************************
Method checkPage:
First argument is the page number referenced, second argument is
the entry offset for this page in the bank. Method returns 0 if the
page is a compulsory miss, returns 1 if it is conflict miss, and
returns 2 for a hit.
***************************************************************/

int BANK::checkPage(int page, int entry)
{
        if (Entry[entry].validBit == 0)
                return 0;
        else if (Entry[entry].tag != page)
                return 1;
        else
        {
                Entry[entry].referenceBit=1;
                return 2;
        }

}

/*******************************************
putPage method

set valid bit and tag for the entry.
*******************************************/
void BANK::putPage(int page, int entry)
{
        Entry[entry].validBit = 1;
        Entry[entry].tag = page;
}

/***************************************************************

Method replacePage:

First argument is the page number referenced, second argument is the
entry to be replaced. Replace the victim page with the new referenced
page and set dirty bit.
***************************************************************/

void BANK::replacePage(int page, int entry)
{
        Entry[entry].dirtyBit = 1;
        Entry[entry].tag = page;

}
```

```
/****************************************************************
 Method getVictim finds the victim page to be replaced in the TLB.

 This method has argument called flag. It checks the victim page
 according to the NNR replacement policy described in the thesis.
 First it checks if the entry clear, if not it checks whether it
 has been replaced or not.
****************************************************************/

int BANK::getVictim(int entry, int flag)
{
	if (flag == 1)
	{
		if (Entry[entry].referenceBit == 0)
			return 1;
		else
			return 0;
	}
	else if (flag == 2)
	{
		if (Entry[entry].dirtyBit == 0)
			return 1;
		else
			return 0;
	}
	else
		return 1;
}

/****************************************************************
 countValid method implementation. It counts the valid pages in the
 BANK by checking the valid bit of each entry.
****************************************************************/

int BANK::countValid()
{
	int valid = 0;
	for (int i=0;i<numOfEntry;i++)
	{
		if (Entry[i].validBit == 1)
			valid++;
	}
	return valid;
}

/*****************************************************************
 reset method. It resets the reference bit at some period
*****************************************************************/

void BANK::reset()
{
```

```
        for (int i=0;i<numOfEntry;i++)
        {
                Entry[i].referenceBit = 0;
        }
}
/*****************************************************************
 Following are the methods implementation for SET
 *****************************************************************/

/*****************************************************************
 Constructor for SET
 *****************************************************************/

SET::SET()
{}

/******************************************************************
 Method checkPage:
 First argument is the page number referenced, second argument is
 bank offset, third one is the entry offset for this page in the bank.
 Method calls checkPage method for the BANK given by second argument
 and checks if the page is in this Set. It returns 0 if the page is
 a compulsory miss, returns 1 if it is conflict miss, and returns 2
 if it is a hit.
 ******************************************************************/

int SET::checkPage(int page, int bank, int entry)
{
        return Bank[bank].checkPage(page, entry);
}

/**************************************************
  putPage method calls putPage method for the BANK in the
  second argument to put the page in the first argument
  into the entry in the third argument.
***************************************************/

void SET::putPage(int page, int bank, int entry)
{
        Bank[bank].putPage(page, entry);
}

/******************************************************************
 Method replacePage:

 It calls the replacePage method for the bank given by second argument
 and replaces the referenced page into the entry.

 ******************************************************************/

void SET::replacePage(int page, int bank, int entry)
```

```
{
        Bank[bank].replacePage(page, entry);
}

/*****************************************************************
 getVictim method implementation for class SET.
 This method decides which page to be replaced from TLB once the
 referenced page is a conflict miss.
 First, it calculates the entry number for the referenced page.
 Then it decides which one to be replaced by checking the entries
 according to the NNR replacement policy described in Thesis.
*****************************************************************/

void SET::getVictim(int page, int& bank, int& entry)
{
        int power = static_cast<int>(pow(2, numOfBits));

        int A1 = page % power;
        int A2 = ((page - A1)/power) % power;

        // we have only two entries, because this is simulator for 2-way
        // set associative TLB.
        // For n-way, there are n entries to be checked.

        int entry1 = XOR(A1,A2); //calculate entry1
        int     entry2 = XOR2(A1,A2); //calculate entry2


        int flag = 1;

        while(true)
        {
                if(Bank[0].getVictim(entry1, flag)==1 && Bank[1].getVictim(entry2,
                  flag)==1)
                {
                        bank = rand()%2;
                        if (bank==0)
                                entry = entry1;
                        else
                                entry = entry2;
                        break;
                }
                else if(Bank[0].getVictim(entry1, flag)==0 && Bank[1].getVictim(entry2,
                      flag)==1)
                {
                        bank=1;
                        entry=entry2;
                        break;
                }
                else if(Bank[0].getVictim(entry1, flag)==1 && Bank[1].getVictim(entry2,
                       flag)==0)
```

```
		{
			bank=0;
			entry=entry1;
			break;
		}
		else if(Bank[0].getVictim(entry1, flag)==0 && Bank[1].getVictim(entry2,
			flag)==0)
		{
			flag++;
		}
	}
}


