國 立 交 通 大 學

資訊工程系

碩 士 論 文

降低行程環境切換導致效能損失之轉換搜尋緩衝器設計

Translation Look-aside Buffer with Low Context Switch Penalty

研 究 生：張繼文

指導教授：陳昌居　教授

中 華 民 國 九 十 四 年 六 月

降低行程環境切換導致效能損失之轉換搜尋緩衝器設計

Translation Look-aside Buffer with Low Context Switch Penalty

研 究 生：張繼文　　Student：Chi-Wen Chang

指導教授：陳昌居　　Advisor：Chang-Jiu Chen

國 立 交 通 大 學

資 訊 工 程 學 系

碩 士 論 文

A Thesis

Submitted to Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Chiao Tung University

in

partial Fulfillment of the Requirements

for the Degree of Master

in

Computer Science and Information Engineering

June 2004

Hsinchu, Taiwan, Republic of China

中華民國九十四年六月

降低行程環境切換導致效能損失之
轉換搜尋緩衝器設計

研究生：張繼文　　　　　　　　　　　　　　　指導教授：陳昌居

國立交通大學 資訊工程學研究所

摘要

一般人所熟悉的轉換搜尋緩衝器是用來將虛擬記憶體轉換成實體記憶體的一種機制，在整個電腦系統運作中扮演極重要的角色。如果有任何的失誤發生時，都會造成處理器的效能嚴重的下降。為了減低任何失誤的可能性發生，有不少的研究方法和理論被提出。有些理論是利用改進轉換搜尋緩衝器的關聯性或增加其容量大小來降低其因衝突產生的失誤和容量不足而導致的失誤，有些研究者則提出使用超級分頁的概念來涵蓋更多的記憶體空間。這些眾多的方法當中，特別是超級分頁對於大部分的應用程式能夠有效率的降低失誤的可能性。可惜的是對於行程切換導致轉換搜尋緩衝器效能降低方面的研究卻是少之又少。為了支援近代的作業系統當中都有多重程式操作的特色，作業系統必須要有行程環境切換機制以方便將正在處理的行程切換到下一個行程，此時須清除目前轉換搜尋緩衝器所暫存的位址轉換資訊。而清除轉換搜尋緩衝器資訊的這樣一個行為，將會造成處理器效能嚴重的下降，特別是對於近代高效能的處理器。這篇論文提出了一種新穎且容易實作的轉換搜尋緩衝器架構來降低行程環境切換機制所帶來的損失同時我們更整合多重分頁的機制來降低失誤的機會。我們修改 SimpleScalar 3.0d 模擬器及使用 SPEC2000 作為我們的模擬平台環境，並比較其它轉換搜尋緩衝器的架構，模擬結果顯示出所提出的轉換搜尋緩衝器架構在傳統的 4KB 分頁大小底下可以比傳統轉換搜尋緩衝器的失誤率小上 1.3 倍，在此可見所提出之架構非常適合用於多重程式環境底下。

Translation Look-aside Buffer with Low Context Switch Penalty

Student : Ji-Wen Zhang Advisor : Dr. Chang-Jiu Chen

Department of Computer Science and Information Engineering
National Chiao-Tung University

Abstract

It is widely known that the Translation Look-aside Buffer (TLB) plays an important role in the address translation mechanism from virtual addresses to physical addresses. If any miss occur, the performance of the processor will seriously degrade. There are many methods for improving TLB performance, such as increasing the associativity, the number of entries, or page sizes, and using superpages to cover more memory spaces. These methodologies, especially superpage, can effectively reduce lots of misses for most applications. However, very few designs really focused on the context switching issue. In order to support the multiprogramming characteristics in all modern OS, the context switching mechanism is needed and it will cause all TLB entries be flushed and will impact on the performance very seriously, especially on today's high performance processors. This thesis presents a novel and easy implemented TLB architecture to reduce the misses in context switching with complete-subblock mechanism. All simulations were done with modified SimpleScalar 3.0d tool suite and SPEC2000 benchmarks. The thesis also compares several designs, including the conventional TLB, the complete-subblock TLB, and the promotion TLB. The simulations show that the new design can achieve about 1.3 times of relative improvement of miss rate in average with 4KB page size and reveal that our methodology can be very useful for multiprogramming environment.

Acknowledgment

I wish to express my gratitude to my prime professor, Dr. Chang-Jiu Chen, and Wei-Min Cheng for their helpful suggestions, detailed comments, and constant support. I am also indebted to a number of my laboratory colleagues at National Chiao-Tung University, especially to Nian-Zhi Huang, Yuan-Teng Chang, Ming-Feng Shen and Shi-Wei Chen, for their encouragement and help. I am most grateful to my mother and father for their economic support.

Contents

摘要 ..... **I**

ABSTRACT ..... **II**

ACKNOWLEDGMENT ..... **III**

CONTENTS ..... **IV**

LIST OF FIGURES ..... **VI**

LIST OF TABLES ..... **VIII**

CHAPTER 1 INTRODUCTION **1**

1.1 PROBLEM DEFINITION 2

1.2 ROADMAP 3

CHAPTER 2 RELATED WORKS **4**

2.1 TLB MECHANISMS 4

2.1.1 Conventional TLB Structure *4*

2.1.2 Several TLB implementations *6*

2.2 ENHANCEMENTS FOR THE TLB 6

2.2.1 Reducing TLB Access Time *6*

2.2.2 Reducing TLB Miss Rate *7*

2.2.3 Reducing TLB Hardware Cost *10*

2.2.4 Low Power TLB *11*

2.3 THE CONTEXT SWITCHING PENALTY IN TLB 13

2.4 RELATIONSHIPS BETWEEN THE MISS RATES, PAGE SIZES AND TLB SIZES 20

CHAPTER 3 PROPOSED MECHANISMS **27**

3.1 THE ORIGINAL TLB STRUCTURE WITH LOW CONTEXT SWITCH PENALTY 27

3.1.1 Structure *27*

3.1.2 OS Support and Implementation *29*

3.1.3 Expected Performance *29*

3.2 THE NEW TLB STRUCTURE WITH LOW CONTEXT SWITCH PENALTY 33

3.2.1 Overview *36*

3.2.2 Proposed TLB Structure *37*

3.2.3 Implementation of the novel TLB *39*

3.2.4 Mechanisms of the novel TLB *39*

CHAPTER 4 SIMULATION RESULTS **42**

4.1 INTRODUCTION TO THE SIMPLESCALAR TOOL SET 42
4.1.1 Overview *42*
4.1.2 Instruction Set Architecture *44*
4.2 EXPERIMENTAL METHODOLOGY 45
4.2.1 TLB Implementation in SimpleScalar Simulator *46*
4.2.2 Modifications to implement our novel TLB structure *47*
4.2.3 Benchmarks *47*
4.3 SIMULATION RESULTS 49
CHAPTER 5 CONCLUSIONS 56
APPENDIX A: DETAILED RELATIVE IMPROVEMENT FIGURES 58
REFERENCE 71

E S A
1896

List of Figures

FIGURE 1: STRUCTURE OF A SINGLE-PAGE-SIZE TLB BLOCK 5

FIGURE 2: STRUCTURE OF CONVENTIONAL TLB. 5

FIGURE 3: SUPERPAGE TLB ENTRY 8

FIGURE 4: SCHEMATIC OF HARDWARE FOR PREFETCHING.. 10

FIGURE 5: COMPLETE-SUBBLOCK TLB BLOCK (SUBBLOCK FACTOR N). 10

FIGURE 6: DUAL TLB STRUCTURE 11

FIGURE 7: THE RELATIONSHIP BETWEEN THE TLB SIZE AND THE MISS RATE WITH CONTEXT SWITCHING DIVIDED BY THE MISS RATE WITHOUT CONTEXT SWITCHING. 17

FIGURE 8: THE RELATIONSHIP BETWEEN THE PAGE SIZE, TLB SIZE AND THE MISS RATE WITH CONTEXT SWITCHING DIVIDED BY THE MISS RATE WITHOUT CONTEXT SWITCHING. 19

FIGURE 9: THE RELATIONSHIP BETWEEN THE TLB MISS RATE AND TLB SIZES WITH TRADITIONAL 4KB PAGE SIZES. 23

FIGURE 10: THE RELATIONSHIP BETWEEN THE TLB MISS RATE AND PAGE SIZE.... 26

FIGURE 11: A LOW CONTEXT SWITCHING MISS RATE TLB ARCHITECTURE. 28

FIGURE 12: DTLB/ITLB MISS RATE COMPARISON WITH 1MB PAGE SIZE. 32

FIGURE 13: TLB MISS RATE COMPARISON WITH 4KB PAGE SIZE. 35

FIGURE 14: NEW PROPOSED TLB ARCHITECTURE WITH LOW CONTEXT SWITCHING PENALTY. 37

FIGURE 15: SIMULATOR STRUCTURE. 43

FIGURE 16: SIMPLESCALAR ARCHITECTURE INSTRUCTION FORMATS. 45

FIGURE 17: THE COMPARISON OF RELATIVE IMPROVEMENT WITH DIFFERENT PAGE SIZES.. 53

FIGURE 18: THE MEAN OF THE AVERAGE RELATIVE IMPROVEMENT FOR THE TWELVE SPEC2000 BENCHMARKS. 54

FIGURE 19: THE RELATIVE IMPROVEMENT OF MISS RATE IN *GZIP*. 59

FIGURE 20: THE RELATIVE IMPROVEMENT OF MISS RATE IN *VPR*............................... 60

FIGURE 21: THE RELATIVE IMPROVEMENT OF MISS RATE IN *GCC*. 61

FIGURE 22: THE RELATIVE IMPROVEMENT OF MISS RATE IN *CRAFTY*.........................62

FIGURE 23: THE RELATIVE IMPROVEMENT OF MISS RATE IN *VORTEX*. 63

FIGURE 24: THE RELATIVE IMPROVEMENT OF MISS RATE IN *LUCAS*...........................64

FIGURE 25: THE RELATIVE IMPROVEMENT OF MISS RATE IN *TWOLF*. 65

FIGURE 26: THE RELATIVE IMPROVEMENT OF MISS RATE IN *SWIM*.............................66

FIGURE 27: THE RELATIVE IMPROVEMENT OF MISS RATE IN *PERLBMK*. 67

FIGURE 28: THE RELATIVE IMPROVEMENT OF MISS RATE IN *APPLU*...........................68

FIGURE 29: THE RELATIVE IMPROVEMENT OF MISS RATE IN *EQUAKE*.......................69

FIGURE 30: THE RELATIVE IMPROVEMENT OF MISS RATE IN *MGRID*..........................70

List of Tables

TABLE 1: SUMMARY OF THE SIMULATED SPEC2000 BENCHMARKS ALONG WITH THE INPUT SETS. 48

TABLE 2: SPEC 2000 BENCHMARKS DESCRIPTION.......... 48

TABLE 3: MEMORY SPACE COVERAGE COMPARISON OF FOUR TLBS.......... 50

Chapter 1 Introduction

To support large memory requirements for modern applications, all new advanced general-purpose processors will support the virtual memory. The virtual memory is one of the few interfaces through which the architecture and operating system interact directly and is developed to automate the movement of program code and data between main memory and secondary storage to give the appearance of a single large memory system.

In order to support virtual memory, the address translation mechanism is needed. It is well known that all the address translations are stored in main memory and maintained by the operating system; to reduce the cost of address translation, the translation look-aside buffers (TLBs) [10] are implemented inside the processor. If there's any TLB miss occurring, at least two or three memory accesses are needed to fetch the translation from main memory by the memory management unit (MMU).

A case study on TLB miss handling [16]: It has been shown to constitute as much as 40% of execution time and up to 90% of a kernel's computation. Studies with specific applications have also shown that the TLB miss rate can account for over 10% of execution time even with an optimistic 30-50 cycles miss overhead.

With the VLSI technology improving rapidly, the new microprocessors become much faster than ever before and it causes the gap between memories and the processor core larger and larger. We can easily find that the TLB is in the critical path of memory accesses. It's an important issue to reduce the miss rate of TLB [14].

1.1 Problem Definition

To enhance the TLB performance, several studies have been made in this field. However, little attention has been given to the context switching problem under multiprogramming environment. The context switches are wiping out locality from one application to another and cause the flush operations for whole of the TLB entries. It affects the TLB performance very seriously.

In [6], we propose a novel and easy implement TLB structure to reduce the miss rate caused by the context switching. We divide the 256 entries into 32 banks storing translation information of different tasks to avoid flushing all TLB when context switching occurs. The structure is so easy to implement and reducing miss rate effectively. However, it needs large page size as base page in our previous structure, for example, 512KB, 1MB or larger page size. Large page sizes will waste memory space seriously by increasing internal fragmentation, although it can provide the advantage of increasing the overall coverage of memory mapping.

To overcome and improve the limitation in our original structure, our research is intended as a study of how to reduce the miss rate under context switching but using smaller base page size, such as 4KB, 8KB, or 16KB, to implement our new novel TLB architecture. In the study of banked-promotion TLB structure proposed by Lee et al. [17] [18], they promoted four consecutive 4KB pages from one banked-TLB into a 16KB page stored in another banked-TLB dynamically via simple hardware control without any O/S support. We improve and develop this idea a little further to design our novel TLB structure. We combine both the features of the dual TLB and our original TLB with many TLB banks in our new structure. Our TLB architecture not only supports two different page sizes but also keeps TLB information when context

switching occurs. With the new design, we can obtain the advantages of low power consumption by decreasing the amount of fully associative TLB entries to be accessed at one time and less internal fragment problem compared with our original TLB design.

The simulations were be done by the modified SimpleScalar version 3.0d tool suite with SPEC 2000 benchmark. We modified the original SimpleScalar version 3.0d tool suite to accommodate our requirements.

1.2 Roadmap

The rest of the thesis is organized as follows. In Chapter 2, we begin with reviewing several hardware enhancements for TLB. Then we show that the context switching will be the performance bottleneck in TLB and discuss the relationships between the miss rates, page sizes and TLB sizes. In Chapter 3, we will first review our recently TLB architecture with low context switching penalty. Then, we will develop our new novel TLB architecture to reduce miss rate in context switching. The expected performance is demonstrated in Chapter 4. Finally, in Chapter 5 we summarize the conclusions and describe the possible future works.

Chapter 2 Related Works

In chapter 1 we discussed that virtual to physical address translation is one of the most critical operations in computer systems since this is invoked on every instruction fetch and data reference. To speed up the address translation, computer systems that use page based virtual memory provide a cache of recent translations called the Translation Look-aside Buffer (TLB). With the instruction level parallelism, clock frequencies, and the working sets increasing, the amount of research about enhancement for TLB increases. In Section 2.1 we will begin with reviewing the conventional TLB structure. Then we classify all the represented method according to their research purposes, and give a survey of these mechanisms. In Section 2.2, we will study the context switching penalty in TLB. In Section 2.3, we will discuss the relationships between the miss rates, page sizes and TLB sizes.

2.1 TLB mechanisms

2.1.1 Conventional TLB Structure

The address translation acceleration mechanism using translation look-aside buffer (TLB) is based on the principle of temporal locality. A TLB can be considered to be a hardware cache used to contain recently used virtual-to-real address translations. If the TLB has a matching translation — a TLB hit, it outputs the physical address and memory access attributes. If the TLB has no matching translation — a TLB miss, special hardware or software fetches the missing translation and loads it into TLB.

A TLB stores translation in TLB blocks, each containing a tag and a data part.

The tag contains the virtual page number (VPN) of the translation and a valid bit (V). The data part stores the corresponding physical page number (PPN) bits and page attributes (ATTR), e.g., protection, cache-ability, referenced/ modified bits, as shown in Figure 1.

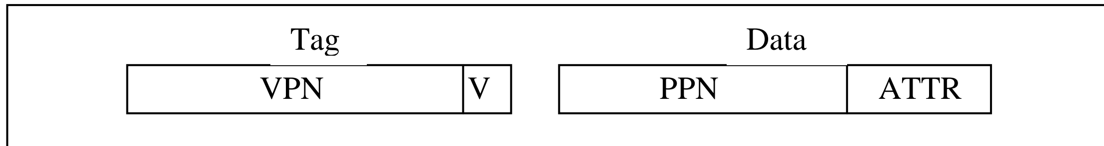


Figure 1: Structure of a single-page-size TLB block

Many such TLB blocks can be combined in either fully-associative or set-associative. In either case, a tag array stores all the tags and includes comparators to compare them with the input VA. A random-access-memory (RAM) stores the data parts of the blocks. During a TLB lookup, the input VA is split into two parts that are the VPN and offset. The offset field, without any translation, appends to the PPN output from the TLB. The TLB compares the VPN stored in the tag with the input VPN. Only TLB blocks that contain a valid translation participate in the comparison. The result of tag comparison outputs the correct PPN and attributes from the RAM. If no block has matching tag, the TLB generate a TLB miss signal, as shown in Figure 2.

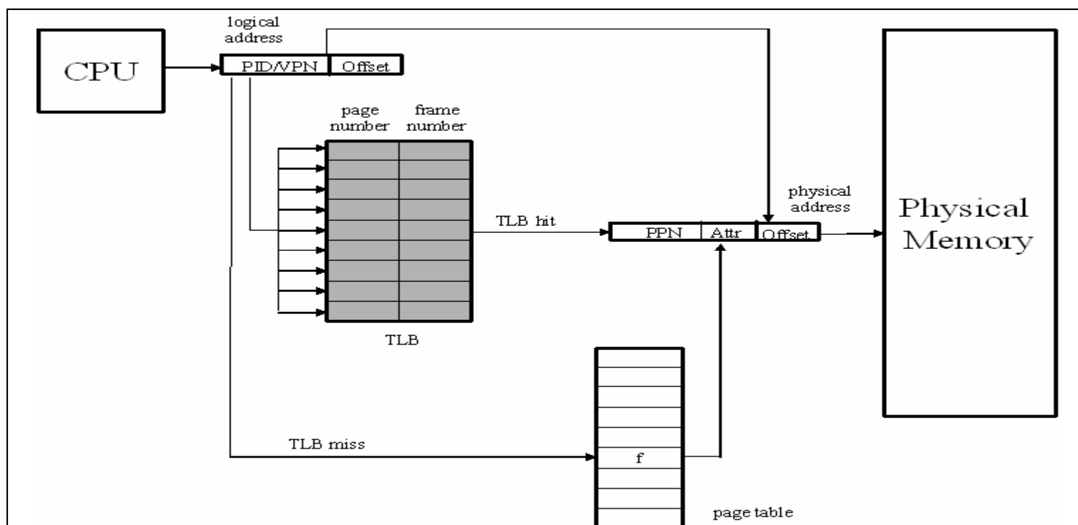


Figure 2: Structure of conventional TLB.

2.1.2 Several TLB implementations

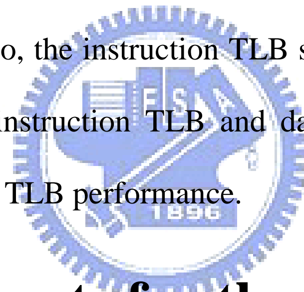
The TLB is a cache used to speeding up access to entries in the page tables, where complete information on virtual memory to physical memory mapping are maintained. We consider two general possibilities for the TLB implementation:

- **A single TLB shared between instruction and data caches.**

  To reduce the contention miss, we can implement dual-ported TLB. This introduces complex circuitry, doubling the size of the TLB without increasing its capacity.

- **Independent TLBs for instruction and data caches.**

  In general, the instruction reference streams exhibit greater locality than data reference streams. So, the instruction TLB should be mad smaller than data TLB. Furthermore, the instruction TLB and data TLB could be implemented independently to get best TLB performance.



2.2 Enhancements for the TLB

2.2.1 Reducing TLB Access Time

- **Multi-level TLB**

  Cache has a property that the smaller hardware is faster. Many processors, such as the Itanium IA-64 [11] (32-entry L1, 96-entry L2), AMD Athlon (32-entry L1, 256-entry L2) etc. provide multi-level TLB structures, instead of a single large TLB. The larger L2 TLB will be accessed only after the smaller L1 TLB miss occurs. With a smaller first level TLB, the average TLB access time can become much less if good hit rates can be obtained in L1 TLB. The performance is conducted by others [5].

Assume that the access time for a single monolithic TLB is **a**. Let the access time for the first and second level TLBs in the hierarchical alternative be a_1 and a_2 respectively. Let us denote the miss fraction of the monolithic TLB, the first and second level of the hierarchical TLB to be m, m_1, and m_2 respectively. Also, let the cost of fetching a translation that is not in the TLB be denoted by ***C.***

Then, the cost of translating an address in the monolithic structure (C_m) is given by

$$C_m = a + m \times C \qquad (2.1)$$

The cost of translating an address in the 2-level TLB (C_s) is given by

$$C_s = a_1 + m_1 \times (a_2 + m_2 \times C) \qquad (2.2)$$

The 2-level TLB is a better alternative when

$$a_1 + m_1 \times (a_2 + m_2 \times C) \prec a + m \times C \qquad (2.3)$$

2.2.2 Reducing TLB Miss Rate

The classical approach to improve TLB performance is to reduce the miss rates, and we present several techniques here to accomplish this goal.

- **Making the TLB hold more entries**

This is a conventional and easy method to improve TLB performance. Processors, such as Intel Pentium !!! Processor [11] use 512 entries 4K page fully-associative or set-associative TLB to reduce the miss rate. But the side effect is that

- Longer memory reference latency can be occurred.
- Power consumption might be larger.

- **Using large page size**

This is a method with less hardware support to improve the overall coverage of memory mapping and to reduce the miss rate effectively; however, the disadvantages is that

  - It wastes memory space seriously because of increasing internal fragment.

- **Using Superpage to improve TLB coverage**

Applications with larger working sets can incur many TLB misses and suffer from TLB penalty. To alleviate the problem of wasting memory coverage without increasing the number of TLB entries or page size, most modern general-purpose CPUs, such as the new Intel Processors from Pentium Pro begin to provide larger page with sizes of 2MB and 4MB [13]. TLBs that support superpage use a single entry for translating a set of consecutive virtual pages as long as these pages are located physically contiguous.

Figure 3 shows the format of a superpage TLB entry. The MASK field prevents certain tag bits from participating in tag comparison for superpage mappings and the SZ attribute controls a multiplexer during physical address generation.

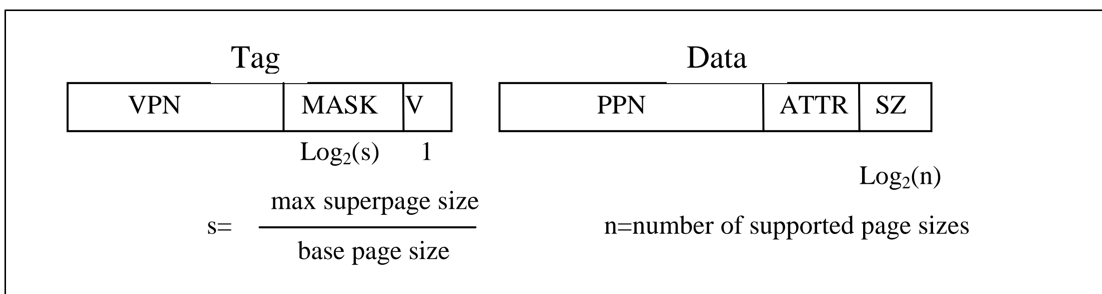


Figure 3: Superpage TLB entry

The restrictions for superpage are:

- The superpage must be mapped only to contiguous and aligned physical pages.
- Using superpage requires large operating system support and causes significant overhead. For example, it increases the amount of I/O, page utilization overhead, and page fault penalty.

- **Prefetching TLB Mechanisms**

Although there are many literatures on prefetching techniques for memory hierarchy, it is only recently [15] [19] that the issue of prefetching TLB entries to hide all or some of the miss costs has started drawing interest. The reason is that the TLB is more important than other levels of memory hierarchy and we fear of slowing down the critical path of TLB accesses due to memory traffic.

Generally speaking, the prefteching mechanisms can be viewed in two classes: Arbitrary Stride prefetching (ASP) and Stride prefetching (SP) capture the strided reference pattern, and Markov prefetching (MP), Recency prefetching (RP) and Distance prefetching (DP) exploit history information about relative recent usage of pages to predict future TLB misses. All of these techniques bring the prefetched entry into a *prefetch buffer* that is concurrently looked up with the TLB; the entry is moved to the TLB entry only after an actual reference to that entry by the application. In the Figure 4 we show the schematic of generic hardware for prefetching in all the considered mechanisms.

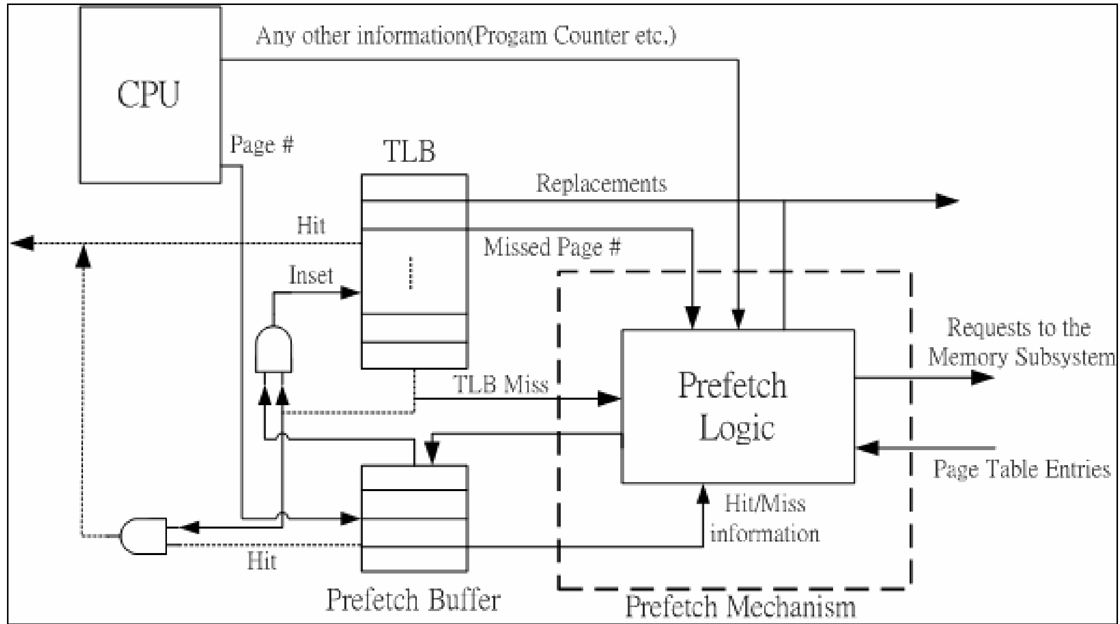


Figure 4: schematic of hardware for prefetching.