/*****************************************************************
 countValid method implementation. Counts valid pages in the SET
 *****************************************************************/

int SET::countValid()
{
	int valid = 0;
	for (int i=0;i<numOfAsso;i++)
	{
		valid = valid + Bank[i].countValid();
	}
	return valid;
}

/******************************************************************
 reset method. It resets the reference bits of each entry in the Set
 ******************************************************************/

void SET::reset()
{
	for (int i=0;i<numOfAsso;i++)
	{
		Bank[i].reset();
	}

}


/*****************************************************************
 Following is the constructor and method implementation for TLB
 *****************************************************************/

TLB::TLB()
{}
```

```
/*****************************************************************
 Method accessPage: This is the first method to be called to each
 referenced page.

 Argument is the page number referenced.
 This method first calculates the set number, bank number and entry
 number for this page in the TLB then calls the corresponding set
 to check whether the page is in TLB.

 Here it is assumed that page number can be written in the form
 page number = A3*2^(2n)+A2*2^n+A1 where 2^n is the number of entry
 in each bank.

 A1 = page number MOD 2^n
 A2 = [(page number - A1)/2^n] MOD 2^n
 We do not need A3

 set number = page number MOD numOfSet

 bank number is calculated in the following way.

 First we calculate entry1 = A1 XOR A2, entry2 = A1 XOR2 A2.
 Then we check these entries. If the referenced page is a hit, we do
 nothing. If not, and both entries are empty (compulsory miss)
 then we randomly choose one of them. If any one of them is empty then
 the page to be out in that entry. If both of them are taken, then
 victim page should be decided and the referenced page to be put in
 the entry of the victim page.

*****************************************************************/

void TLB::accessPage(int page)
{

        int A1;
        int A2;
        int check1,check2;
        int set;
        int bank;
        int entry,entry1,entry2;

        int power = static_cast<int>(pow(2, numOfBits));
        A1 = page % power;
        A2 = ((page - A1)/power) % power;
        set = page % numOfSet;
        entry1 = XOR(A1,A2);
        entry2 = XOR2(A1,A2);
        check1 = Set[set].checkPage(page, 0, entry1);
        check2 = Set[set].checkPage(page, 1, entry2);
        if(check1 ==0 && check2 ==0)
        {
```

```
			miss++;
			bank=rand()%2;
			if(bank==0)
				Set[set].putPage(page, 0, entry1);
			else
				Set[set].putPage(page, 1, entry2);
		}
		else if(check1==0 && check2==1)
		{
			miss++;
			Set[set].putPage(page,0,entry1);
		}
		else if(check1==1 && check2==0)
		{
			miss++;
			Set[set].putPage(page,1,entry2);
		}
		else if(check1==1 && check2==1)
		{
			miss++;
			Set[set].getVictim(page, bank, entry);
			Set[set].replacePage(page, bank, entry);
		}
}

/*****************************************************************
 countValid method implementation
 *****************************************************************/

int TLB::countValid()
{
		int valid = 0;
		for (int i=0;i<numOfSet;i++)
		{
			valid = valid + Set[i].countValid();
		}
		return valid;
}

/*****************************************************************
 reset method. It resets the reference bit at some period
 *****************************************************************/

void TLB::reset()
{
		for (int i=0;i<numOfSet;i++)
		{
			Set[i].reset();
		}

}
```

```
/*******************************************************************
 Main function
*******************************************************************/

void main()
{

	ifstream INFILE;
	int access=0;
	unsigned long random, page;
	char line[80];
	char* stop="\n";


	TLB myTLB;
	INFILE.open ("spice2g6"); // open the input file

	while(!INFILE.eof())
	{
		INFILE.getline(line,80);
		random=strtoul(line,&stop,10);
		page=random/pageSize;
		myTLB.accessPage(page);
		access++;
	}
	INFILE.close ();
	cout<<"Miss = "<<miss<<endl;

}
```

ə

VITA

Abdurashid Abdurahman

Candidate for the Degree of

Master of Science

Thesis: MODIFIED SET ASSOCIATIVE TLB

Major Field: Computer Science

Biographical:

Personal Data: Born in Atush, Uyghuristan, eldest son of Abdurahman Aji and Reyimgul Mamut.

Education: Received Bachelor of Science degree in Computational Mathematics from Xinjiang University, Urumqi, Xinjiang, China in July 1991. Received Master of Science degree in Applied Mathematics from Oklahoma State University in May 2001. Completed the requirements for the Master of Science degree in Computer Science at Oklahoma State University in August 2001.

Experience: Employed as Patent Agent & Patent Searcher by Science & Technology Commission of Xinjiang Uyghur Autonomous Region from Aug. 1991 to Aug. 1997.
Employed as Teaching Associate & Lecturer by Mathematics Department of Oklahoma State University from Aug. 1997 to Dec. 2000.
Employed by School of Mechanical & Aerospace Engineering and Department of Microbiology as Research Assistant from May 2000 to July 2001.