2.2.3 Reducing TLB Hardware Cost

- **Complete-Subblock TLB**

Complete-subblock TLB structure is a TLB that have the same TLB reach advantages of medium sized superpages and exploit spatial locality to improve TLB performance without any operating system support [25] [26]. The main idea of complete-subblock is to allow a single TLB block to map multiple base pages to increase the coveraged memory space, as shown in Figure 5.

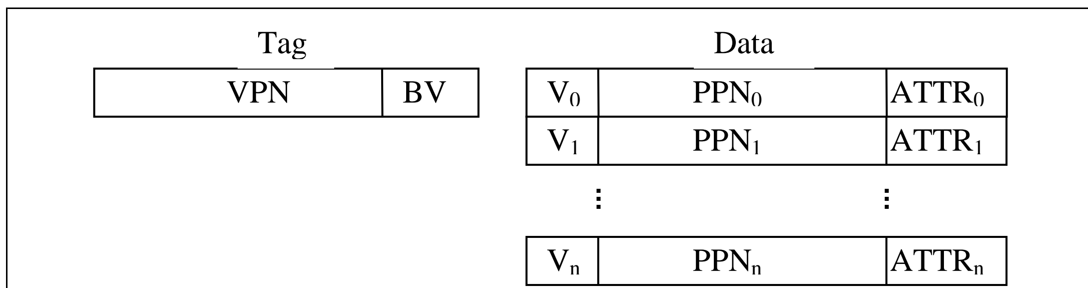


Figure 5: Complete-subblock TLB block (subblock factor n).

Take a complete-subblock TLB with subblock facor 4 and with 4KB base

page for example, the VPN in tag RAM represents the tag of a 16KB page and each PPN in data RAM represents one of four sequential 4KB pages. The disadvantages of complete-subblock TLB are:

- The unused slots at each TLB block may occur and seriously waste hardware cost.
- If one small page entry is to be updated, all four 4KB pages could be invalidated, resulting in performance degradation.

2.2.4 Low Power TLB

- **Banked-promotion TLB structure**

The proposed dual TLB is a new structure which counteracts the defects of the complete-subblock TLB and supports two page sizes via a hardware approach. The proposed dual TLB is organized as two parts of a conventional small page (4KB size) TLB and a large page (16KB size) TLB. Both are designed with fully associative structures. Figure 6 shows the promotion TLB.

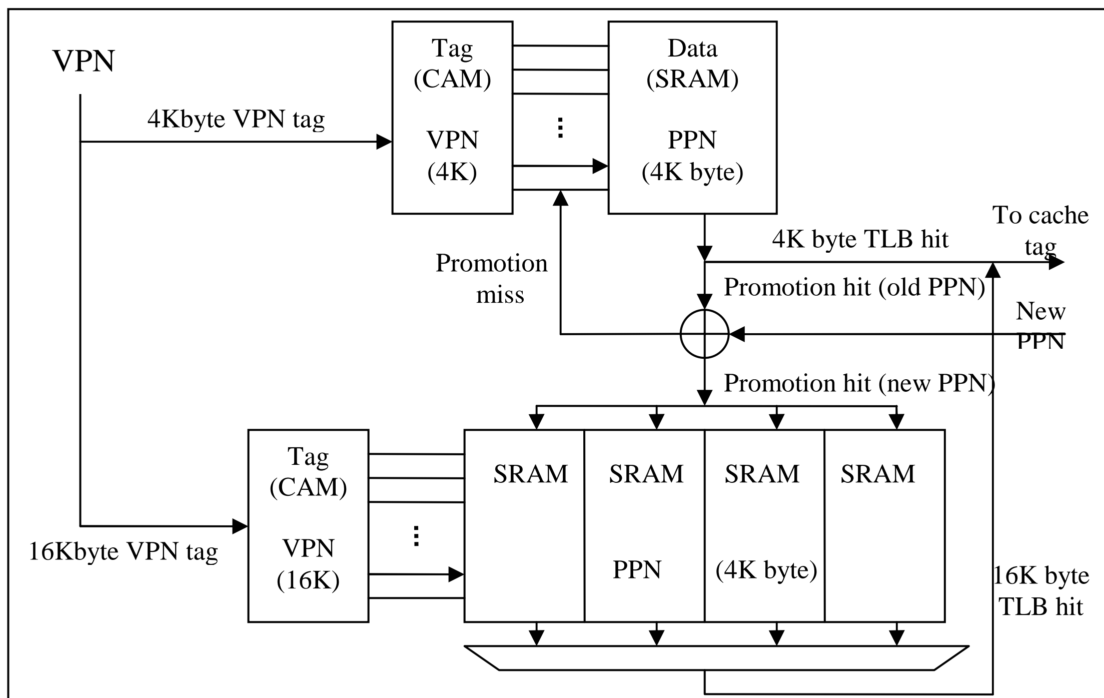


Figure 6: Dual TLB structure

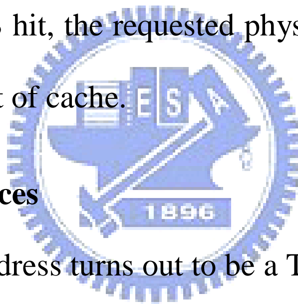
When one virtual address is generated, the 4KB page and 16KB page TLBs will be searched at the same time. The behavioral principle of the dual TLB is as follows:

Case 1: Hit in small page TLB

If a small page is founded in the small page TLB, the actions are not different at all from any conventional TLB hit. The requested physical address is sent to the cache and compared with tag bit of cache.

Case 2: Hit in large page TLB

If a hit occurs at the large TLB, only one of four PPNs in the large TLB is enabled at the same time. Thus the power consumption is decreased. As in the case of a small page TLB hit, the requested physical address is sent to the cache and compared with tag bit of cache.



Case 3: Miss in both places

When one virtual address turns out to be a TLB miss in both small page and large page TLB, O/S performs miss handling. When a TLB miss occurs and if its corresponding three sequential VPN exists in the small page TLB, those four sequential VPNs belonging to a 16KB page boundary are chosen to be promoted, and the three sequential VPNs in the small page TLB are invalidated at the same time.

2.3 The Context Switching Penalty in TLB

In the chapter 1 we discuss the context switching problem which causes the TLB in the MMU to be flushed, and the miss penalty will impact on TLB performance seriously. However, very few attempts have been made for this issue. In this section we will show some simulations to confirm that flushing TLB entries when context switching would cause the miss rate increase.

Before the first step in our analysis of TLB misses, we must know what category of the TLB misses caused by context switching belongs to. General speaking, we can classify all TLB misses into three simple categories:

- **Compulsory misses** — The first access to a block cannot be found in the TLB, so the block must be brought from main memory into the TLB. These are also called *cold-start misses* or *first-reference misses*.
- **Capacity misses** — If the TLB cannot contain all the blocks need during execution of a program, capacity misses will occur because of blocks being discarded and later retrieved.
- **Conflict misses** — If the block placement is set associative or direct mapped, conflict misses will occur because a block may be discarded and later retrieved if too many blocks map to its set. These are also called *collision misses* or *interference misses.*

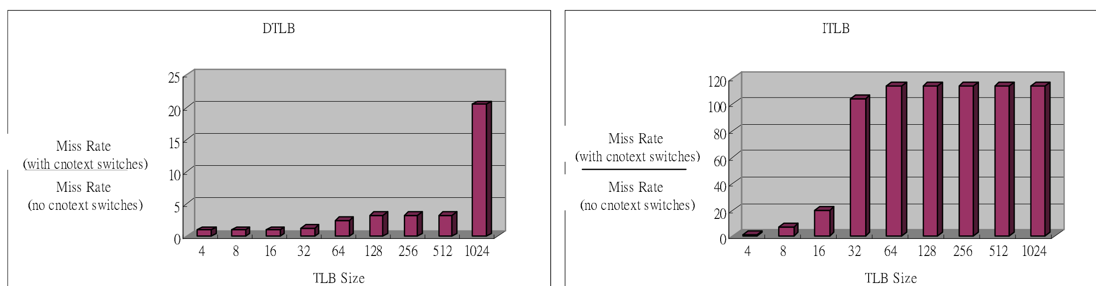
Obviously, the TLB misses caused by context switching belong to the compulsory misses due to flushing TLB when context switching. Now, we will take a close look at some simulations of TLB misses under context switching in comparison with TLB misses without context switching. We assume the context switching would happen after executing one million instructions.

We want to see how many times the miss rate with context switching are as large as with no context switching. Figure 7 shows the times of the miss rate with context switching divided by the miss rate without context switching for the twelve SPEC2000 benchmarks using from 4-entry to 1024-entry fully-associative TLB with a page size of 4KB. In addition, we make a comparison between different page size, 4KB, 16KB and 64KB for the times in Figure 8.

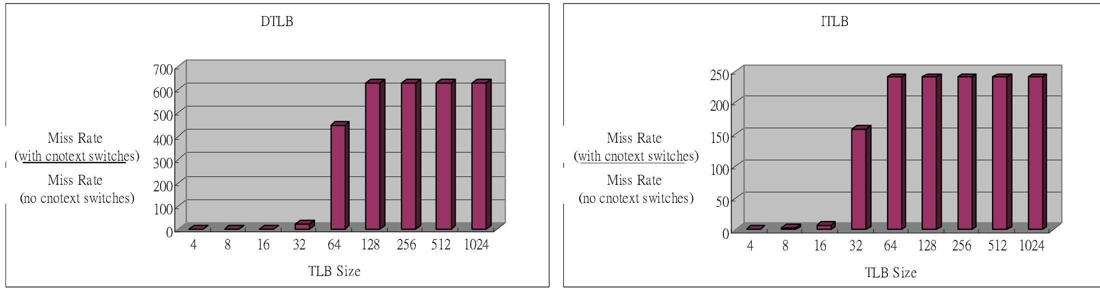
The Figure 7 and Figure 8 tell us that:

- The times of the miss rate with context switching divided by the miss rate without context switching would increases if the number of TLB blocks increases. In other words, the context switching would impact on the TLB performance more seriously if we have more TLB entries. However, the most simple way of reducing the TLB miss rate is to increase TLB entries; for example, the lease AMD OpteronTM processor has both 512-entry L2 instruction TLB (ITLB) and L2 data TLB (DTLB) [1] and the IBM POWER processor has a common 1024-entry TLB for each processor core [28].
- If the application, such as *vortex,* need only short computation time and the computation can be finished before the first context switching, the TLB could keep performance from decreasing.
- The context switching would impact on the TLB performance more seriously if we increase page size.

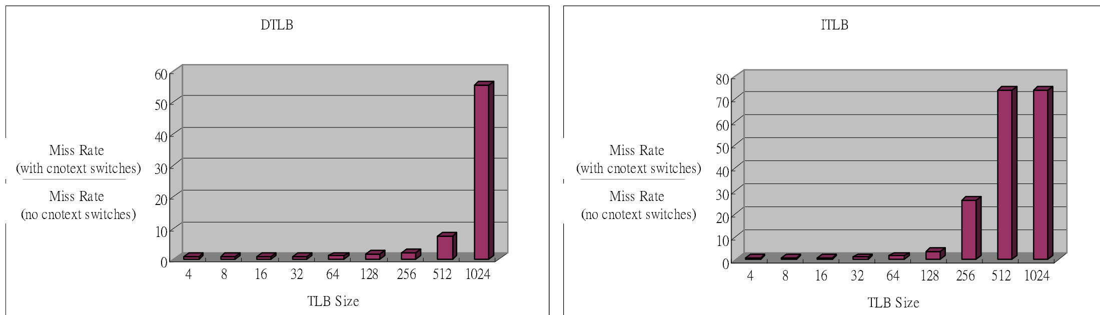
GZIP



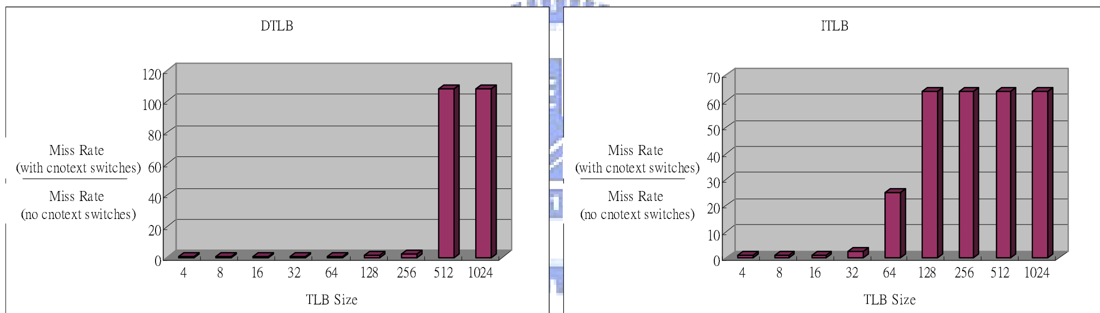
VPR



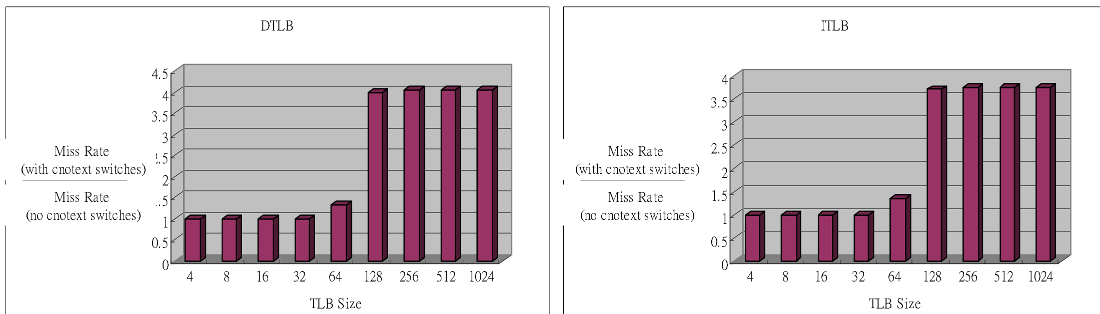
GCC



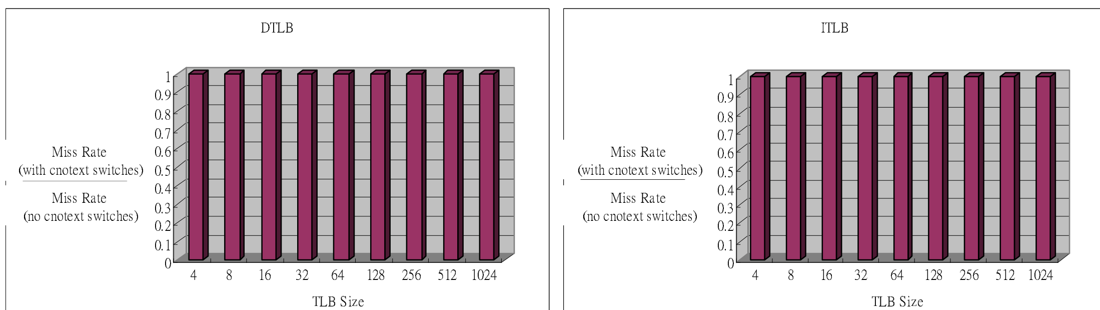
CRAFTY



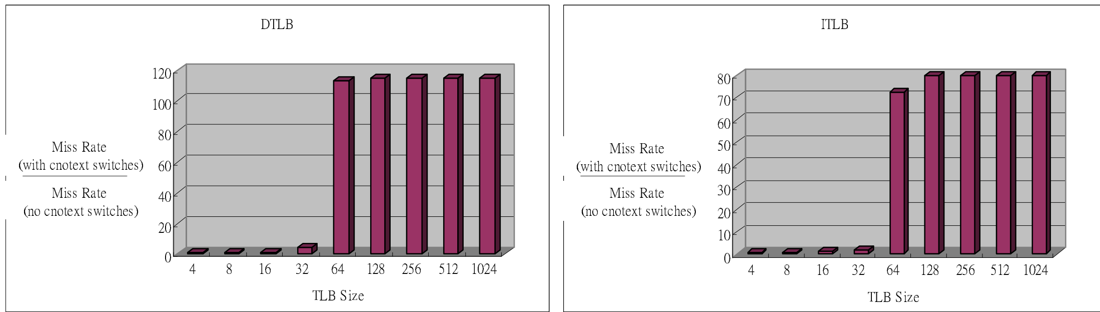
PERLBMK



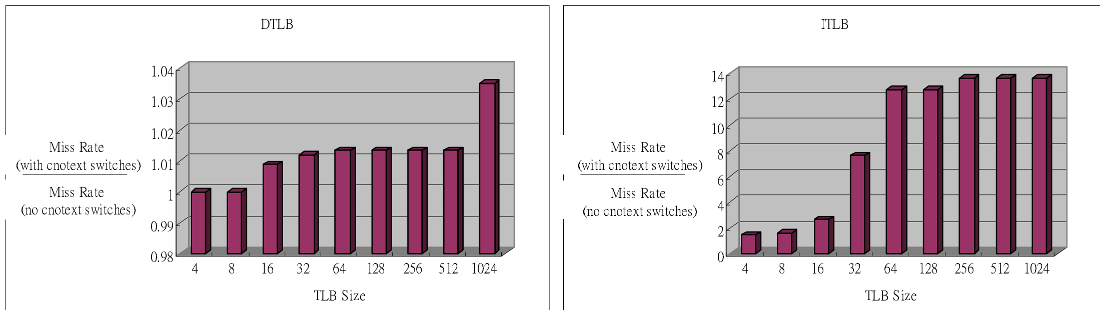
VORTEX



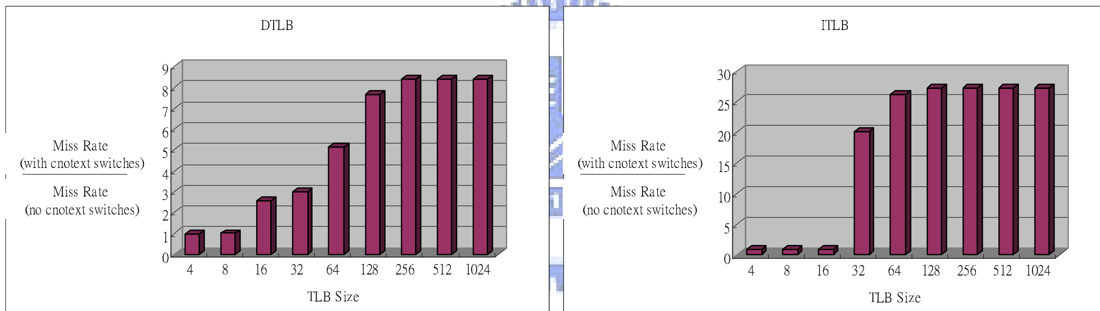
TWOLF



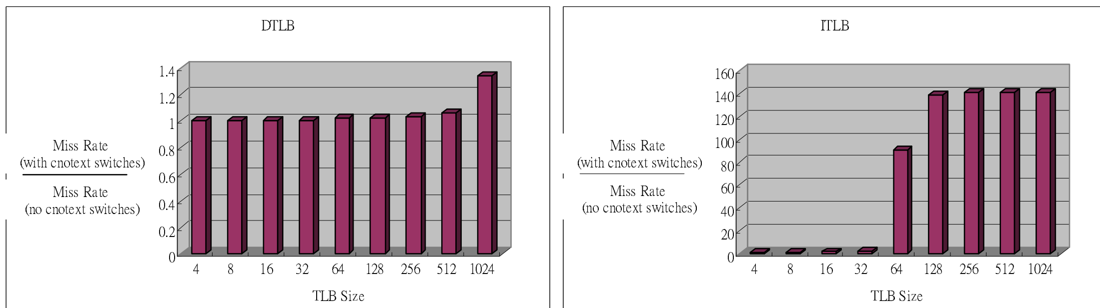
SWIM



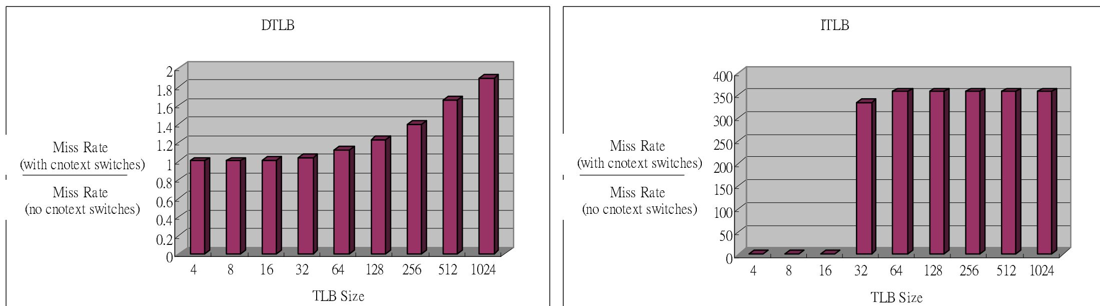
LUCAS



APPLU



EQUAKE



MGRID

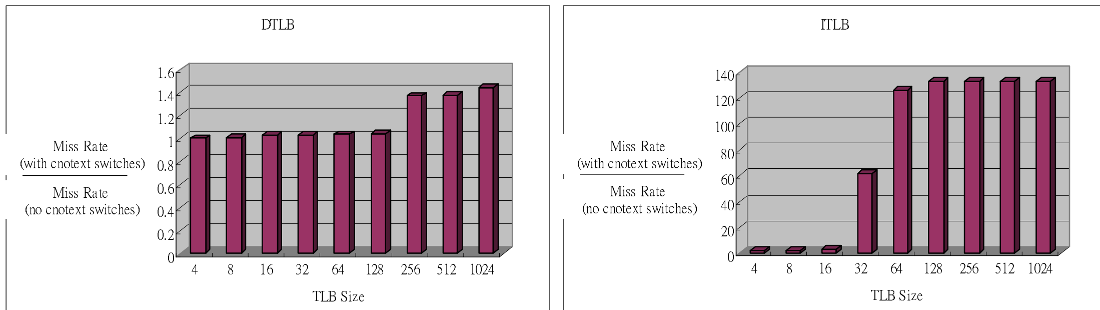
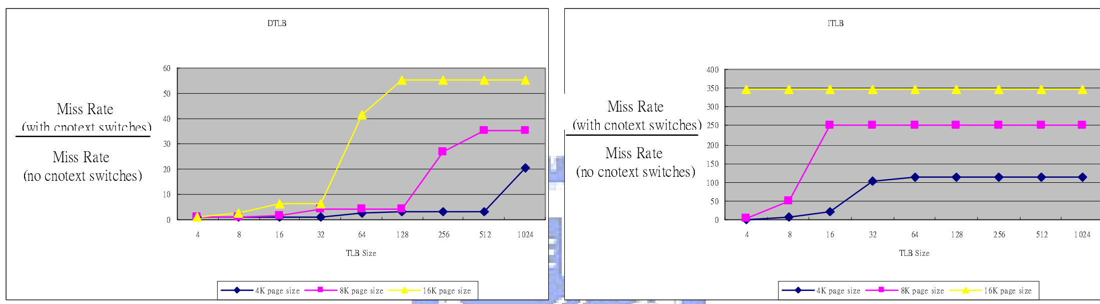
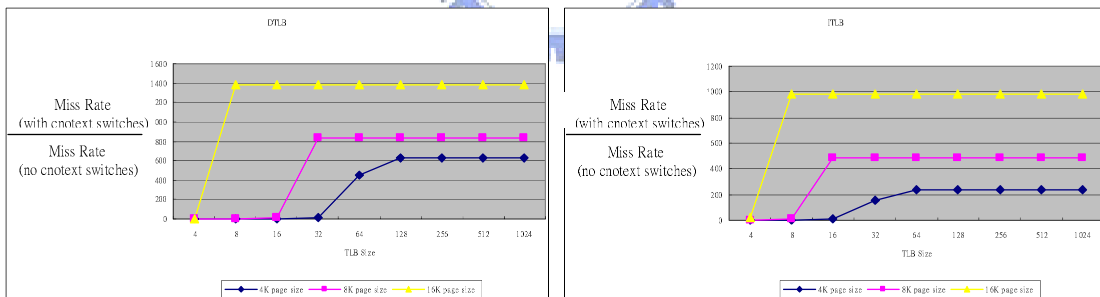


Figure 7: The relationship between the TLB size and the miss rate with context switching divided by the miss rate without context switching.

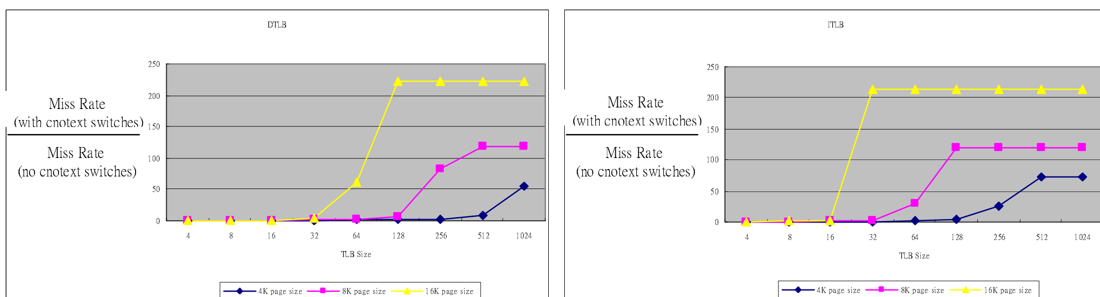
GZIP



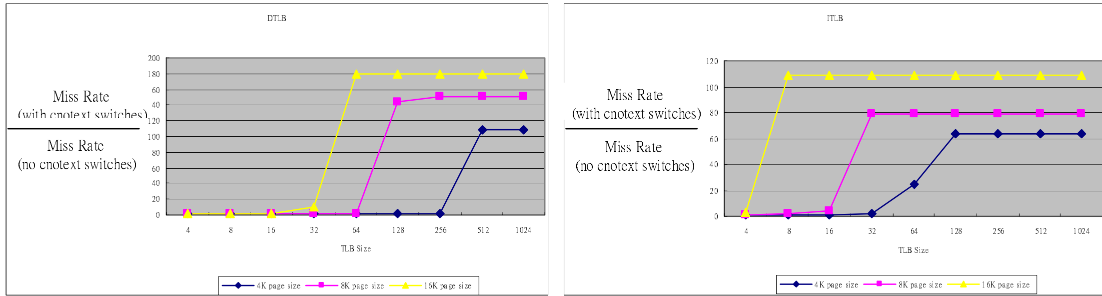
VPR



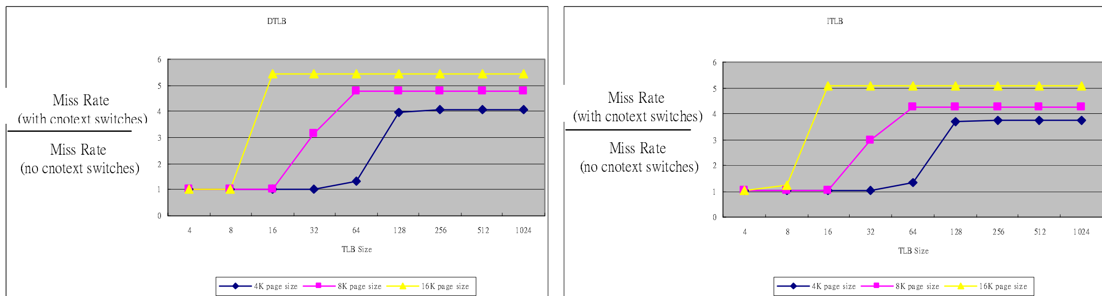
GCC



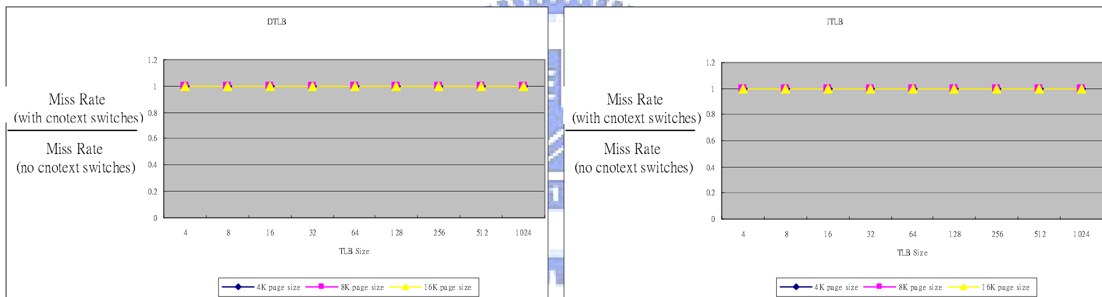
CRAFTY



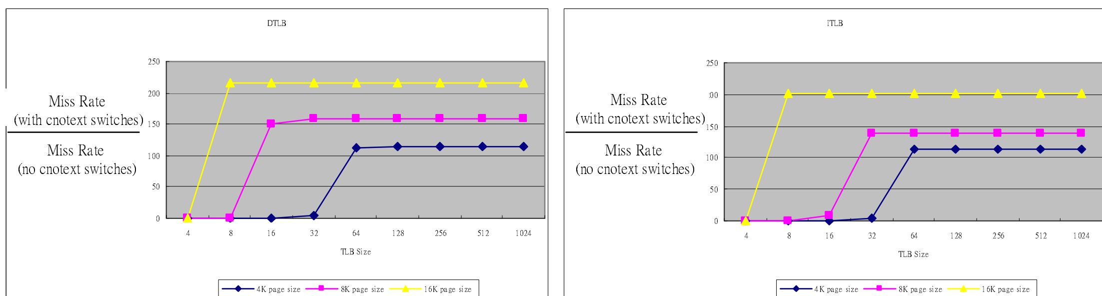
PERLBMK



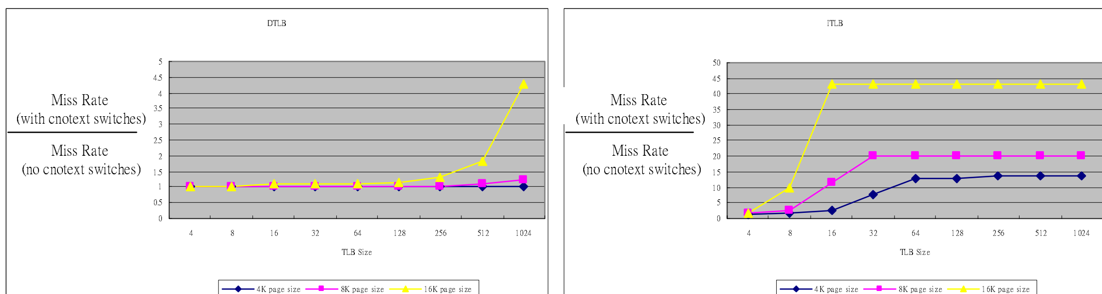
VORTEX



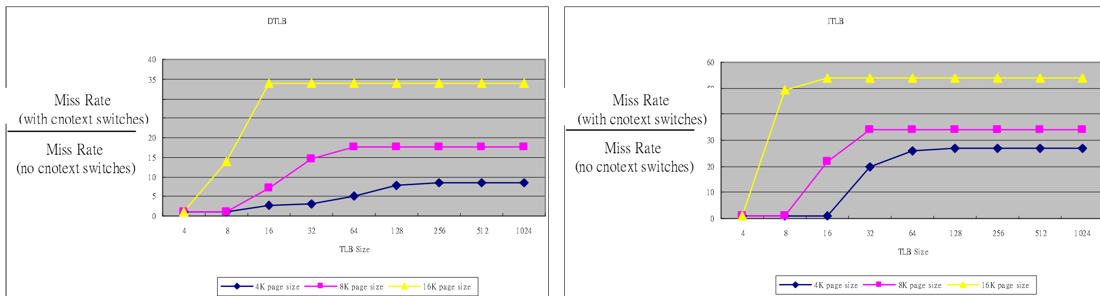
TWOLF



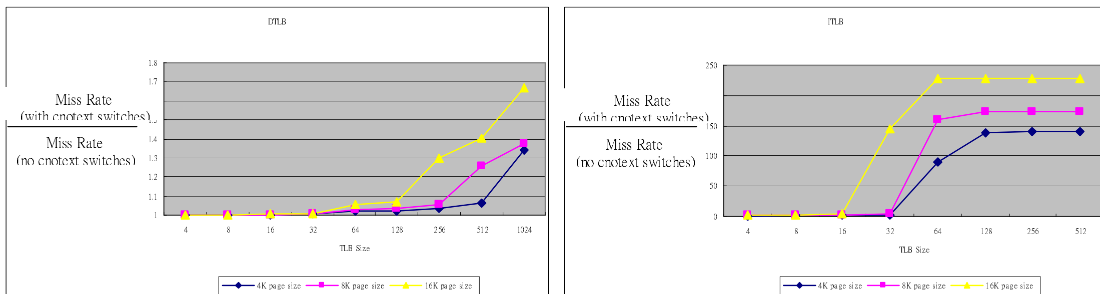
SWIM



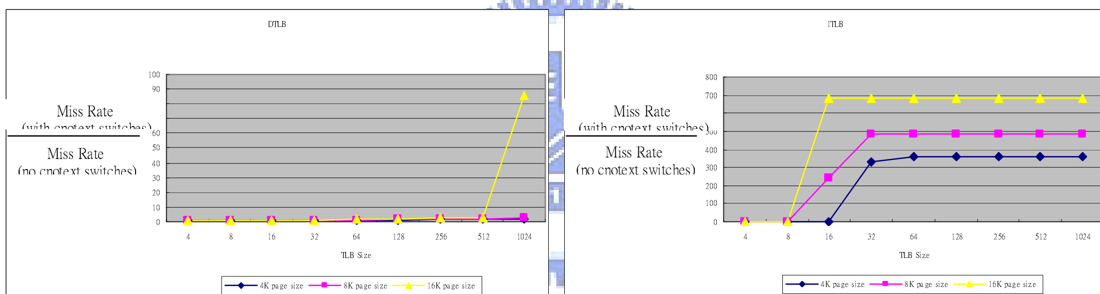
LUCAS



APPLU



EQUAKE



MGRID

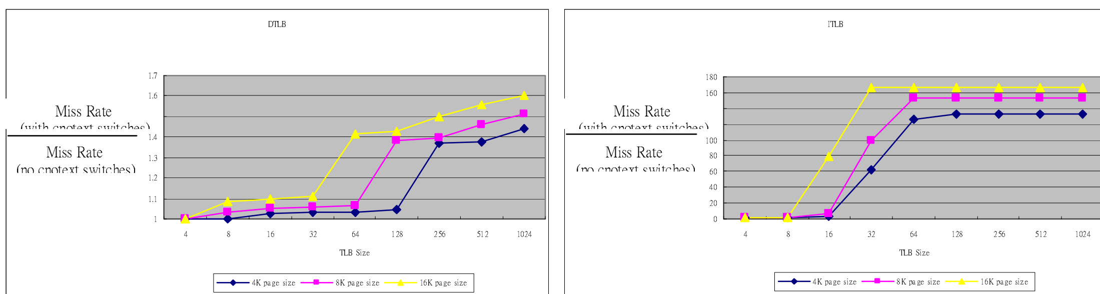
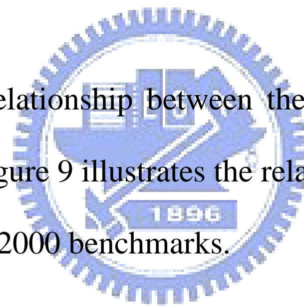


Figure 8: The relationship between the page size, TLB size and the miss rate with context switching divided by the miss rate without context switching.

2.4 Relationships between the Miss Rates, Page Sizes and TLB Sizes

In this section we will limit the discussion to the relationships between the miss rates, page sizes and TLB sizes, and will not be concerned with the issue of context switching. It is well-known that the most important two issues for cache system performance are to reduce the miss rate and miss penalty. It is almost the same for the TLB performance. In fact, the most important of all is the miss rate issue. That's why we focused on the miss rate in our research. In order to select the suitable page size, we did some study on the relationships between the miss rates, page sizes and TLB sizes.

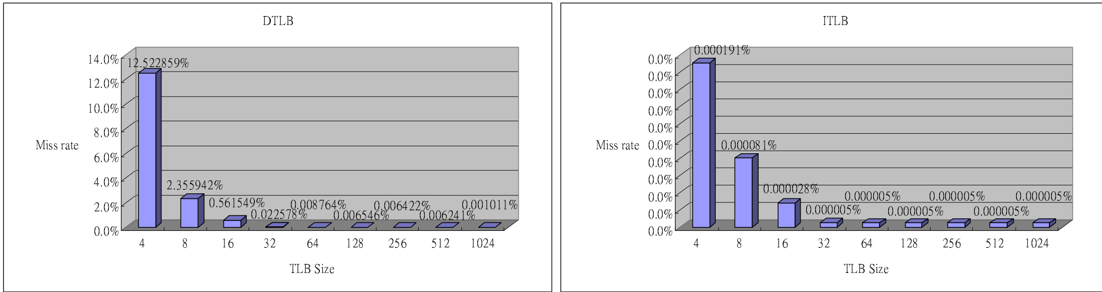


First, we consider the relationship between the miss rate and TLB sizes with traditional 4KB page sizes. Figure 9 illustrates the relationship between TLB sizes and the miss rate for twelve SPEC2000 benchmarks.

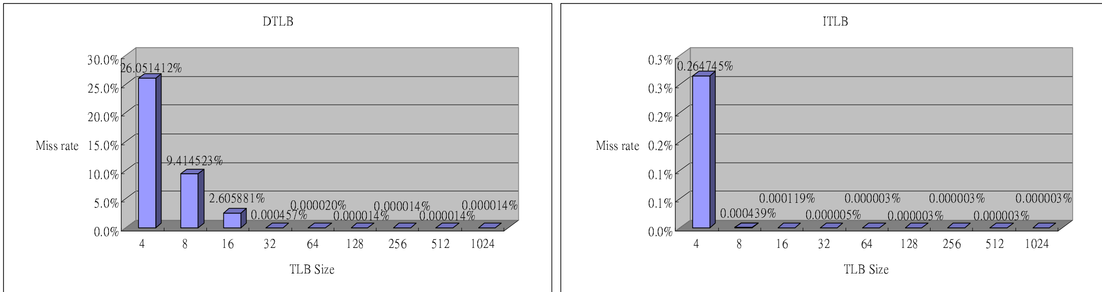
The Figure 9 tells us that:

- Some applications such as *gzip*, *gcc*, *crafty*, *perlbmk*, *vortex*, *swim, applu, equake and mgrid* have better performance with sizes over 128 entries.
- For some applications such as the *vpr*, *twolf* and *lucas* benchmarks, it is very clear that only 64 or less entries are enough and it is helpless to increase the number of TLB entries.
- These benchmarks seeks to capture the fact that it does not really need to provide TLB with over 128 or 256 entries with 4KB page size and we can consider what these extra entries can do.

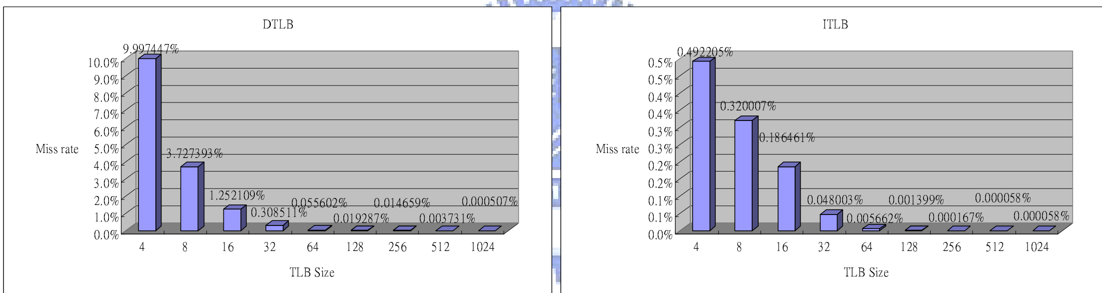
GZIP



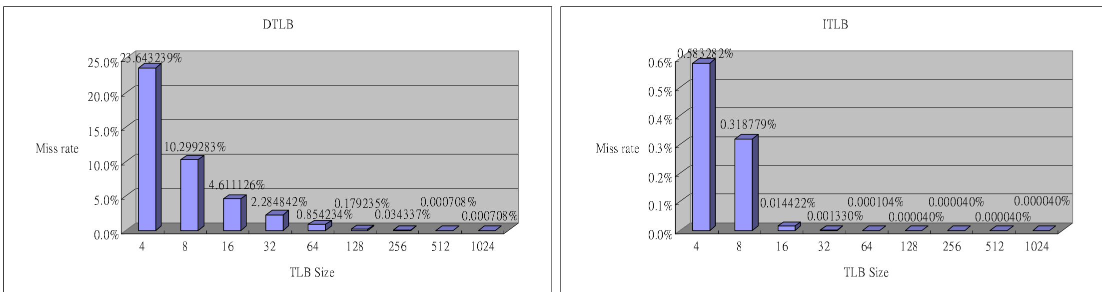
VPR



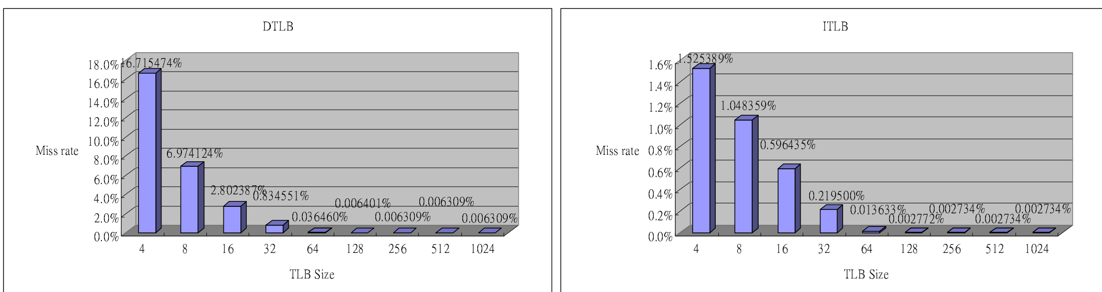
GCC



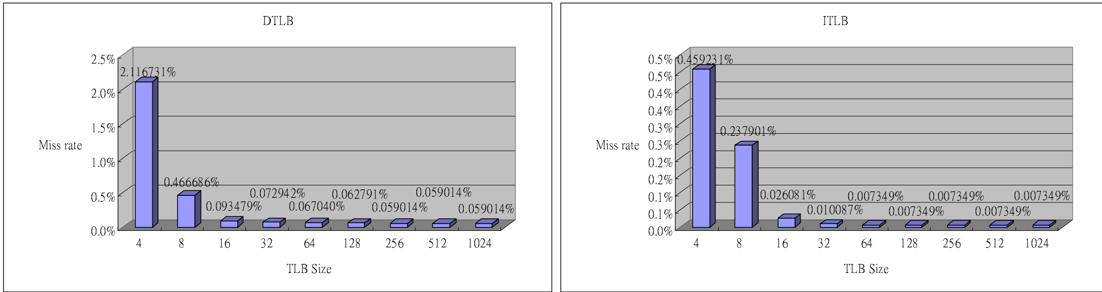
CRAFTY



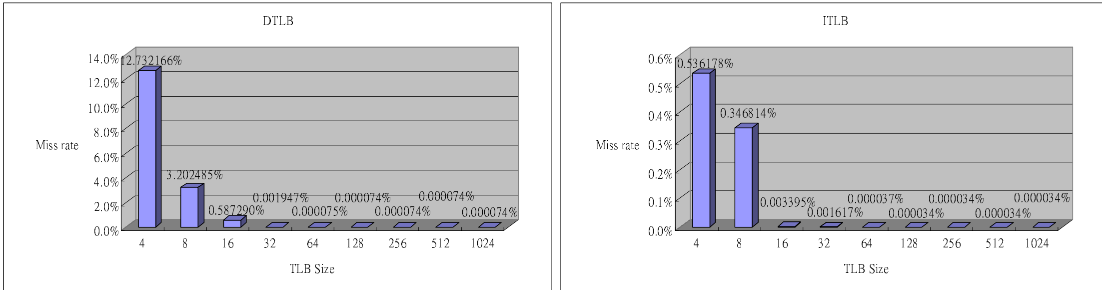
PERLBMK



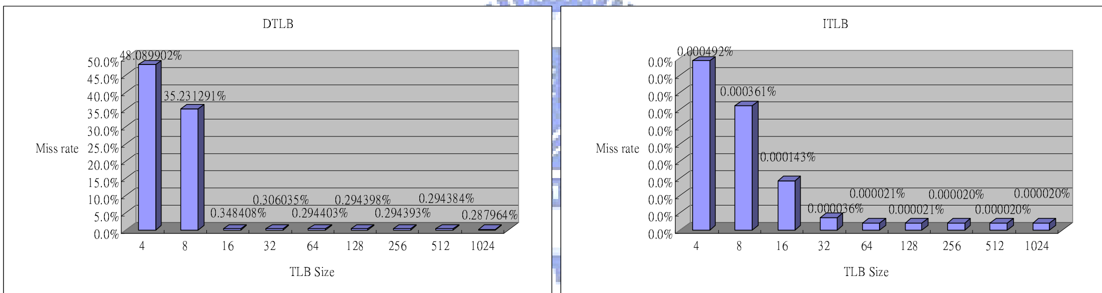
VORTEX



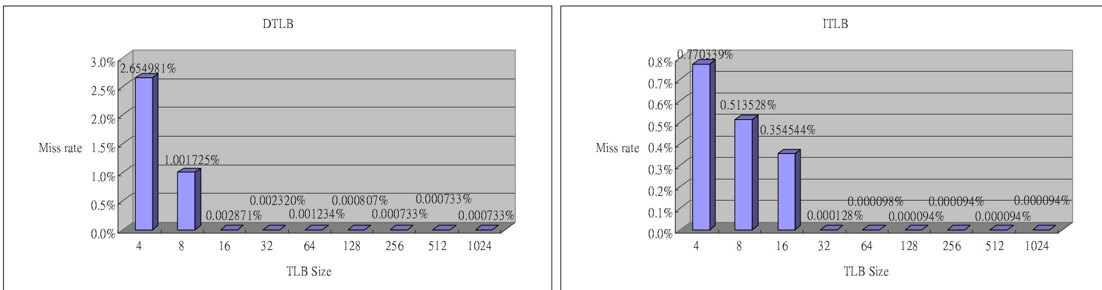
TWOLF



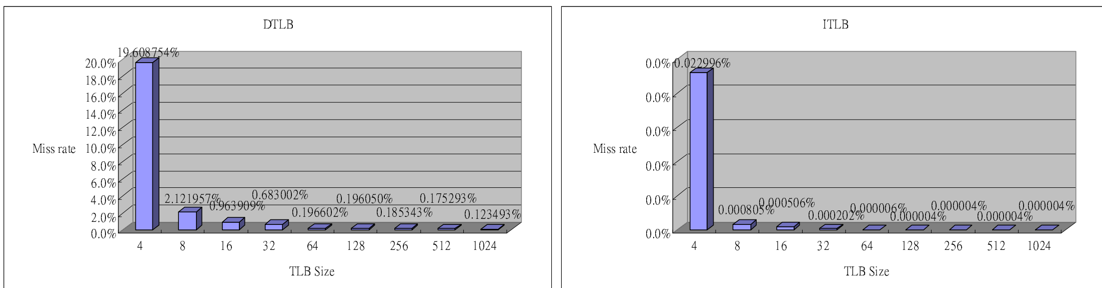
SWIM



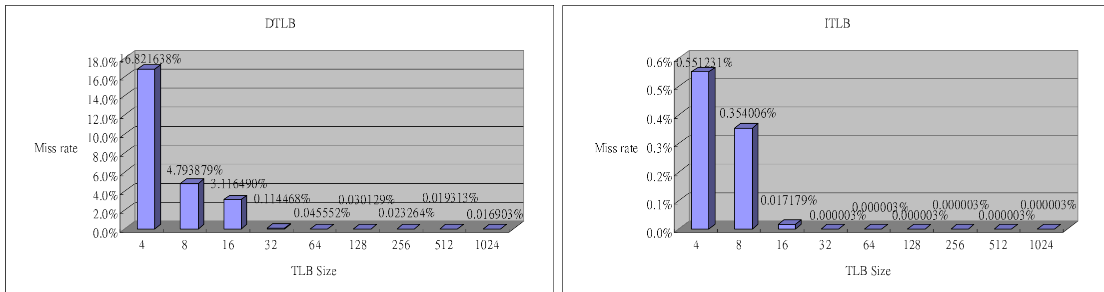
LUCAS



APPLU



EQUAKE



MGRID

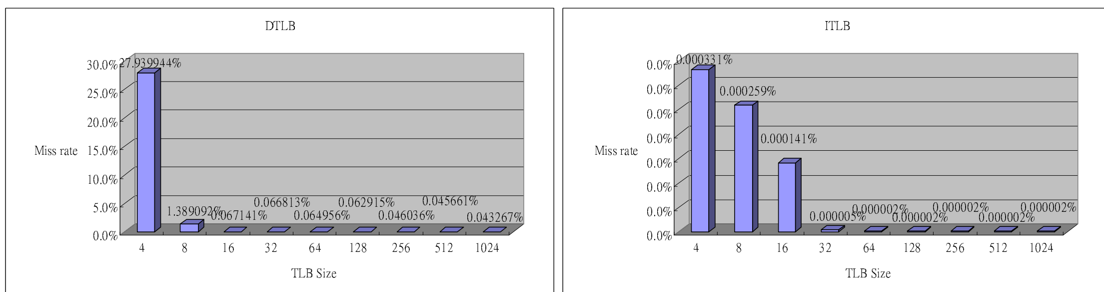
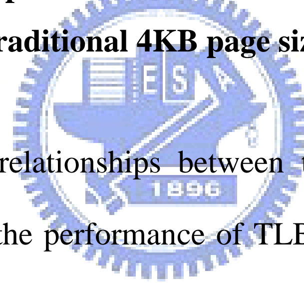


Figure 9: The relationship between the TLB miss rate and TLB sizes with traditional 4KB page sizes.



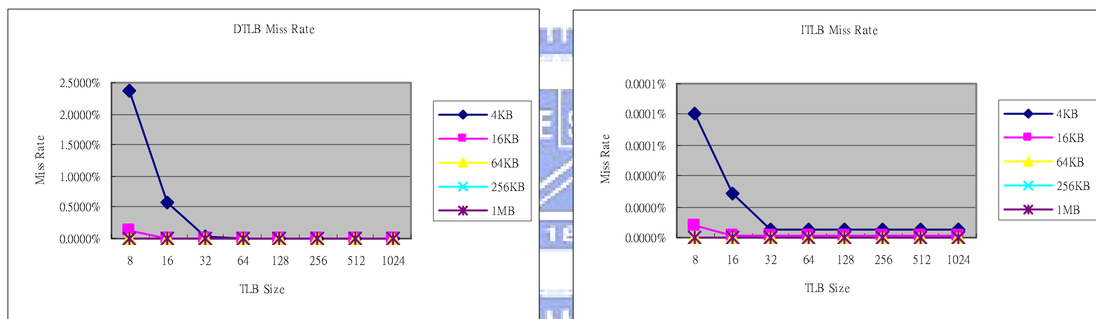
Next, we consider the relationships between the miss rates and page sizes. Another solution to improve the performance of TLB is to extend the page size into larger one. It is easy to find that some modern processors begin to provide multiple page sizes, such as 4KB, 2MB and 4MB sizes, on all Intel advanced x86 processors after the Pentium Pro processor [13]. The advantages of larger page sizes are not only obtaining better performance but saving the implementation cost with less tags (virtual page number, VPN) and translations (physical page number, PPN) needed to be stored. It is also a good method to reduce the cost on TLB implementation of processors with larger addressing space, such as processors with 64-bit addressing space. Certainly, it is suitable to implement on the processors core of SoC or embedded systems.

What would happen if we extend the page size to 16KB, 64KB, 256KB and 1MB.

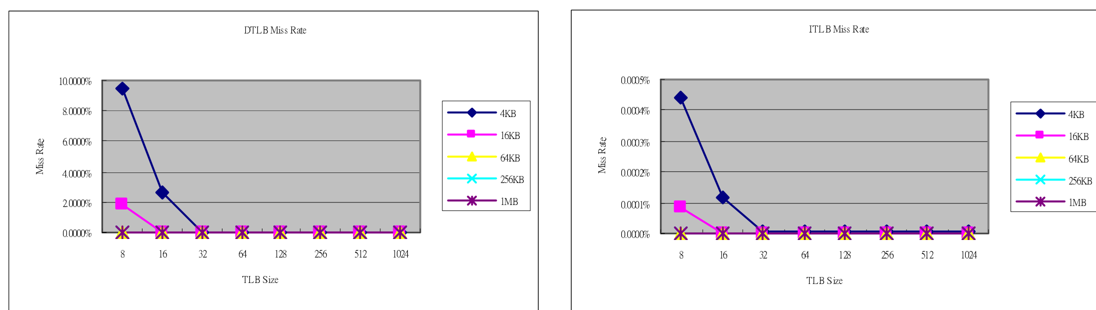
Figure 10 below shows the miss rate for twelve SPEC2000 benchmarks of 4KB, 16KB, 64KB, 256KB and 1MB page sizes with eight TLB sizes – 8, 16, 32, 64, 128, 256, 512 and 1024. Observing the results, the figure indicates that for all benchmarks:

- The ITLB/DTLB performance of 1MB (or 256KB) page with eight entries can greatly outperform that of 4KB page with 256 entries.
- It is helpless to increase the number of TLB entries when we using large page size for all benchmarks.
- Benchmarks which scatter references across a sparse address have little needs from large pages without significantly increased memory usage.

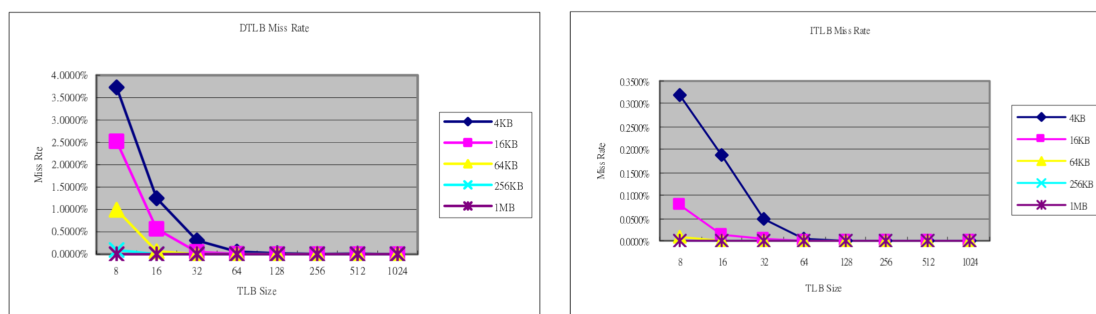
GZIP



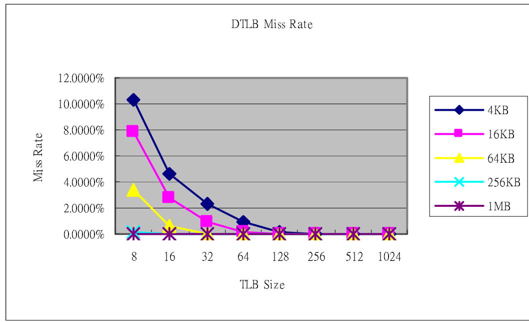
VPR

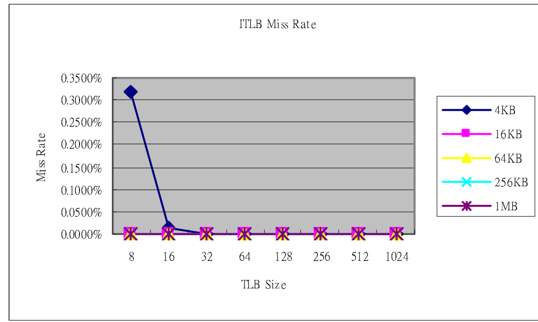


GCC

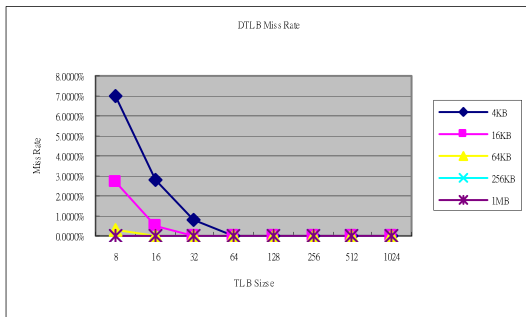


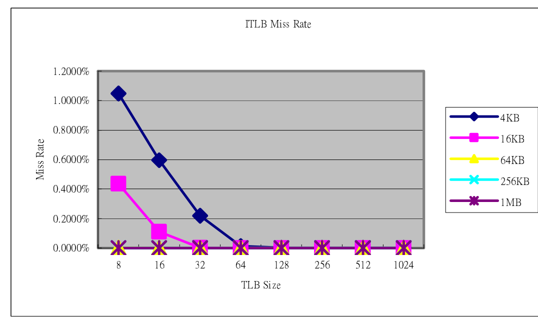
CRAFTY



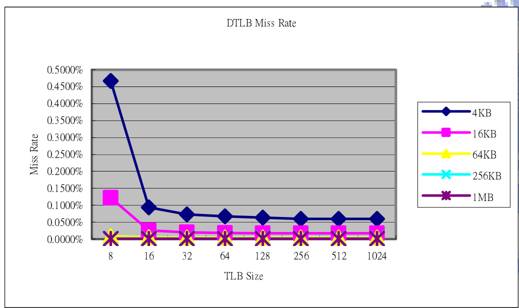


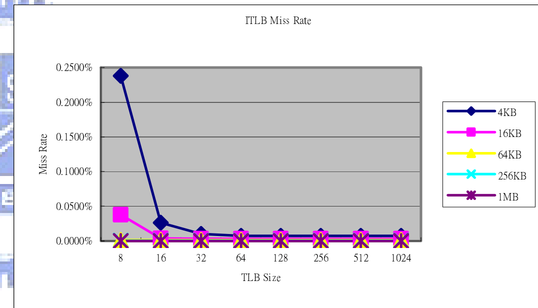
PERLBMK



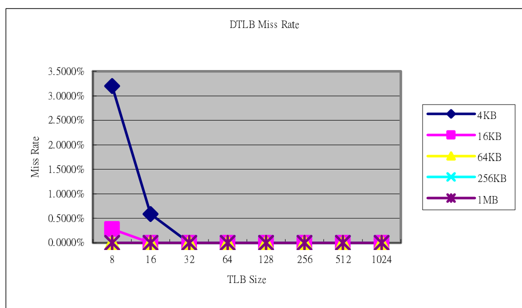


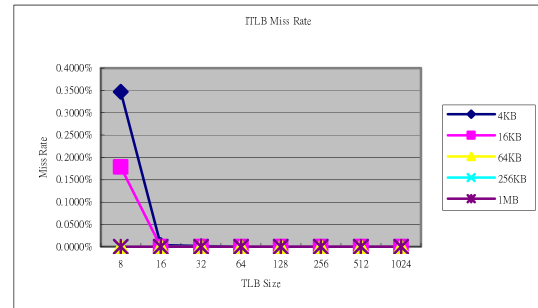
VORTEX



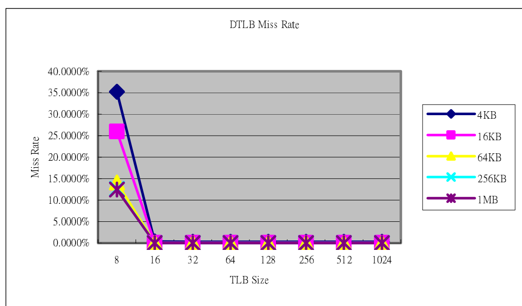


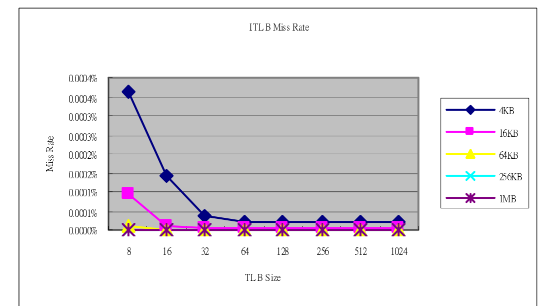
TWOLF



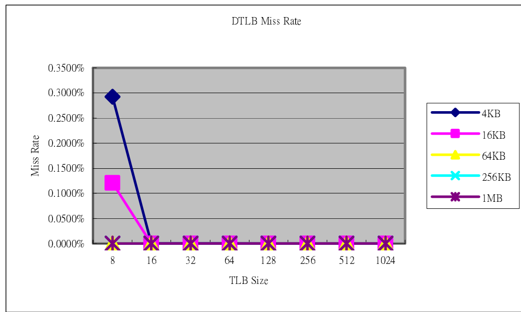


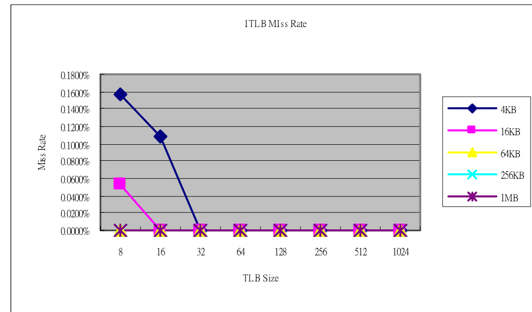
SWIM



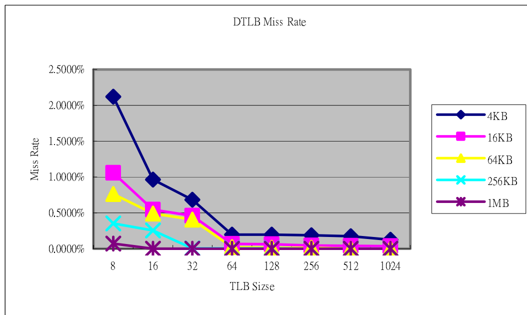


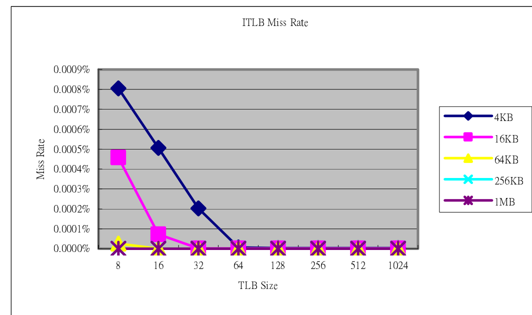
LUCAS



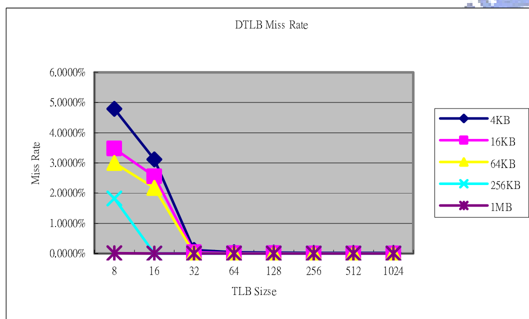


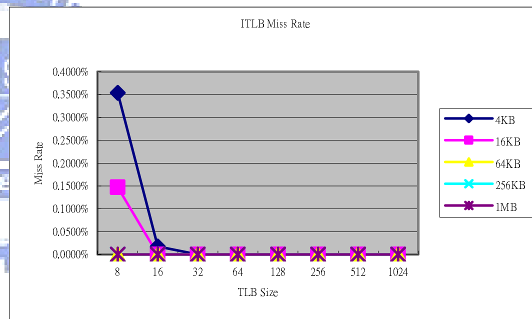
APPLU



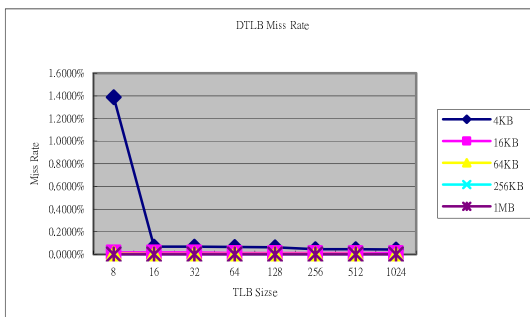


EQUAKE





MGRID



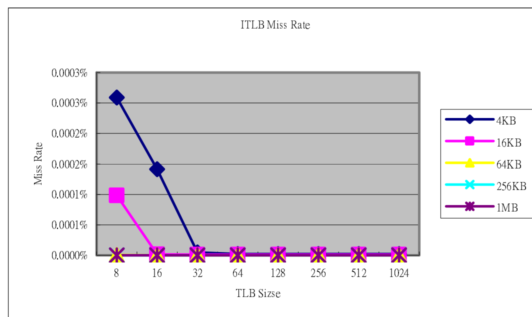


Figure 10: The relationship between the TLB miss rate and page size.

Chapter 3 Proposed Mechanisms

Chapter 3 describes in detail with the two TLB structures we proposed for low context switching penalty. Our TLB structures can be implemented not only in contemporary processors but future processors comprised with one billion of transistors. Furthermore, they are especially suitable to be implemented on processors with larger addressing space than current processors with just 32-bit addressing ability. We will review our original TLB architecture for processors with larger page size support in Section 3.1. Then, we propose our new TLB architecture with general page size, such as 4KB, 8KB, or 16KB, in the Section 3.2. Last, we will discuss the mechanisms of our new novel TLB in Section 3.3.

3.1 The Original TLB Structure with Low Context Switch Penalty

To reduce the miss rate, most designs just try to increase the TLB size to reduce the capacity misses; however, we have showed in previous chapter that it is also helpful if the page sizes can be enlarged. Furthermore, with large page size, we can make use of more redundant TLB entries to store translation information of other tasks and the size of tags and translations needed to be stored can be much smaller. Thus, we used 1MB page size in our design.

3.1.1 Structure

Figure 11 shows in detail our original TLB structure to reduce the miss rate in context switches. The original TLB structure consists of the following parts — the

TLB banks with group tags, and a multiplexer to select a specific TLB banks.

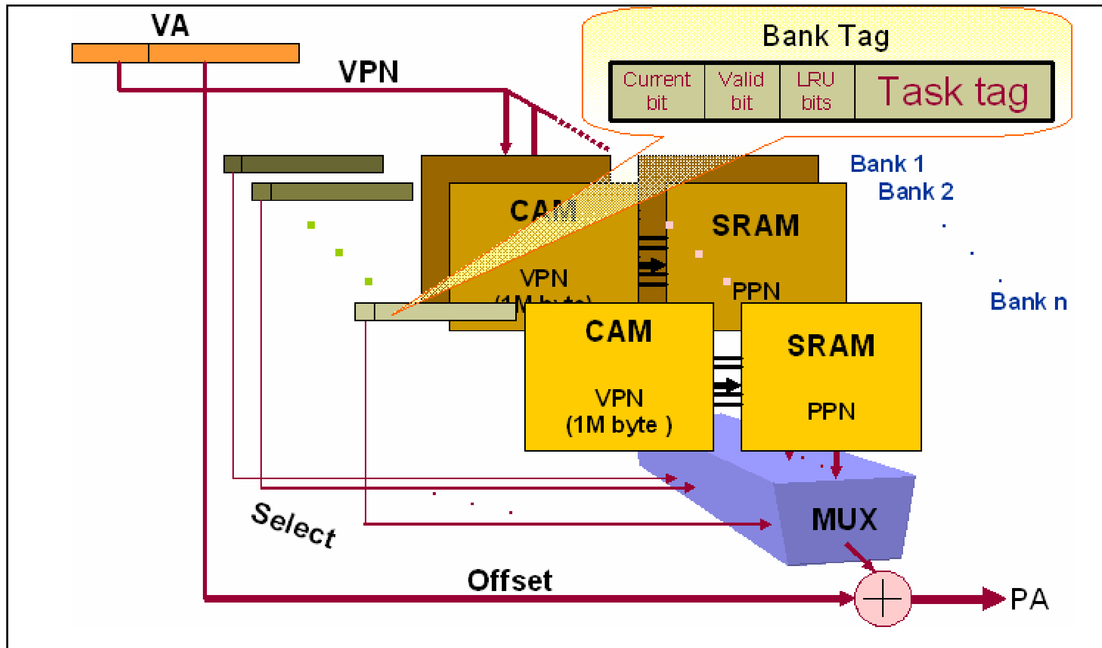


Figure 11: A low context switching miss rate TLB architecture.

Each TLB bank contains eight entries, and the tag can be implemented with CAM (content addressable memory) which is the same as that being implemented on conventional TLB. In addition, each TLB bank is implemented with fully associative with LRU replacement policy. There are total 32 or more TLB banks. Though there are 32 banks, compared with 256-entry conventional TLB the total cost is not increased very much. In fact, there are also total 256 (32x8) entries in our original proposed structure. Furthermore, because of larger page size, the cost of each entry is decreased. Thus the increased cost can be counteracted.

Except the 32 TLB banks, there are also 32 extra registers to store the bank tag as shown in Figure 3.1. The register contains

- *Task tag*: Identify each task.
- *Current bit*: Identify current working task.
- *Valid bit*: Validate/Invalidate a bank.
- *LRU bits*: Replace the victim bank.

We have to point out that the task tag can be PID (process ID) or the PPN (physical page number) of the executing instruction when context switching occurs. The PID is selected as task tag on systems that the PID will be sent into the processor; otherwise, the PPN of the executing instruction when context switch occurs from the PPN field (or last translation) is selected. Considering the general case, the PPN is selected; however, the PID can be more easily selected and implemented under the previous situation. The discussion will be ignored in this thesis.

3.1.2 OS Support and Implementation

In order to implement our original TLB mechanism, the OS is need to do a little modification. Except larger page size, the OS needs to send 'the clear TLB signal' to the processor only when swapping pages with disks occurs or page frames release. Fortunately, it is very easy to realize. Most modern processors provide some ways to flush TLB entries, such as using STA instruction with alternative address on Sparc processors [22].

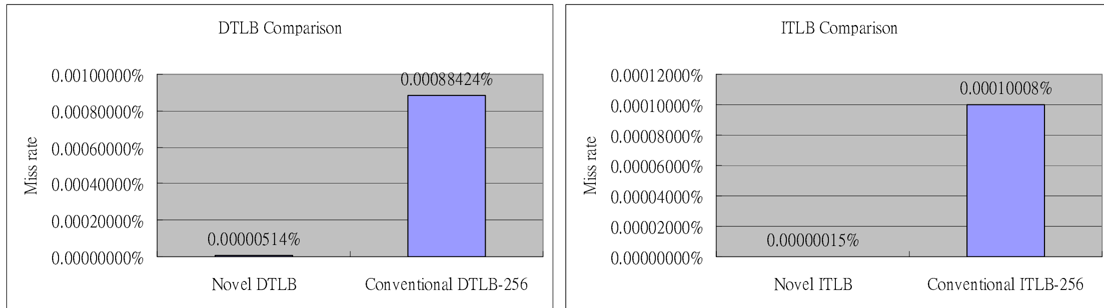
3.1.3 Expected Performance

We simulated the twelve SPEC2000 benchmarks to demonstrate the expected performance. We assumed that the context switching would happen after executing one million instructions. We compared the miss rates of conventional 256-entry TLB with flushing all entries after context switching and our novel TLB structure with 8-entry each bank after correctly keeping entries. The page size is 1MB. Figure 12 shows the simulation results of the SPEC2000 benchmarks.

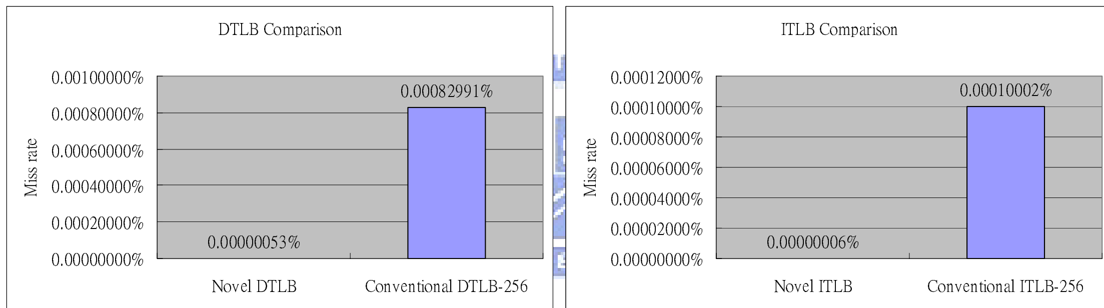
We can find that for most benchmarks we can deliver better performance than the conventional structure for both DTLB and ITLB except the DTLB in the *swim,*

applu and *equake* benchmarks because the working set of the three benchmarks need more coverage space of memory mapping than other benchmarks. It is noteworthy that the *vortex* benchmark needs only shorter computation time than others, and therefore has the same performance as conventional TLB.

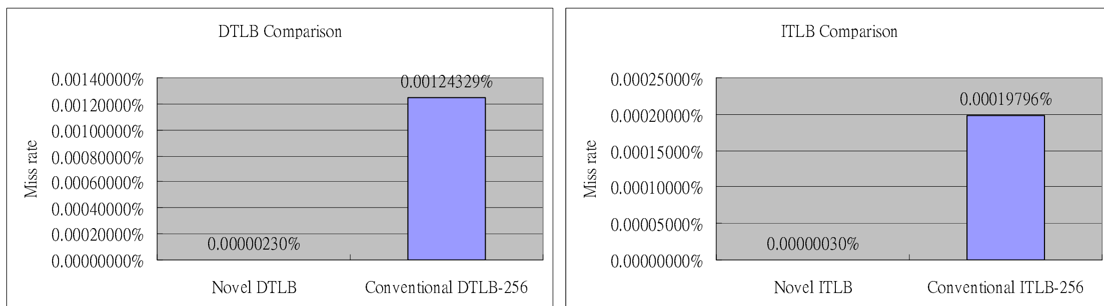
GZIP



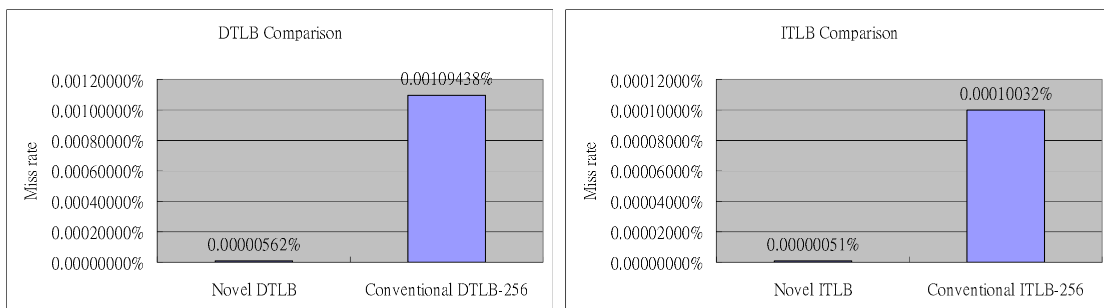
VPR



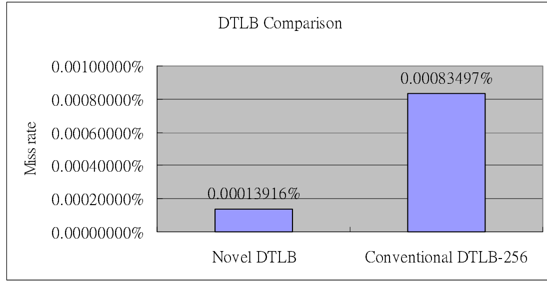
GCC

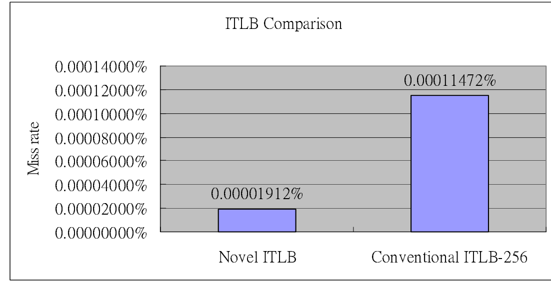


CRAFTY

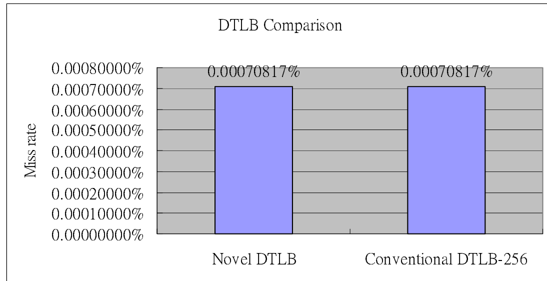


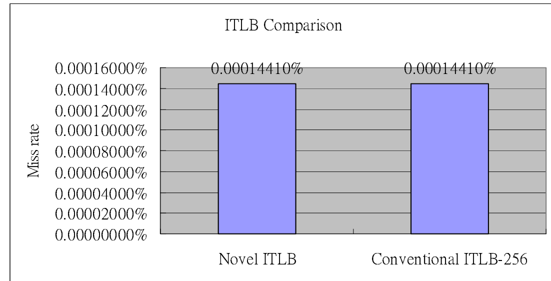
PERLBMK



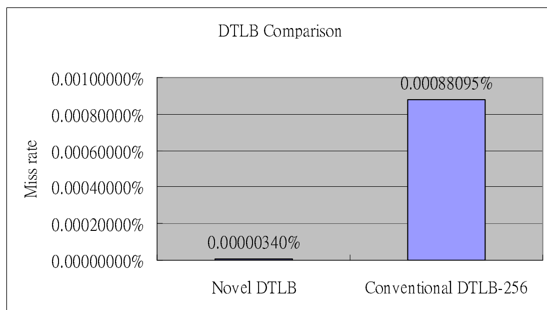


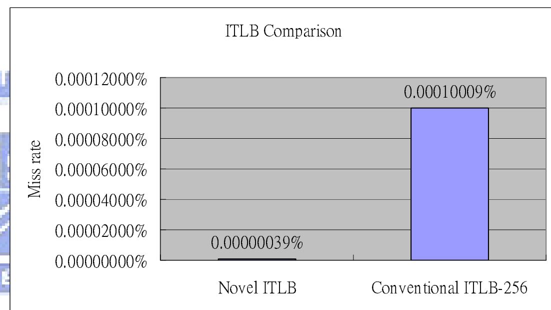
VORTEX



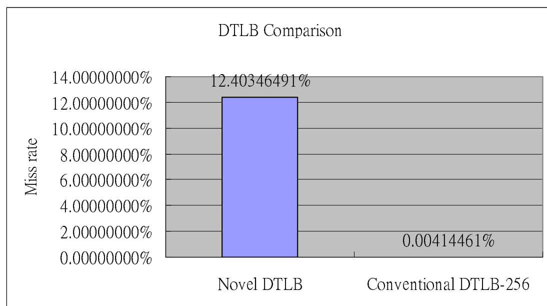


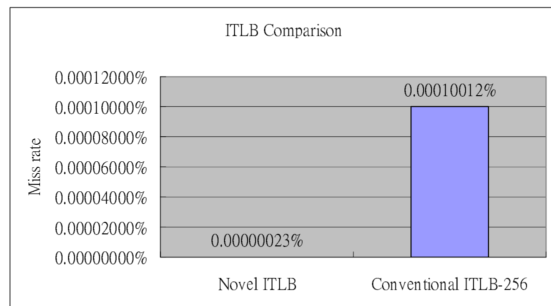
TWOLF



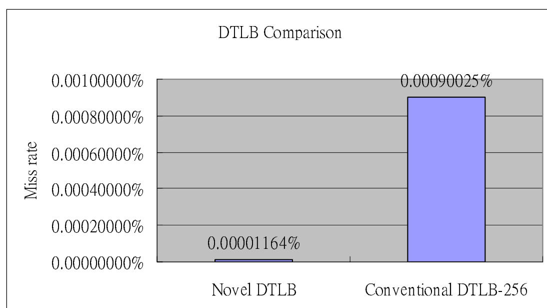


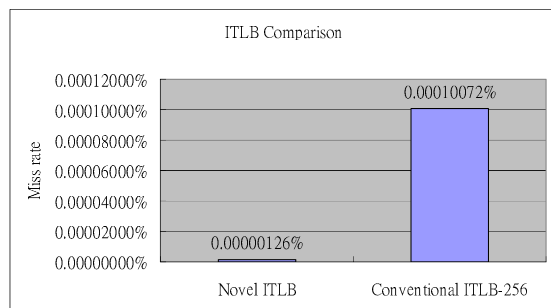
SWIM



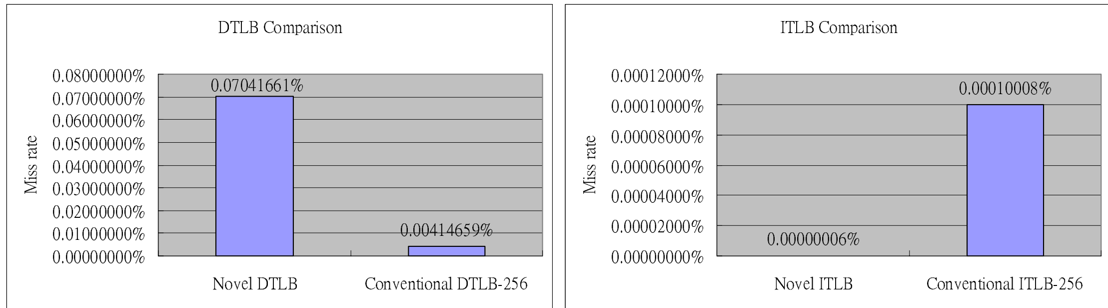


LUCAS

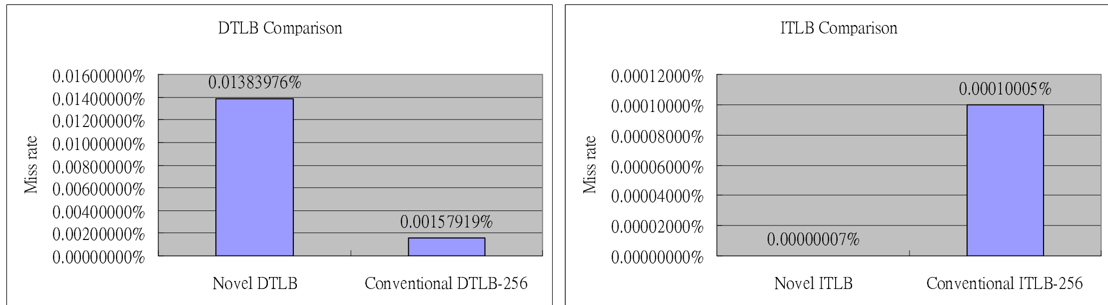




APPLU



EQUAKE



MGRID

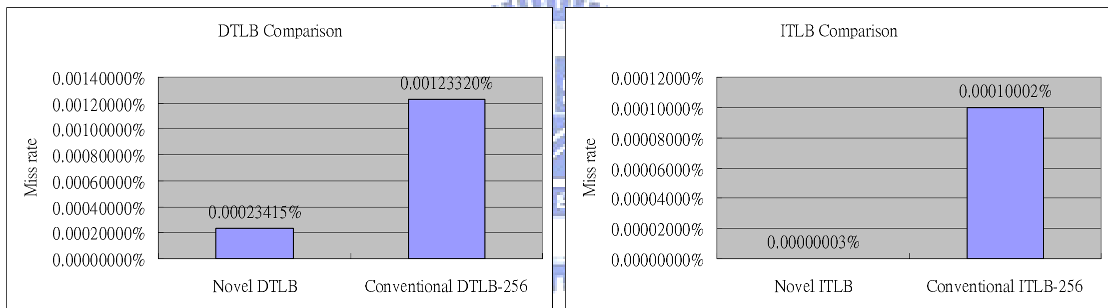
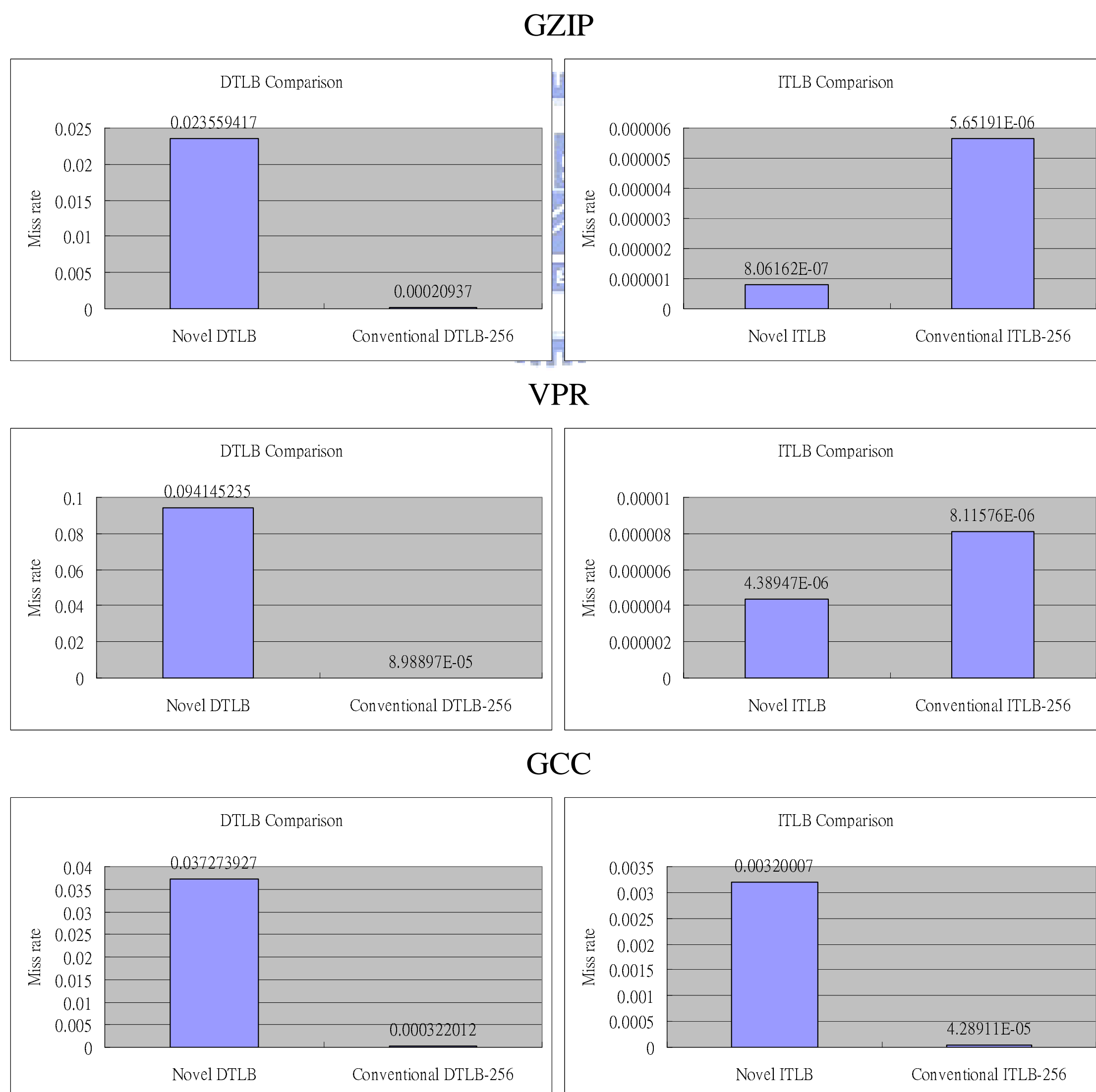


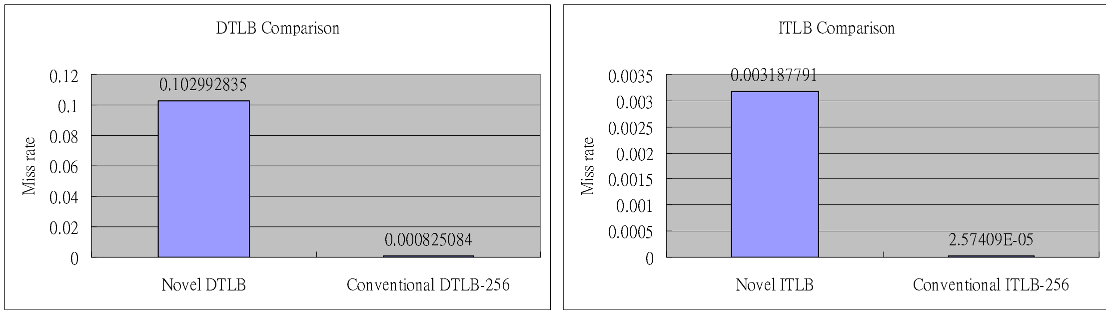
Figure 12: DTLB/ITLB miss rate comparison with 1MB page size.

3.2 The New TLB Structure with Low Context Switch Penalty

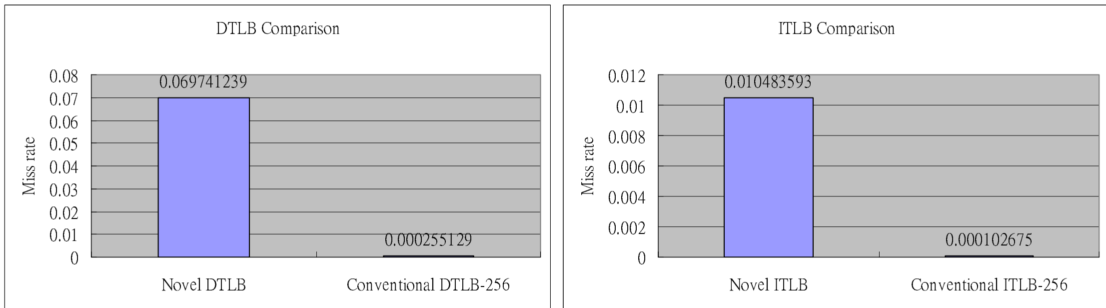
We have surveyed our original TLB mechanism with low context switching penalty in section 3.1. It works well and has good performance when page size is large. On the contrary, it works badly with small page size, such as 4KB, 8KB or 16KB page size. We represent the results when using 4KB page size in our original TLB with 8-entry per bank in Figure 13. As the diagram indicates, the performance obviously degrades and our original structure is not suitable for small page size.



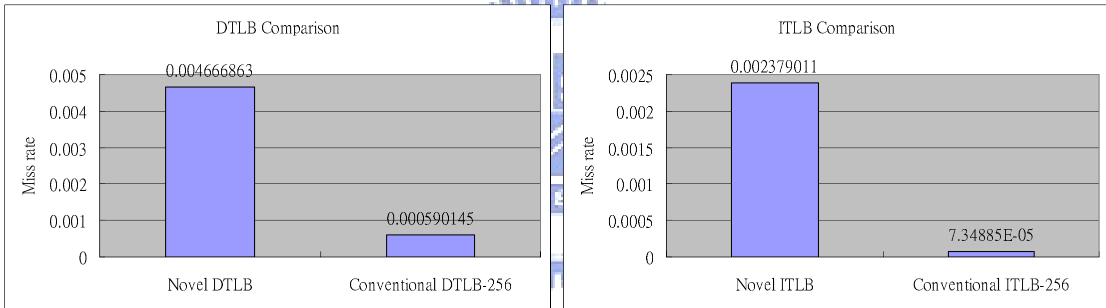
CRAFTY



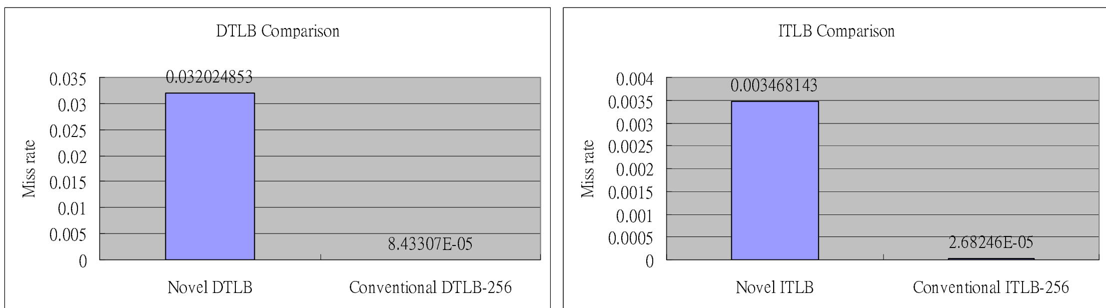
PERLBMK



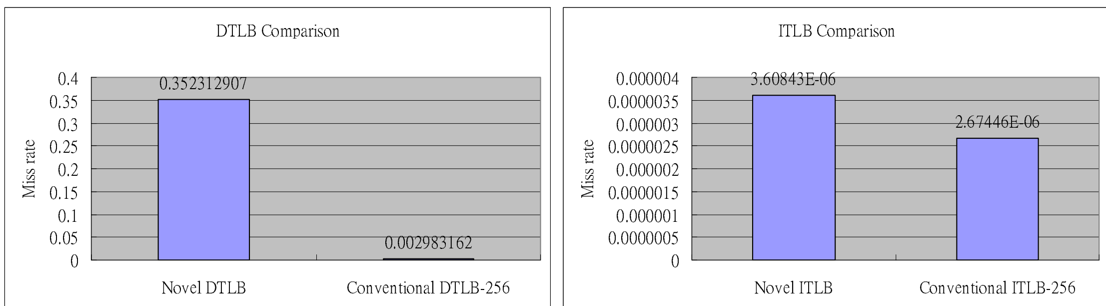
VORTEX



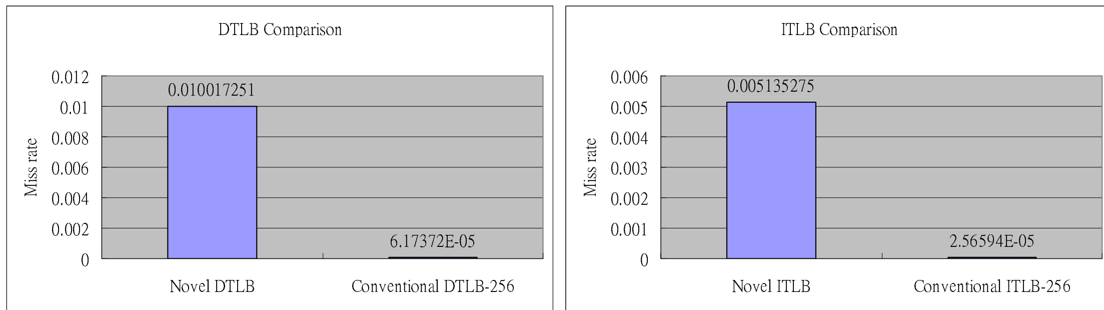
TWOLF



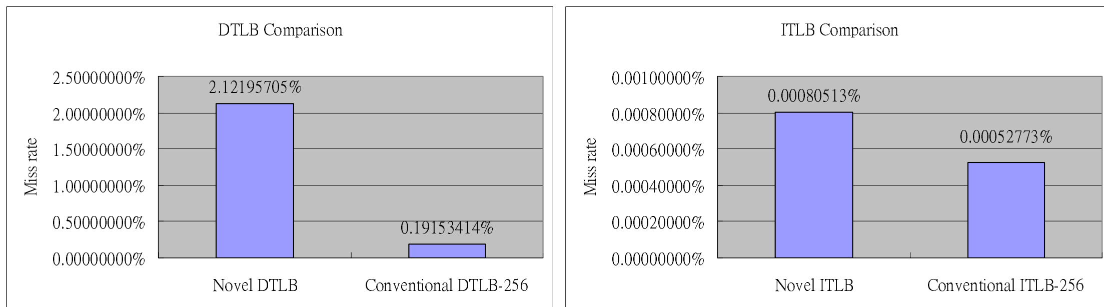
SWIM



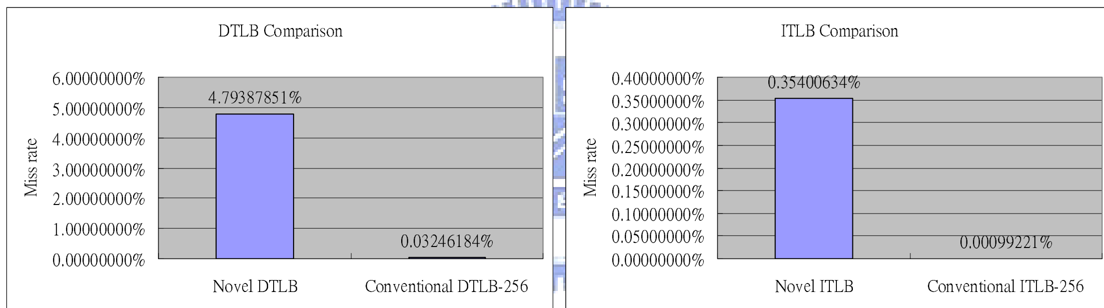
LUCAS



APPLU



EQUAKE



MGRID

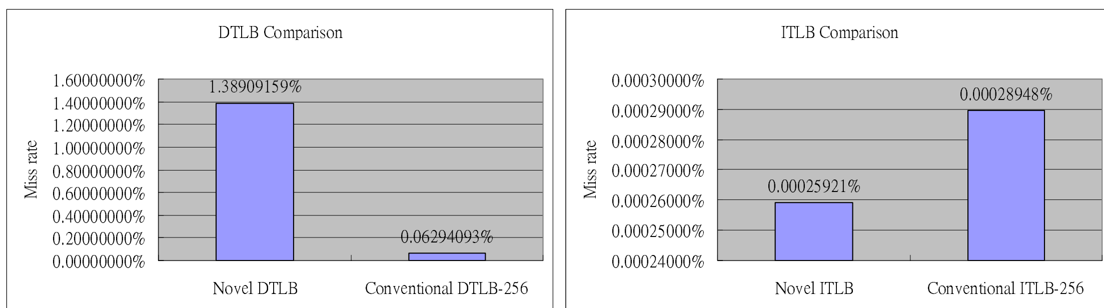


Figure 13: TLB miss rate comparison with 4KB page size.

This section presents our second novel TLB structure designed which is targeted toward using small page improving the limitation in the original design. Although using large page size can have better coverage of memory space and the translation needed to be stored can be smaller, it also waste memory due to internal

fragmentation and most modern OS only support small page size under multiprogramming environment. Our new proposed TLB is not only capable of reducing miss rate in context switches with small page size but supporting multiple page sizes. Further, the hardware cost in our new design is almost as same as the conventional TLB.

3.2.1 Overview

Our new TLB structure combines both the features of Lee's dual TLB [17] [18] and our original TLB [6] with many TLB banks. We use a shared conventional small page (4KB size) TLB and many large page (16KB size) promotion-TLB banks. The difference compared to our original design is that only the shared TLB need to be flushed when context switching. The shared TLB works together with only one of the promotion-TLB banks at a time. The TLB banks can keep from flushing when context switching. As a result, we can reduce the miss rate in context switches with small page size because the shared TLB can effectively reduce the miss rate. The remainder of this section we will present our new TLB structure, implementations and mechanisms.

3.2.2 Proposed TLB Structure

Figure 14 shows in detail our new novel TLB structure to reduce miss rate in context switch. The proposed structure can be broken down into four parts to discuss:

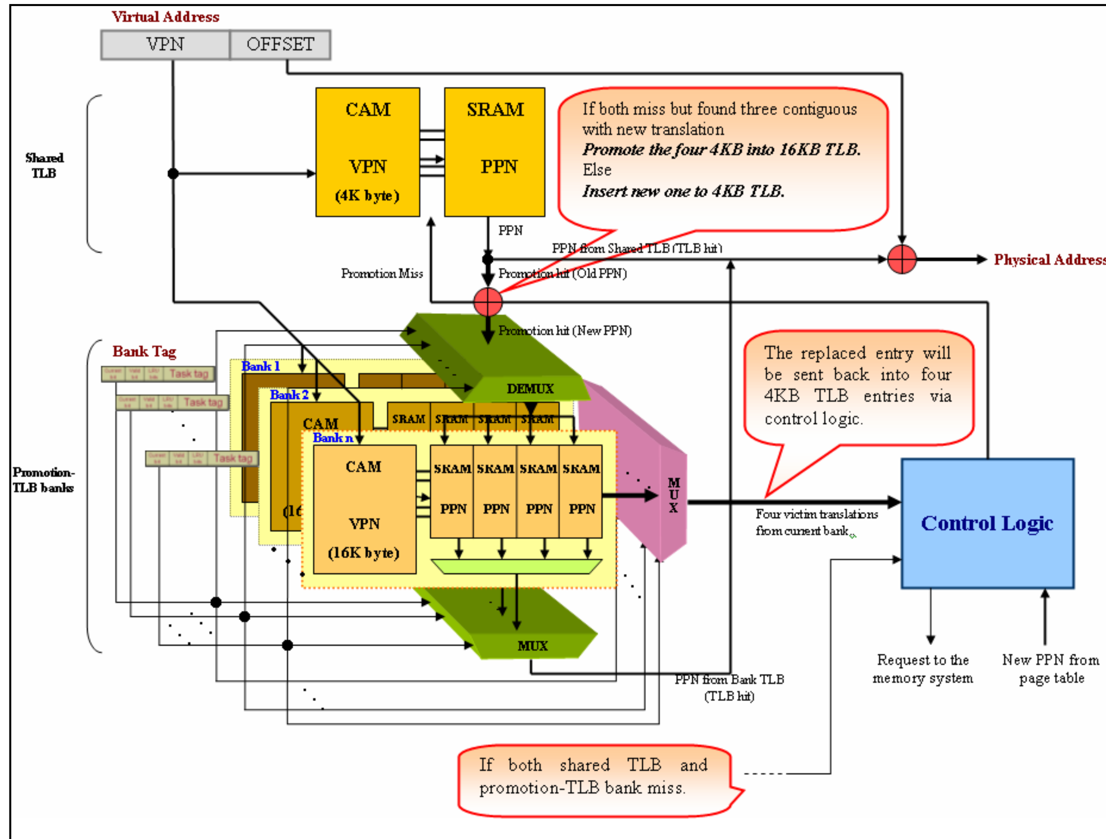


Figure 14: New proposed TLB architecture with low context switching penalty.

1. The Shared TLB and the promotion-TLB banks

Only one of the promotion-TLB banks can work together with the shared TLB at the same time. The shared TLB and all the promotion-TLB banks are designed as fully associative structures. The shared TLB is the same as that being implemented on conventional TLB while the promotion-TLB banks are all implemented as the complete-subblock TLBs.

The shared TLB is constructed as a set of m page entries, where the page size is 4KB in this example. However, each promotion-TLB bank consists of n large sized

page entries, i.e., 16KB. Thus the total number of 4KB page entries is $m+n \times (16KB/4KB) \times$ *the number of banks* in this example.

In our novel TLB structure we assume that the ***m*** is 128, ***n*** is 2 and total 4KB page entries is 256. In other words, our structure has 16 promotion-TLB banks. In comparison with the conventional 256-entry TLB, the area cost does not increased very much although we add 16 bank tag registers, multiplexers and de-multiplexers. The reason is that we use complete-subblock TLB structure in our promotion-TLB banks which several based pages are managed with one TLB tag such that the total increased cost can be ignored.

In addition, it should also be added that the shared TLB can also work as the victim cache. When the least recently used entry would be evicted from the current TLB bank, the large page entry would be broken down many small page entries and then sent back to the shared TLB through control logic. We do this action while one virtual address misses in both the shared TLB and the current bank TLB. We have sufficient time to do it because fetching the translation from main memory needs more time.

2. Control logic

The control logic has two functions. The one is that it requests the memory system for the translation when both shared TLB and current bank TLB miss. Then, the control logic will send the translation to shared TLB when getting it. The other is that it sends the evicted entry from current bank TLB back to the shared TLB.

3. Multiplexers and de-multiplexers

The multiplexers and de-multiplexers are used to select right current bank TLB. The select signal is from the current bit of the bank tag register.

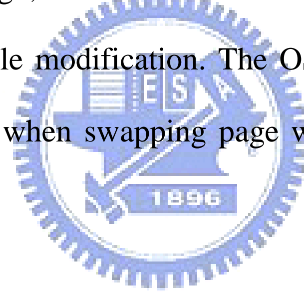
4. Bank tag registers

Each bank TLB has its own bank tag register. The behavior of the bank tag registers are the same as of our original TLB structure as described in Section 3.1. It consists of four parts:

- *Task tag*: Identify each task.
- *Current bit*: Identify current working task.
- *Valid bit*: Validate/Invalidate a bank.
- *LRU bits*: Replace the victim bank.

3.2.3 Implementation of the novel TLB

As our original TLB design, in order to realize our new proposed mechanism, the OS is just needed to do a little modification. The OS needs to send 'the clear TLB signal' to the processor only when swapping page with disk occurs or page frames release.



3.2.4 Mechanisms of the novel TLB

The mechanisms of the novel TLB can be divided into four situations to consider:

1) Task matching in one of the bank tag registers.

Once the virtual address is generated from the CPU, the virtual page number is sent to the shared TLB and all TLB banks at the same time. The shared TLB works the same as conventional TLB and each bank works the same as the complete-subblock TLB. In addition, the select signals are obtained from the current bit of all group tags in order to select right bank. The possible three cases are:

(Here we assume that the shared TLB is 4KB page size and the promotion-TLB banks

are all 16KB page size in this example.)

- **Hit in the shared TLB.**

If the VPN is found in the shared page TLB, the actions are the same as any conventional TLB hit. Then, the requested physical address would be sent to the cache.

- **Hit in a bank TLB which is current working.**

If a hit occurs at a bank TLB which is current bank, only one of four PPNs in the bank TLB is enabled at the same time. The actions are the same as any complete-subblock TLB hit. Then, as in the case of the shared TLB hit, the requested physical address would be sent to the cache.

- **Miss in both places.**

When one virtual address misses in both the shared TLB and the current bank TLB, O/S perform miss handling. When a TLB miss occurs and if its corresponding three sequential VPNs exist in the shared TLB, those four sequential VPNs belonging to a 16KB page boundary are chosen to be promoted. The 16KB page is stored into the current bank TLB as a new single entry. And also the three sequential VPNs in the shared TLB are invalidated at the same time, causing to increase the effective entry space in the shared TLB.

What has to be noticed is that we did some difference from Lee et al. When the space of the current working bank is not enough, the least recently used entry has to be discarded in Lee et al. But we divide the large page into four small page entries and send back to the shared TLB through control logic to reuse them. The reason is that the bank TLB usually have higher hit rate than the shared TLB and we can get better performance from the additional action than Lee's dual TLBs.

2) No task tag matching in any bank tag register.

The situation happens only when the first instruction fetching after a context switching for ITLB, the system initialization, or the swapping pages with disk occurring. Under this situation, no valid physical address can be provided via TLB translation from the shared TLB or any bank TLB. The address should be generated in general way by the MMU and OS. After the physical address (or PID if it is available) is generated, it is compared with the task tag field of bank tags. If any of it is hit in a valid bank tag, the current bit of that bank tag is set. Otherwise, the MMU should try to select a victim bank with invalid bit and LRU bits from the bank tag and flush all content of the victim bank. Then the current bit of this bank should be set and LRU bits of all bank tags should be updated. Finally, the correct translation is stored into the shared TLB entry and the task tag of current working bank should be set. Moreover, it is the generated PPN (or PID under the situation which PID is available) that is stored into the task tag field of the current bank tag.

3) Context switching

Once the context switching happens, the MMU just need to flush the shared TLB and the current bit of the bank tag.

4) Page swapping with disk occurs or page frame releases.

If the page swapping with disk occurs or page frame releases, the modified OS sends the 'clear TLB signal' to flush the MMU. Hence, the MMU will flush the shared TLB and clear the valid bit of all bank tags.

Chapter 4 Simulation Results

In this chapter, we will investigate the performance of our new novel TLB architecture with low context switch penalty. We compare the miss rates in our architecture with other TLB architectures, such as conventional TLB, complete-subblock TLB, and Lee's TLB. First, we will introduce the SimpleScalar tool set in Section 4.1. Secondly, we will describe our experimental methodology and benchmarks in Section 4.2. Lastly, we will show some simulation results of comparing with other TLB architectures in Section 4.3.

4.1 Introduction to the SimpleScalar Tool Set

The SimpleScalar tool set is a suite of powerful computer simulation tools that provide both detailed and high-performance simulation of modern microprocessors. In addition, the SimpleScalar tools have the advantages of high flexibility, portability, extensibility and performance. In this Section we will briefly introduce the tool set, the architecture and the instruction set architecture (ISA) of the SimpleScalar.

4.1.1 Overview

SimpleScalar was created by Todd Austin [2] [21]. Development began while he was a Ph.D. student at the University of Wisconsin in Madison. Early versions of the tool set included contributions by Doug Burger and Guri Sohi. Today, SimpleScalar is developed and supported by SimpleScalar LLC.

The SimpleScalar tool set is a system software infrastructure used to build modeling applications for program performance analysis, detailed micro-architectural modeling, and hardware-software co-verification. We can build modeling applications that simulate real programs running on a range of modern processors and systems. Figure 15 below shows the SimpleScalar simulator structure.

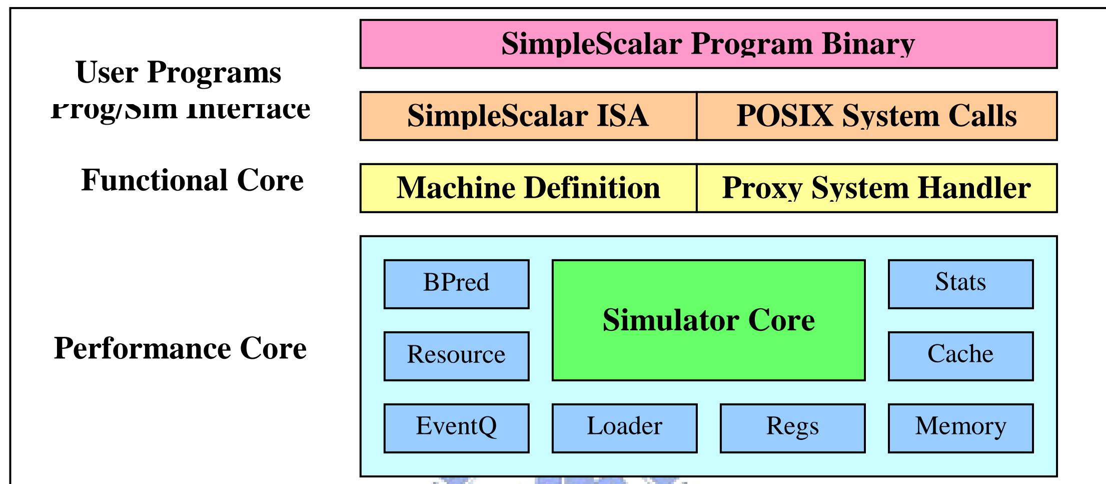


Figure 15: Simulator Structure.

SimpleScalar simulators can emulate the Alpha, PISA, ARM, and x86 instruction sets. The tool set includes a machine definition infrastructure that permits most architectural details to be separated from simulator implementations. The tools can be built on a wide range of host platforms, including Linux/x86, Win2000, SPARC Solaris, and others.

The tool set includes sample simulators ranging from a fast functional simulator to a detailed, dynamically scheduled processor model that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction. In addition to simulators, the SimpleScalar tool set includes performance visualization tools, statistical analysis resources, and debug and verification infrastructure.

In this thesis, we use the *sim-cache* cache simulator to simulate our novel TLB design because we just care about the miss rate in TLB performance.

4.1.2 Instruction Set Architecture

The SimpleScalar instruction set architecture is derived from the MIPS-IV ISA. The SimpleScalar ISA defines bi-endian instruction set to facilitate portability. The semantics of the SimpleScalar ISA are a superset of MIPS with the following notable difference and additions:

- *There are no architected delay slots.* Load, store, and control transfers do not execute the succeeding instruction.
- *Loads and stores support two addressing modes.* These are indexed (register + register), and auto-increment/decrement.
- *A square-root instruction, which implements both single-precision and double-precision floating point square roots.*
- *An extended 64-bit instruction encoding.*

We can classify all SimpleScalar instructions into four main groups:

- *Control*
- *Load/Store*
- *Integer Arithmetic*
- *Floating Arithmetic*

In figure 16, we depict the three instruction encodings of SimpleScalar instructions: *register*, *immediate*, and *jump* formats. All instructions are 64 bits in length.

- The *register* format is used for computational instructions.
- The *immediate* format supports the inclusion of 16-bit constant.
- The *jump* format supports specification of 26-bit jump target.

The register fields are all 8 bits, to support extension of architected registers to 256 integer and floating point registers. Each instruction format has a fixed-location, 16-bit opcode field to facilitate fast instruction decoding. The annote field is useful when synthesizing new instructions without having to change and recompile the assembler.

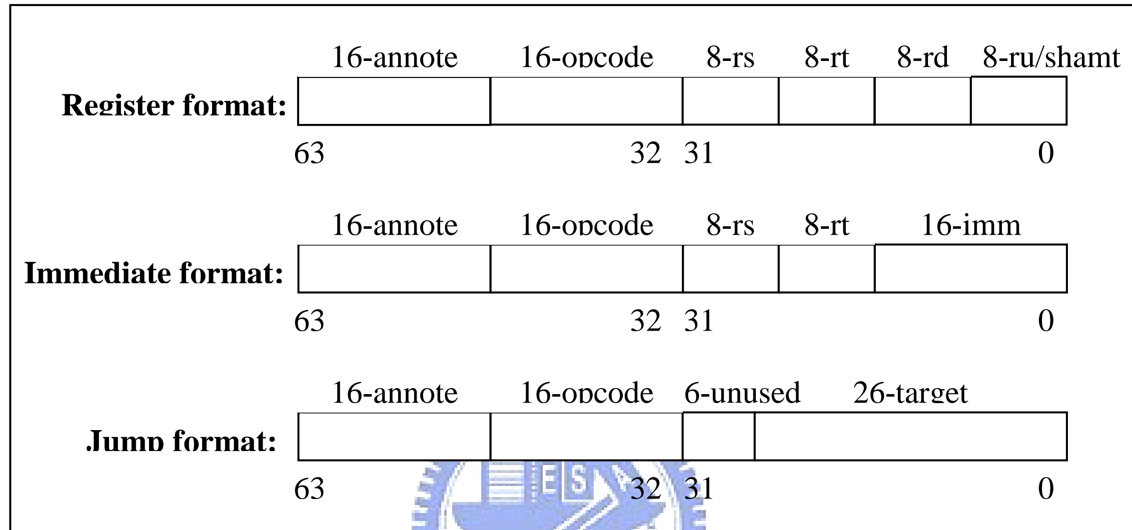


Figure 16: SimpleScalar architecture instruction formats.

4.2 Experimental Methodology

In order to simulate the performance of our novel TLB architecture and other TLB design, we have implemented them using the SimpleScalar 3.0d Simulator, a open source software that emulates the Alpha, PISA, ARM and x86 instruction sets. We have modified certain modules of the SimpleScalar Simulator code to implement our and other TLB mechanisms. Specifically, we have modified the modules that implement the cache. Here, we briefly discuss how TLBs are implemented in the SimpleScalar Simulator in Section 4.1.1, and later we will discuss the modifications in order to implement our required TLB in Section 4.1.2.

4.2.1 TLB Implementation in SimpleScalar Simulator

1. The TLB is implemented as a cache; all modules of cache are also applicable to TLB.

2. Highly associative caches are maintained as hash-table, with pointers to maintain LRU stack within each set.

3. Three replacement policies are supported by the cache module:

   a) Least-recently-used replacement policy.

   b) First-in-first-out replacement policy.

   c) Random replacement policy.

4. The function *cache_create()* is used to create an instance of cache. The arguments to this function are cache parameters such as the cache name, cache size, block size, associativity, hit latency etc.

5. The function *cache_access()* simulates the accesses to the cache and returns the latency of the operation.

6. The function *sim-cache()* is used to start simulation, program loaded, processor precise state initialized.

7. The structure *cache_t* define the properties of the cache such as cache name, number of sets, block size in bytes, maintain cache contents, user allocated data size, cache associativity, cache replacement policy, and cache hit latency etc.

4.2.2 Modifications to implement our novel TLB structure

1. The *cache_access()* function is modified to support the complete-subblock TLB mechanisms, promotion mechanisms and dual TLB in our architecture.
2. We add the mechanism of flushing all TLB when context switching in the *sim-cache()* function to simulate the OS operation.
3. In order to implement our novel TLB structure, we add some extra properties and structure in the structure *cache_t*.

4.2.3 Benchmarks

We will use seven SPECint2000 benchmarks and five SPECfp2000 benchmarks in our simulations to study our novel TLB structure. All the simulations are conducted using the SimplesSalar-3.0 toolset with Alpha processor architecture and we use sim-cache component of this toolset to simulate our TLB performance. These benchmarks and their short description are listed in Table 1 and Table 2, respectively.

BENCHMARK	INPUT CLASS	INPUT FILE	#INSTRUCTIONS (DTLB)	#INSTRUCTIONS (ITLB)
GZIP	train	input.combined	214195197	683485209
VPR	test	net.in, arch.in,	566450029	1566705222
GCC	test	print-tree.i	304674907	662772323
CRAFTY	ref	crafty.in	71181816	195369780
PERLBMK	test	test.pl	2155769	5230077
VORTEX	ref	lendian2.raw	423625	693986

TWOLF	test	Test.in	88200375	258755188
SWIM	test	swim.in	137986483	431489758
LUCAS	test	lucas2.in	25780475	79439485
APPLU	test	applu2.in	1361697211	3604281628
EQUAKE	test	inp.in	551866489	1443347875
MGRID	test	mgrid2.in	1276109289	3488289746

Table 1: Summary of the simulated SPEC2000 benchmarks along with the input sets.

BENCHMARK	LANGUAGE	TYPE	CATEGORY
GZIP	C	Integer	Compression
VPR	C	Integer	FPGA Circuit Placement and Routing
GCC	C	Integer	C Compiler
CRAFTY	C	Integer	Game Playing: Chess
PERLBMK	C	Integer	PERL Programming Language
VORTEX	C	Integer	Object-oriented Database
TWOLF	C	Integer	Place and Route Simulator (CAE)
SWIM	Fortran90	Floating-Point	Shallow Water Modeling
LUCAS	Fortran90	Floating-Point	Number Theory / Primality Testing
APPLU	Fortran77	Floating-Point	Parabolic/Elliptic Partial Diff. Eqns
EQUAKE	C	Floating-Point	Seismic Wave Propagation
MGRID	Fortran77	Floating-Point	Multi-grid Solver: 3D Potential Field

Table 2: SPEC 2000 benchmarks description.

4.3 Simulation Results

In this Section we will make a comparison between the four TLB structures:

(Below we define a variable ***PS*** which means the *size of a page*.)

1. Conventional fully associative TLB.

We assume the conventional TLB has 256 entries.

2. *Complete-subblock TLB* (as described in Section2.2).

We assume the subblock factor is 4. It means the complete-subblock TLB has 64 entries of **$4 \times PS$** page size.

3. *Lee's TLB* (as described in Section2.2).

Lee et al. observed that when the mapping size is larger than 256KB mapping size, a combinational of 4KB small and 16KB large TLB can shows the best performance. So, We will use the combination of ***PS*** small page TLB with 128 entries and **$4 \times PS$** large page TLB with 32 entries to make a comparison.

4. Our new novel TLB.

We did several experiments to determine the optimal number of pages to promote in our new TLB structure and choose the combination of ***PS*** page size in shared TLB with 128 entries and **$4 \times PS$** page size in promotion-TLB banks with 2 entries per bank. We assume our structure can support to 16 banks if we have 256 ***PS*** small page compared to conventional TLB.

In our simulation, we change the variable ***PS*** from 4KB to 16KB to make a comparison. In Table 4.3 we show that memory mapping size comparison between these four TLB structures. We will show that we can still deliver better performance in our structure, and even our design has lest memory mapping size than others.

TLB structures	TLB entry size	Memory mapping size
Conventional TLB	256 entries	$256 \times$ page size
Complete-subblock TLB	256 entries (64 blocks x 4 Subblocks)	$64 \times 4 \times$ page size
Lee et al's dual TLB	256 entries (128 entries + 32 blocks x 4 Subblocks)	$(128 + \mathbf{32} \times 4) \times$ page size
Our novel TLB	256 entries (128 entries + 2 blocks x 4 Subblocks x 16 banks)	$(128 + \mathbf{2} \times 4) \times$ page size

Table 3: Memory space coverage comparison of four TLBs.

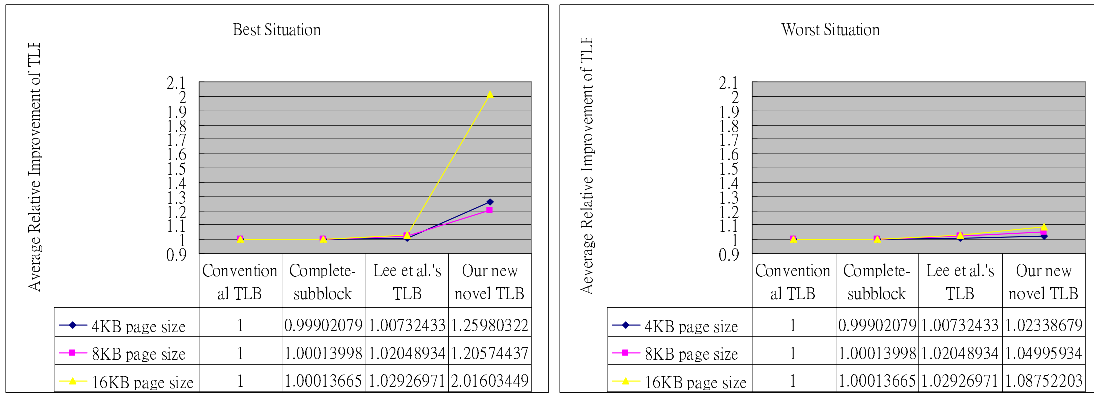
In order to simplify the OS behavior of causing context switching in our novel TLB structure, we just only consider two situations. The one is the *best situation* that assumes our structure can completely preserve some benchmark's translation from flushing during any context switching and the other is the *worst situation* that assumes our TLB must be flushed after executing one million instructions as conventional TLB. We will show that we can still get best performance even in the worst situation. By the way, we only consider the worst situation for the other three TLB structures because they don't consider the context switching problem. We still assume that the context switching would happen after executing one million instructions.

TLB performance will be shown in Figure 17 for the twelve SPEC2000 benchmarks. In our simulation we define the relative improvement of miss rate for all TLBs with respect to the conventional TLB. The relative improvement of miss rate is given by:

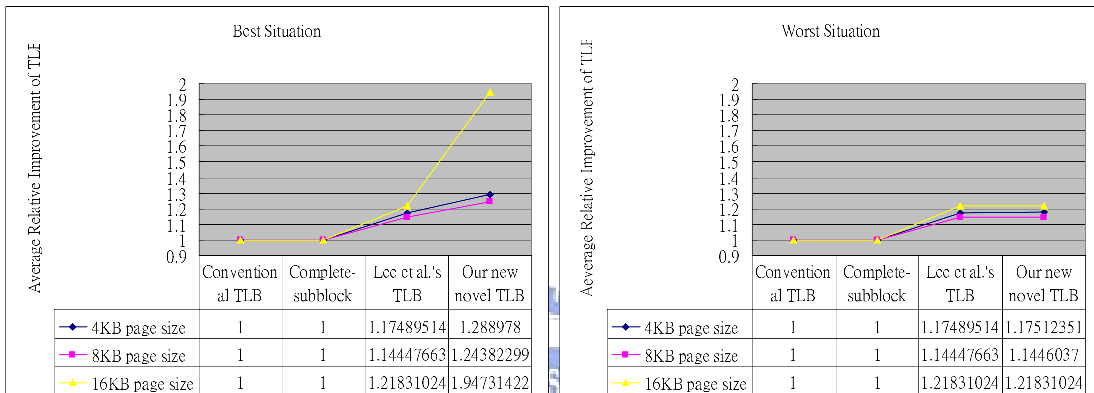
$$\text{Relative improvement of miss rate} = \frac{\text{The miss rate of conventional TLB}}{\text{The miss rate of other TLBs}}$$

The relative improvement of miss rate in conventional TLB is always 1. In the other words, the higher the relative improvement of miss rate is, the better the performance is. In the Figure 17 we average the relative improvement of miss rate for ITLB and DTLB and the Appendix A shows more detail analysis for DTLB and ITLB.

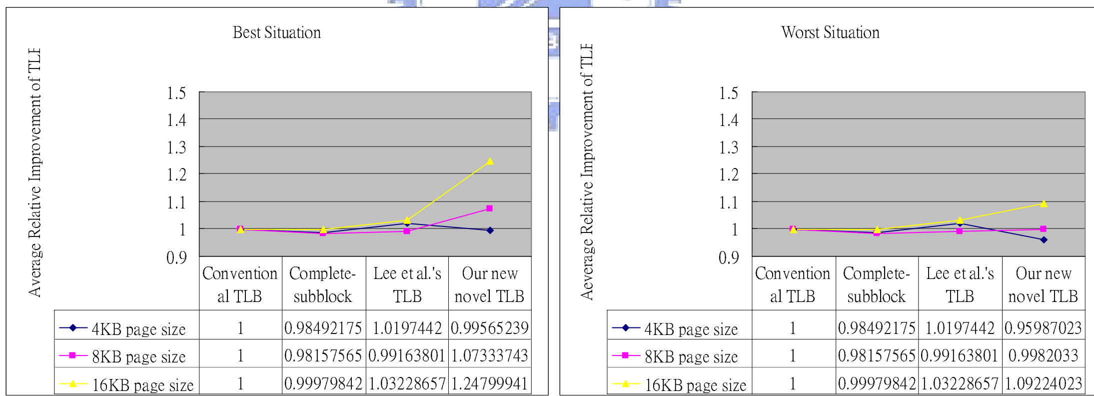
gzip



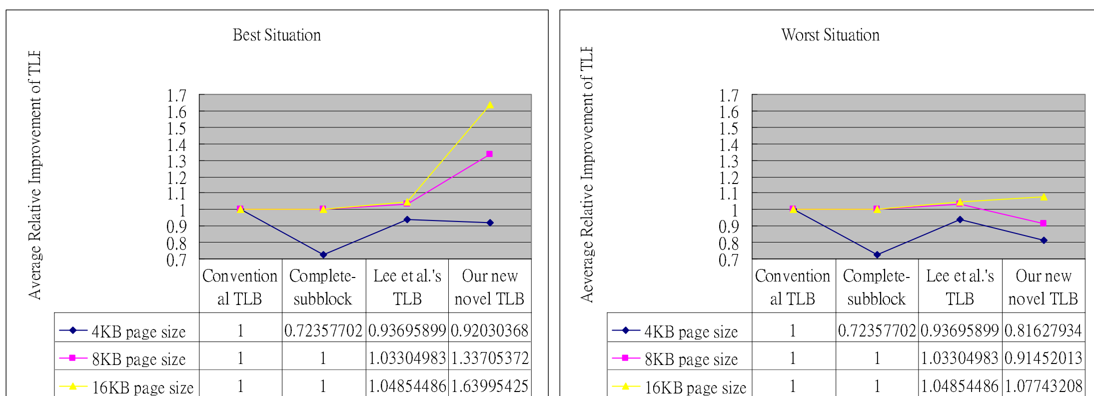
vpr



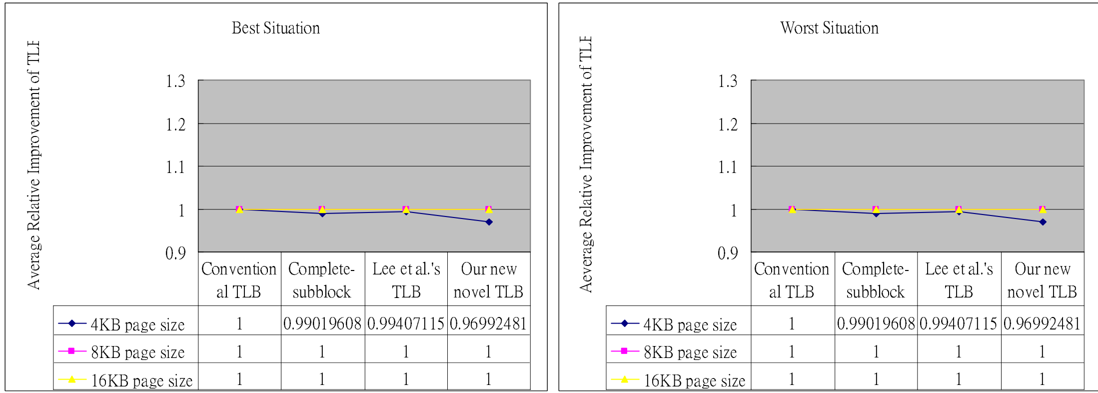
gcc



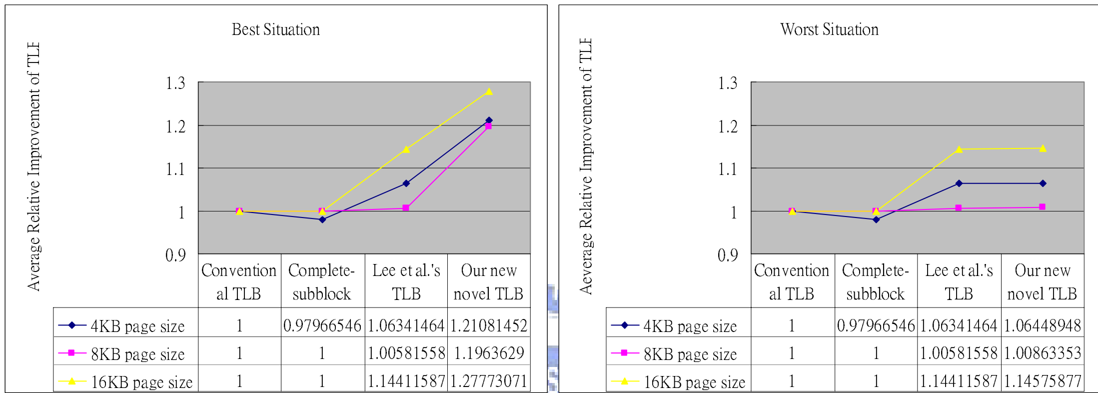
crafty



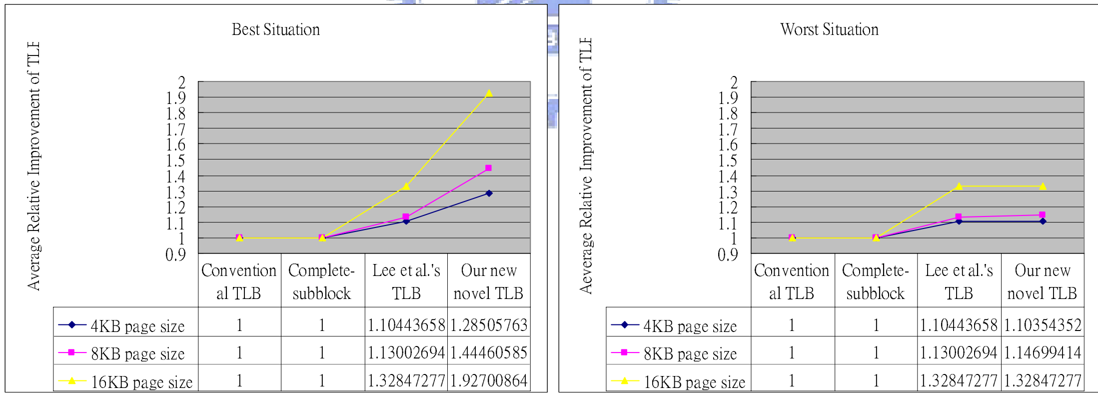
vortex



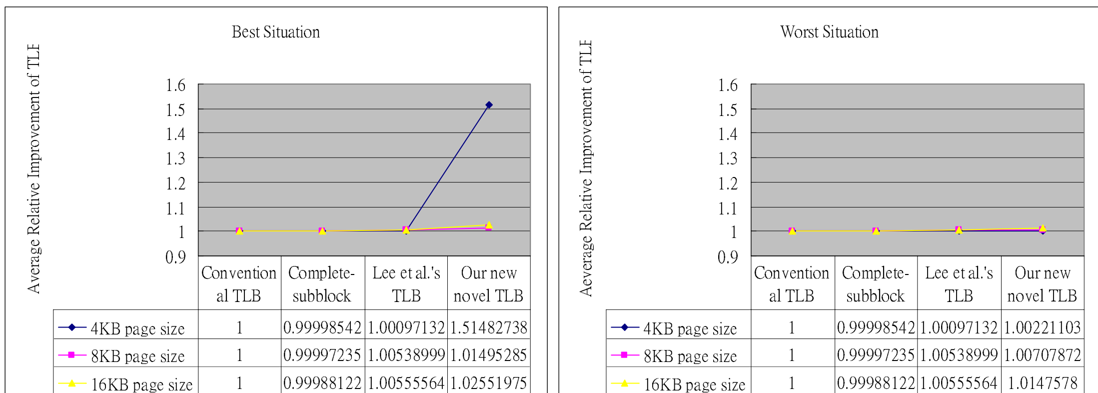
lucas



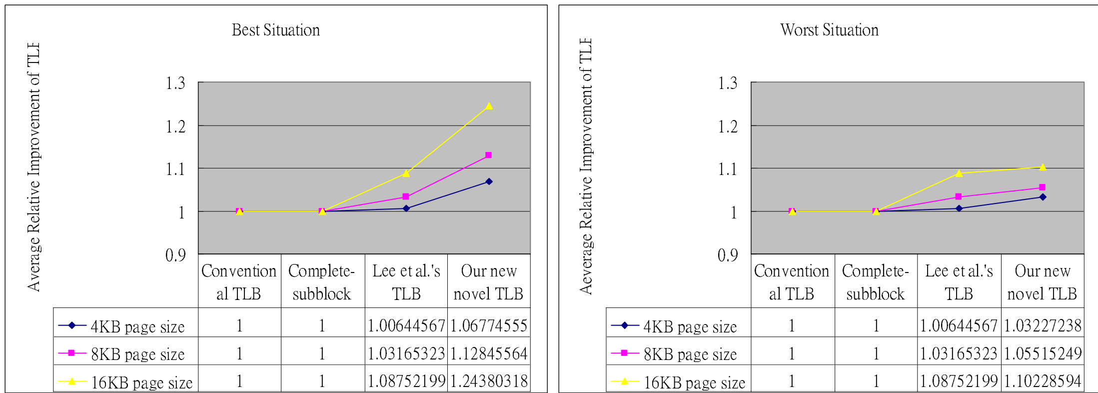
twolf



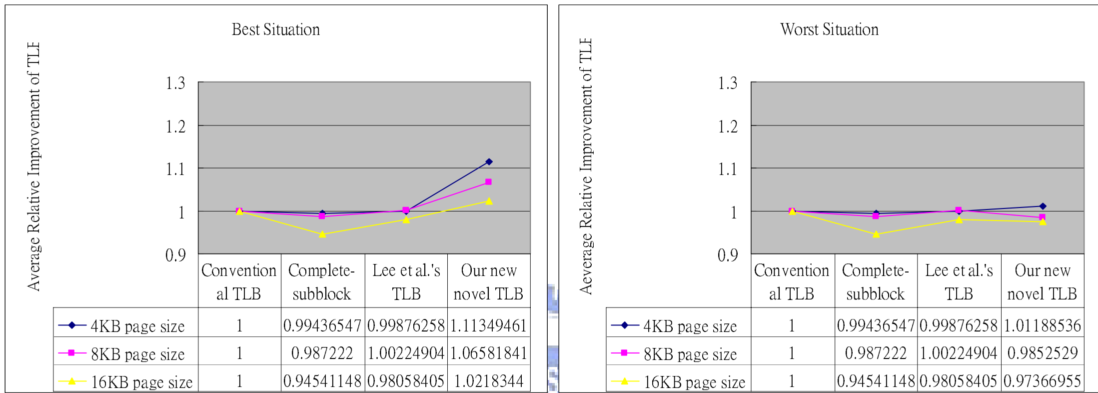
swim



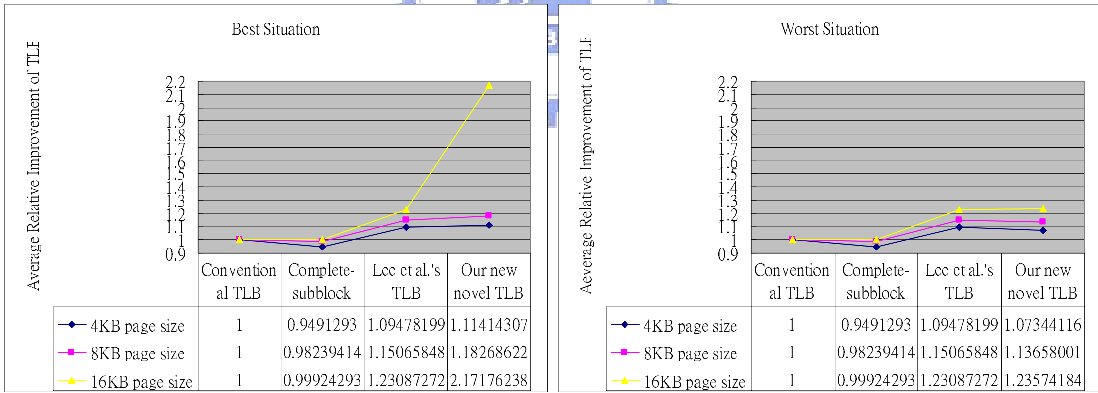
perlbmk



applu



equake



mgrid

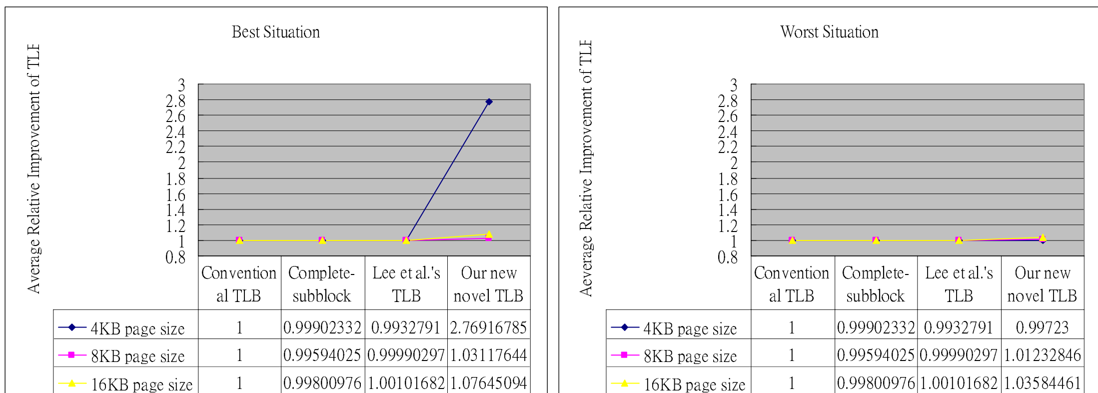


Figure 17: The comparison of relative improvement with different page sizes.

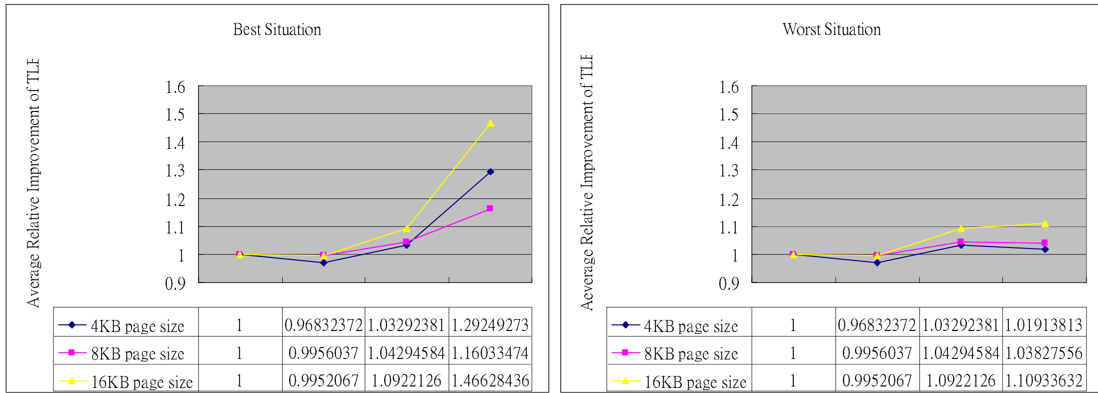


Figure 18: The mean of the average relative improvement for the twelve SPEC2000 benchmarks.

As can be seen in Figure 17, Figure 18 and Appendix A, we can find that:

- For the *gzip*, *vpr*, *lucas*, *twolf*, *perlbmk, swim* and *mgrid* benchmarks, our design can deliver better performance than the other three TLB structures with all small page sizes, even our structure is in the worst situation.

- Because the *gcc*, *applu*, *equake* and *crafty* benchmarks have larger working set than other benchmarks, the relative improvement of our structure may be worse than the other TLBs, especially for the DTLB, when the page size is 4KB or 8KB. However, we should point out that the situation can be easily solved with more entries or larger page sizes.

- The *vortex* benchmark has the shortest computation time than the other benchmarks. The computation time can be finished before the first context switching. To put it plainly, the relative improvement does not increase when we increase the page size.

- For the *swim* benchmark, we only lose the Lee's TLB but we can still defeat the conventional and complete-subblock TLB in any situations.

- On the whole, our design can deliver the best performance when we increase the page size for all benchmarks.

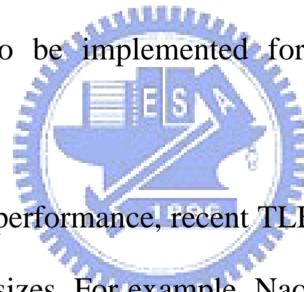
- The simulation results reveal that our design seems to have better relative improvement with 4KB page than with 8KB or 16KB page in some benchmarks, such as *gzip*, *vpr*, *lucas*, *swim*, *applu* and *mgrid*. That is because decreasing miss rate with reducing context switch penalty is more important than that with increasing page size.
- Although our design has less memory mapping size than the other three TLB structures, we can still deliver better performance in the best or worst situations overall. The reason is that our novel TLB has better utilization and we can reduce the penalty caused by the context switchings. In addition, the control logic tries to send back the victim entry from the current promotion-TLB bank to the shared TLB. That is the reason why we still have better performance in the worst situation than the others.
- We only consider the situation with total 256 entries for 4KB, 8KB and 16KB page but it's possible to provide TLB with over 512 entries on contemporary processors, such as 512 entries on latest AMD Opteron™ processor. We believe that our design can deliver better performance with 512 entries because the conventional design is not helpful for small applications.
- Taking the OS effect into account, we considered the best situation and the worst situation individually. In addition, even in the worst situation our design can still deliver better performance than the conventional ones and we believe that the processes with higher priority, such as kernel processes, could get more benefits from our design.

Chapter 5 Conclusions

The TLB misses have very enormous impact on the overall performance for the processor. In order to achieve higher performance, recent TLB designs tend to provide more entries, larger page size, or even superpage mechanism. However, very few attempts have been made for the context switching issue which causes TLB cold-start misses very seriously. In fact, in our knowledge, almost no research really seriously considers this issue. In this thesis, we presented two novel TLB mechanisms to reduce the miss rate in context switching. The one is a TLB structure suitable for large page size, such as 1MB size. The other is a novel TLB structure which reduces the miss rate in context switching with small page sizes, such as 4KB, 8KB or 16KB sizes.

Our studies of new novel TLB structure focus on how to reduce TLB size for small page size TLB. We combine both the features of Lee's dual TLB and our original TLB with many TLB banks in our new structure. We use one shared conventional small page (4KB size) TLB and 16 large page (16KB size) promotion-TLB banks. Each bank has translations for each process and only the shared TLB need to be flushed when context switching. The shared TLB works together with only one of the promotion-TLB banks simultaneously. In this structure we can reduce the miss rate in context switches with small page size by keeping the translation in each bank and with the help of the shared TLB which is sufficient to reduce the miss rate. We also proposed some mechanisms to implement the new TLB structure and how to modify OS to support it. In addition, we improve the utilization of TLB entries by making use of Lee et al.'s promotion mechanism and sending back the least recently used entry from the one current TLB bank into the shared TLB.

In order to make reasonable and fairly comparisons of our design with others and to exclude the influence of multiple parameters varying simultaneously in OS, we just consider both the best case situation and worst case situation of our design. It is shown that the relative performance improvement of miss rate of the new design is almost better or equal to that of other new proposed TLB structure or conventional fully-associative TLB structure. Our proposed TLB can even achieve about 1.3 of the relative improvement of miss rate in average with 4KB page size. In fact, if the page size is simply increase to 16 KB, the proposed TLB can achieve almost to 1.45. That means our TLB can obtain much more improvement with larger page size than other structures. Furthermore, the cost of the new design is not much higher than other new structures or conventional fully-associative TLB. Thus the suggested new TLB structure is worthy to be implemented for contemporary or future high performance processors.



In order to achieve high performance, recent TLB research tends to support large page sizes or multiple page sizes. For example, Naohiko Shimizu and Ken Takatori proposed a Linux superpage kernel for Alpha, Sparc64 and IA32 [20]. In conventional approaches, to support multiple page sizes needs lots of OS modifications. In our research, we take the advantage of complete-subblock TLB structure without OS help to support two different page sizes. However, the complete-subblock TLB needs much more hardware cost than the superpage TLB with OS help. We have already begun to find a solution to integrate the superpage mechanism with low hardware cost. Furthermore, we would integrate our proposed structure with prefetching mechanism to reduce the miss rate. We believe that still lots of works should be done in this field.

Appendix A:
Detailed Relative Improvement Figures

This appendix includes relative improvement of miss rate for twelve SPEC2000 benchmarks. The behavior of individual workloads is shown here. We can look at the comparison of simulation results for data TLB (DTLB) and instruction TLB (ITLB) with different TLB structures in more detail. Usually, the ITLB has better relative improvement of miss rate than the DTLB because the ITLB exhibits greater locality than the DTLB. These figures below from Figure 19 to Figure 30 correspond to the Figure 17, where we average the relative improvement of miss rate for DTLB and ITLB.

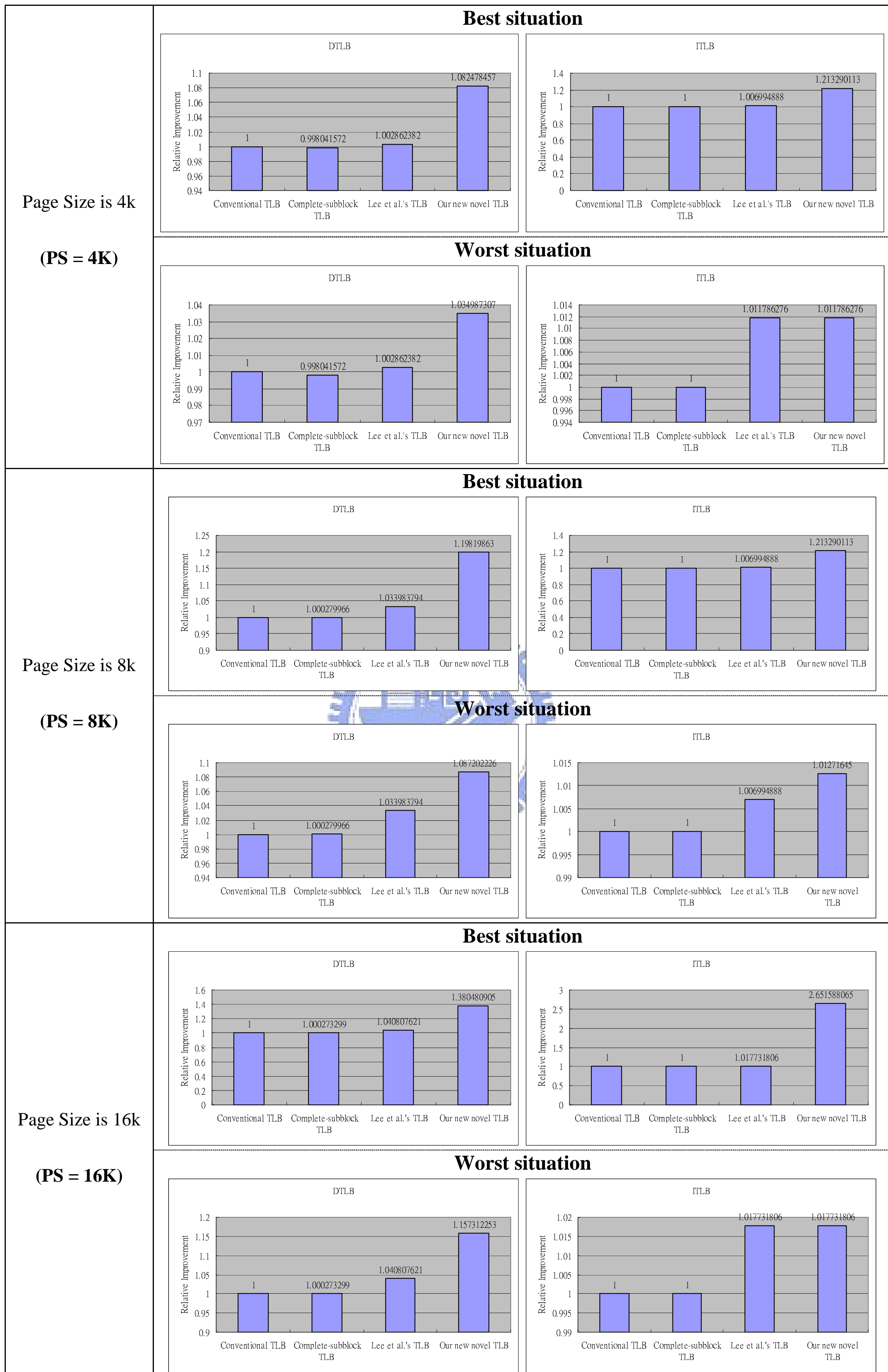


Figure 19: The relative improvement of miss rate in *gzip*.

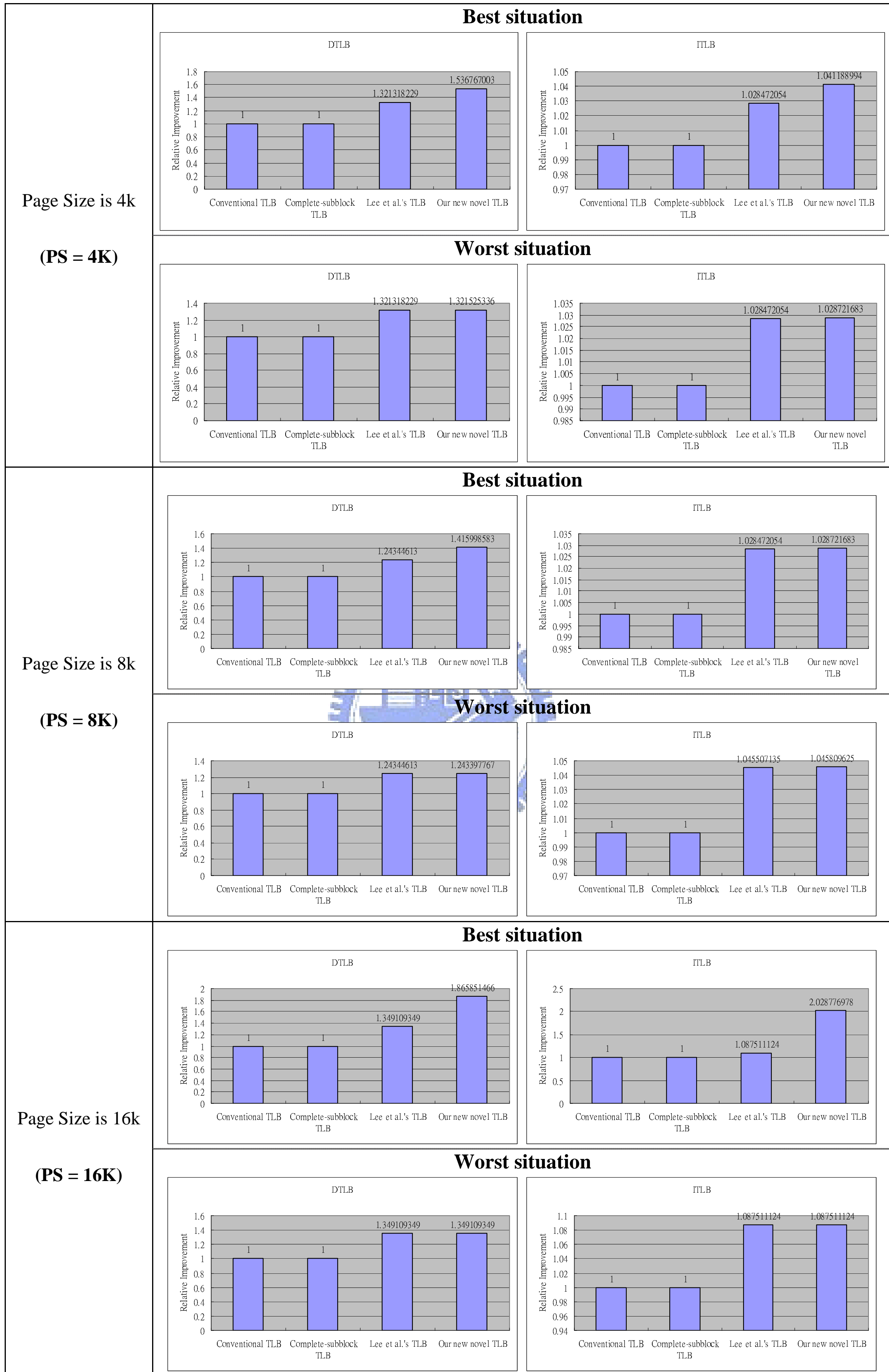


Figure 20: The relative improvement of miss rate in *vpr*.

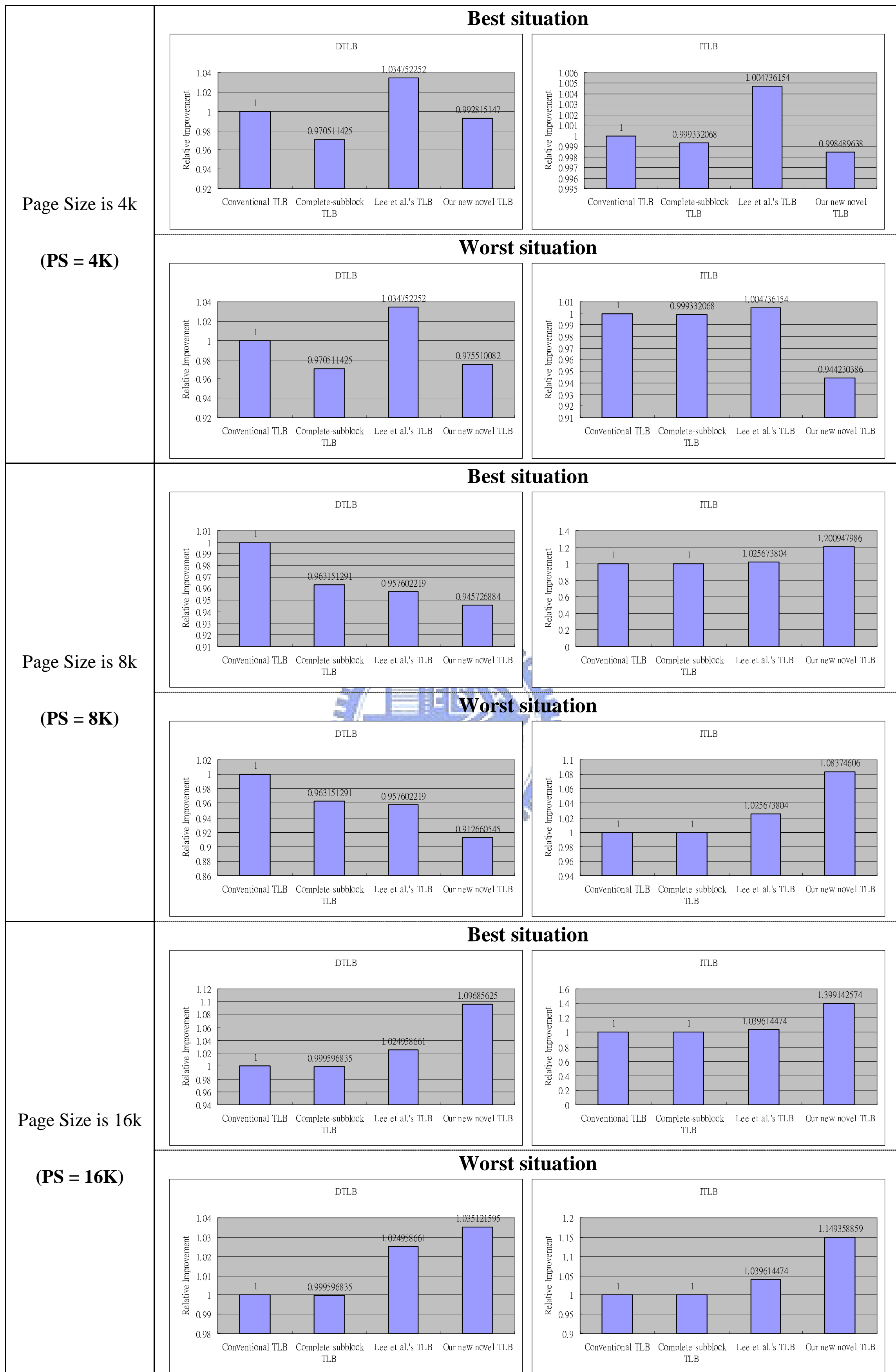


Figure 21: The relative improvement of miss rate in *gcc*.

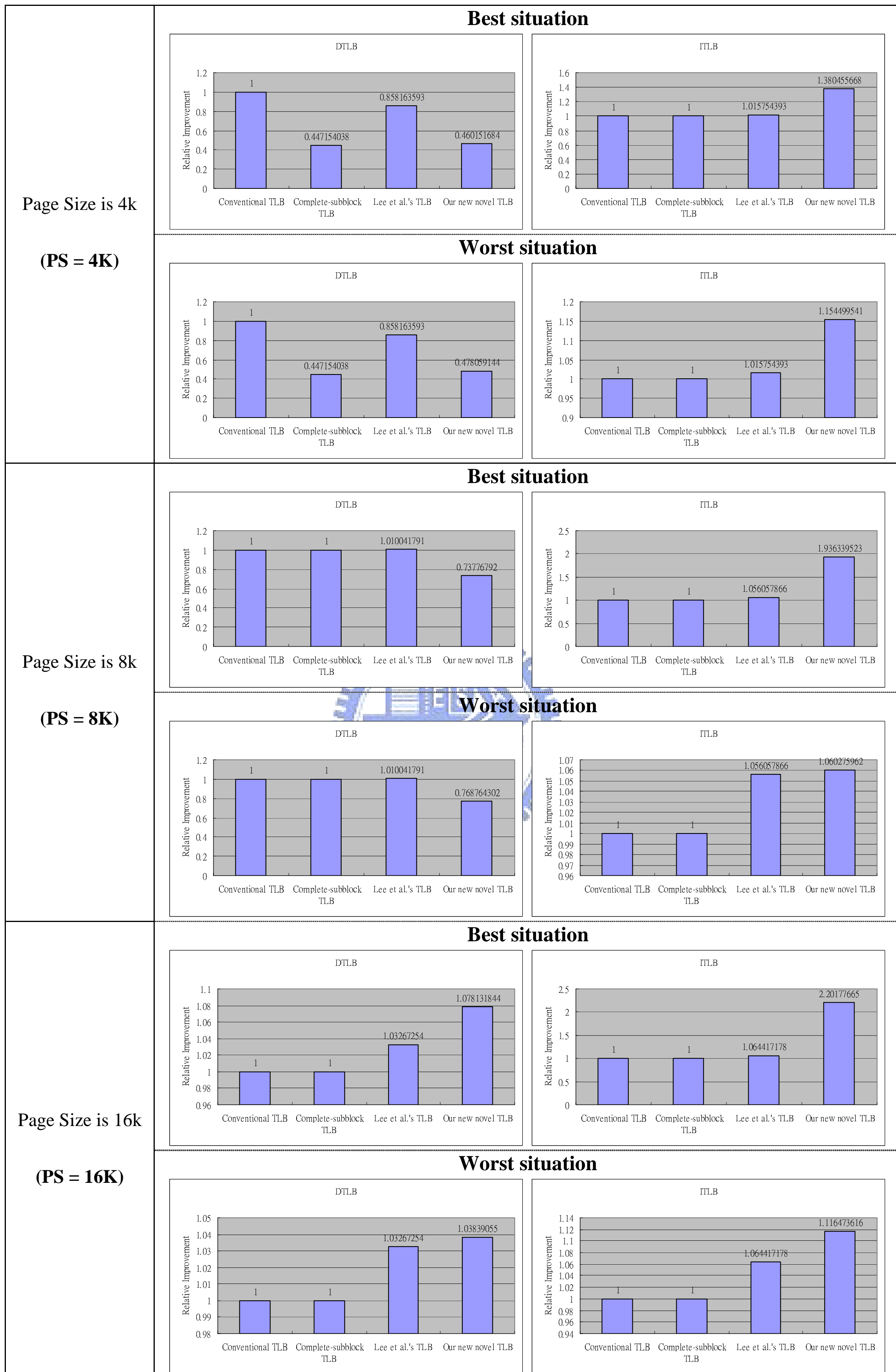


Figure 22: The relative improvement of miss rate in *crafty*.

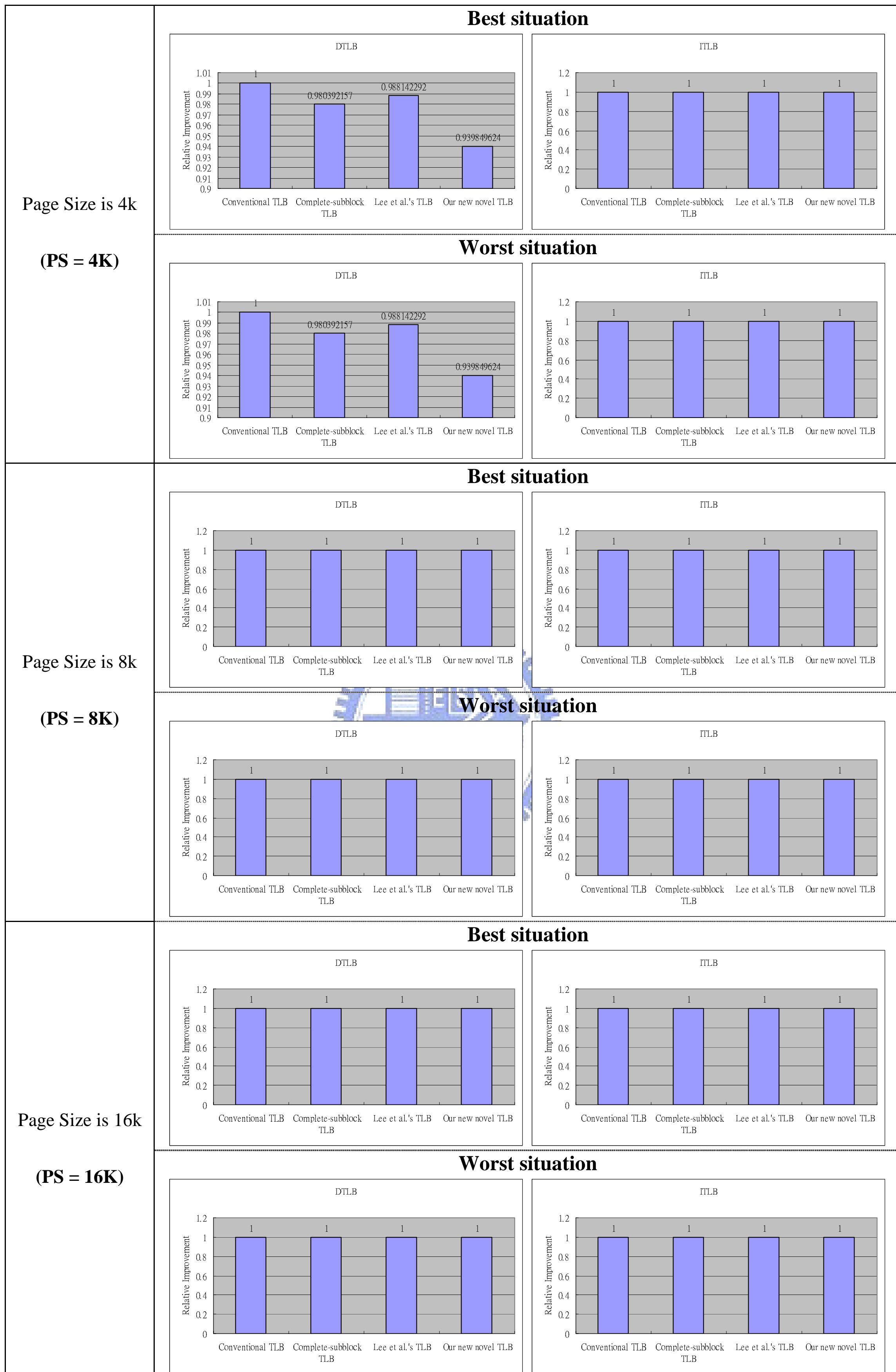


Figure 23: The relative improvement of miss rate in *vortex*.

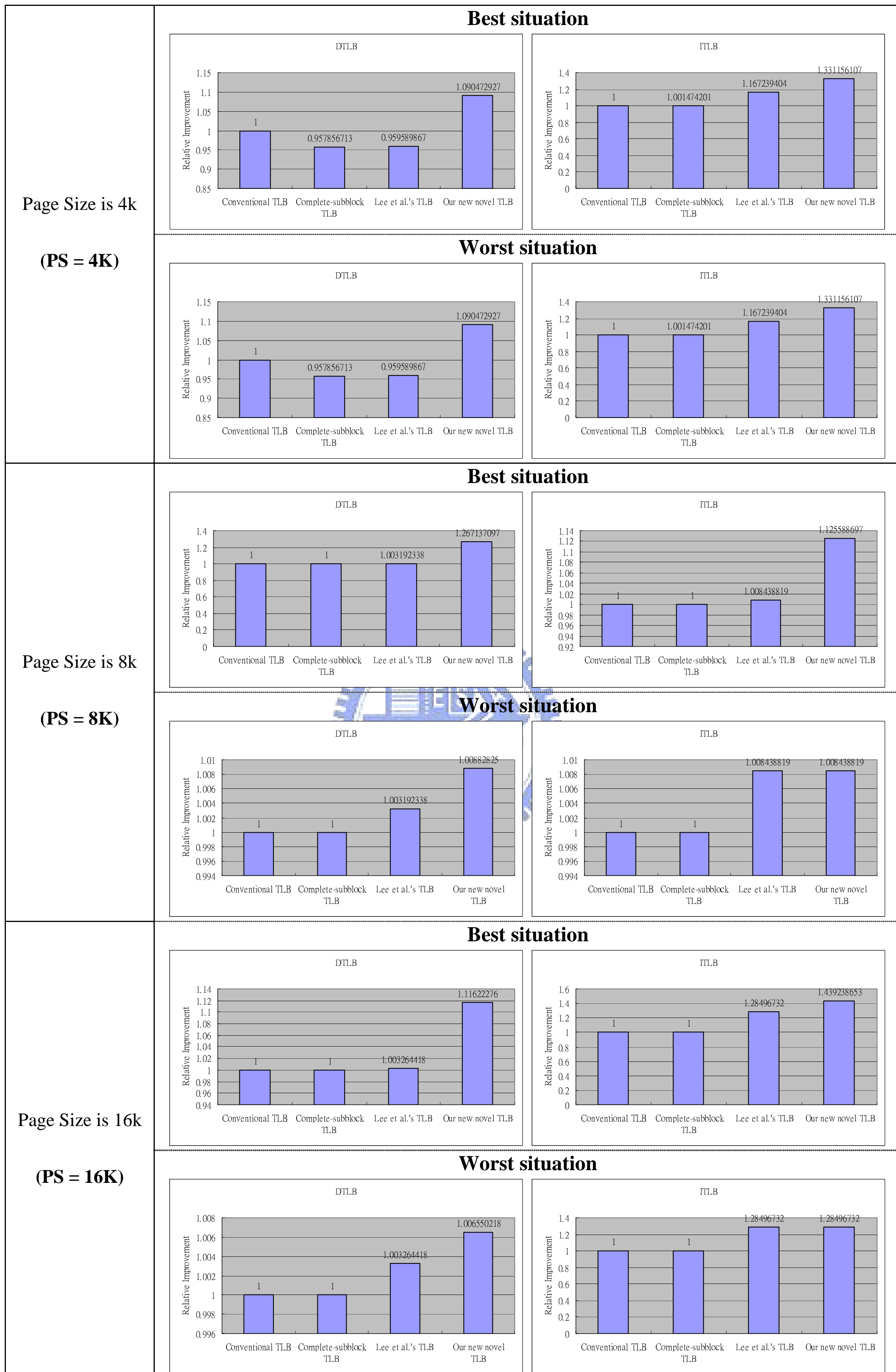


Figure 24: The relative improvement of miss rate in *lucas*.

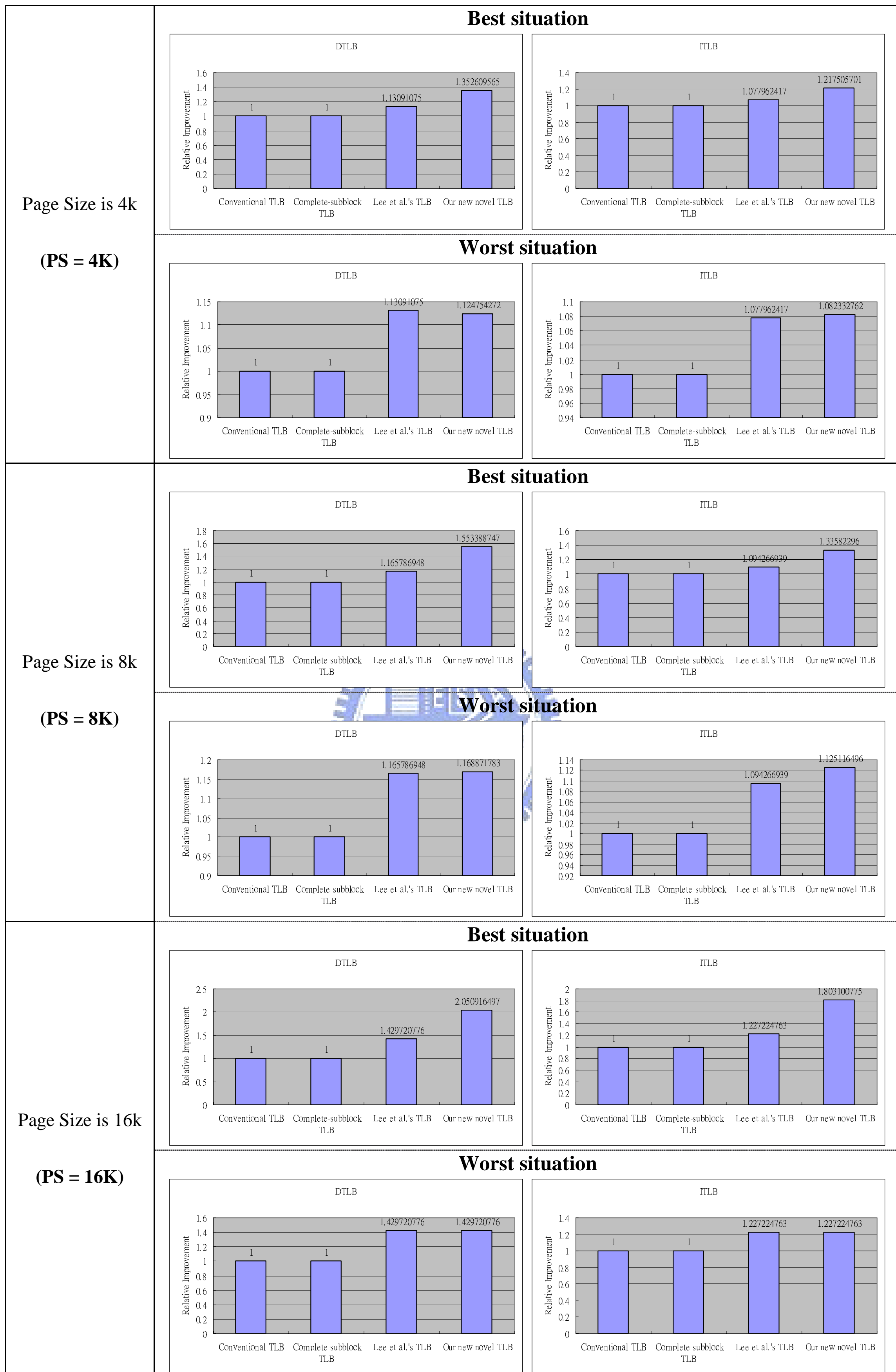


Figure 25: The relative improvement of miss rate in *twolf*.

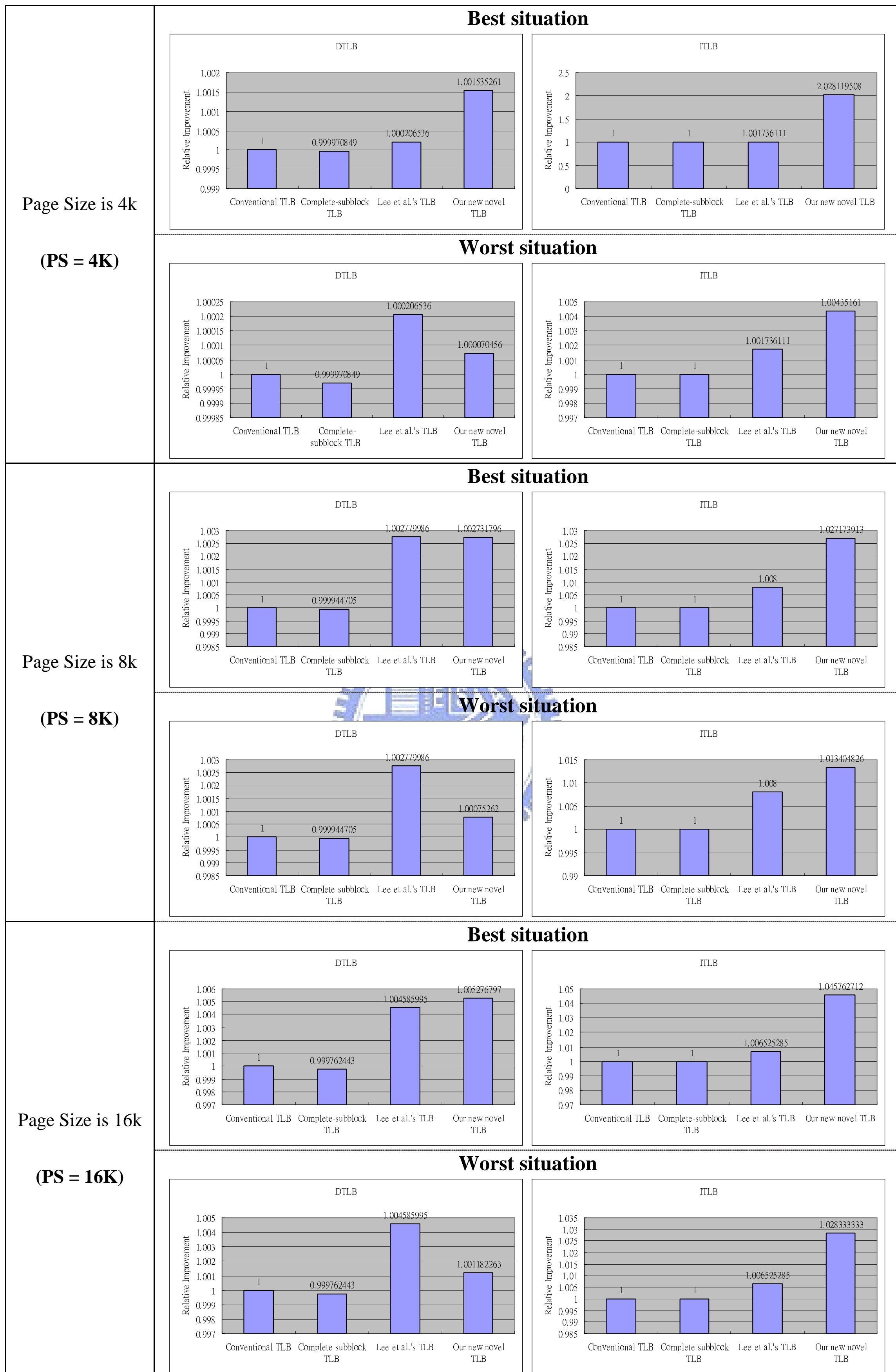


Figure 26: The relative improvement of miss rate in *swim*.

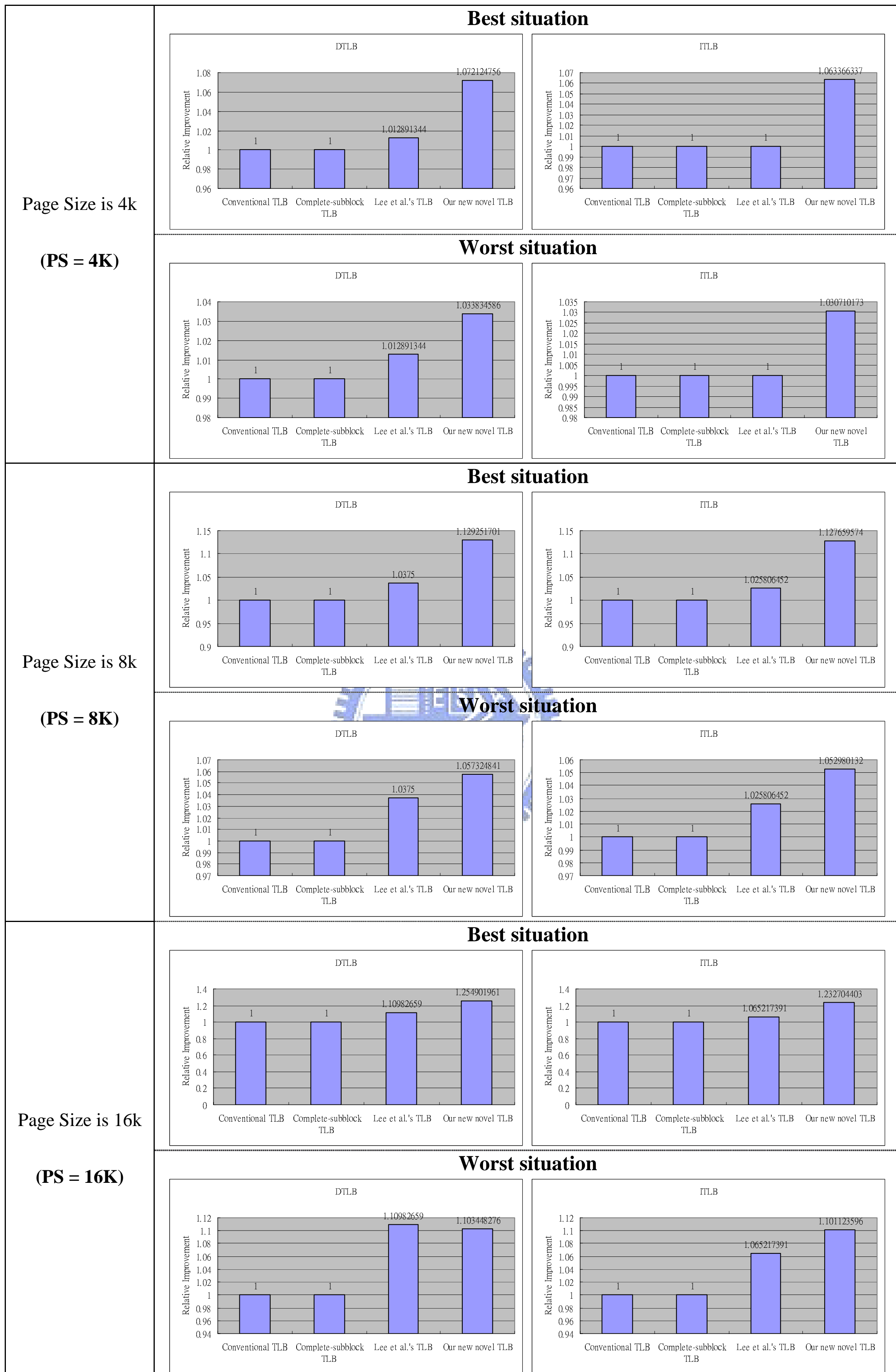


Figure 27: The relative improvement of miss rate in *perlbmk*.

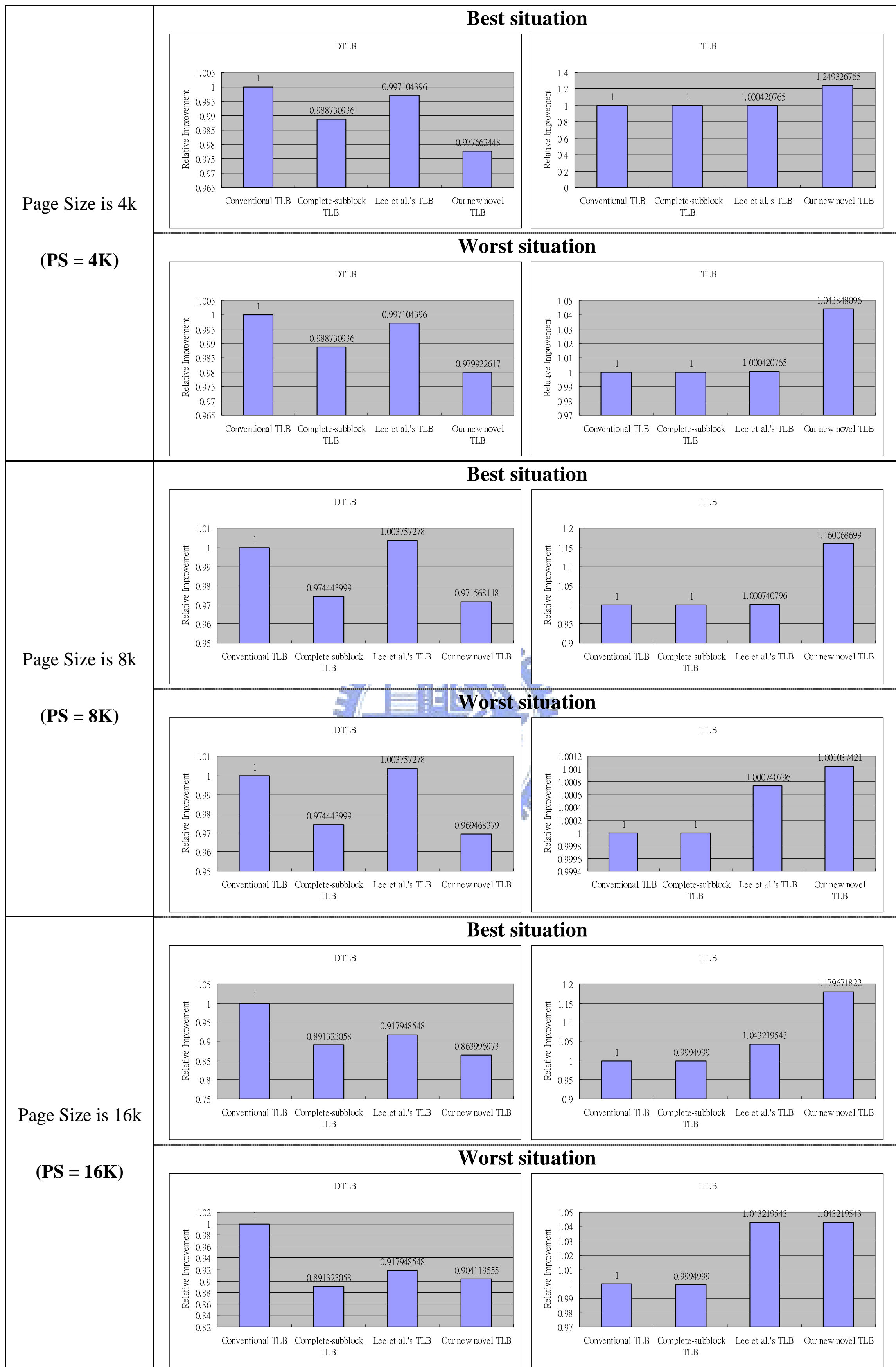


Figure 28: The relative improvement of miss rate in *applu*.

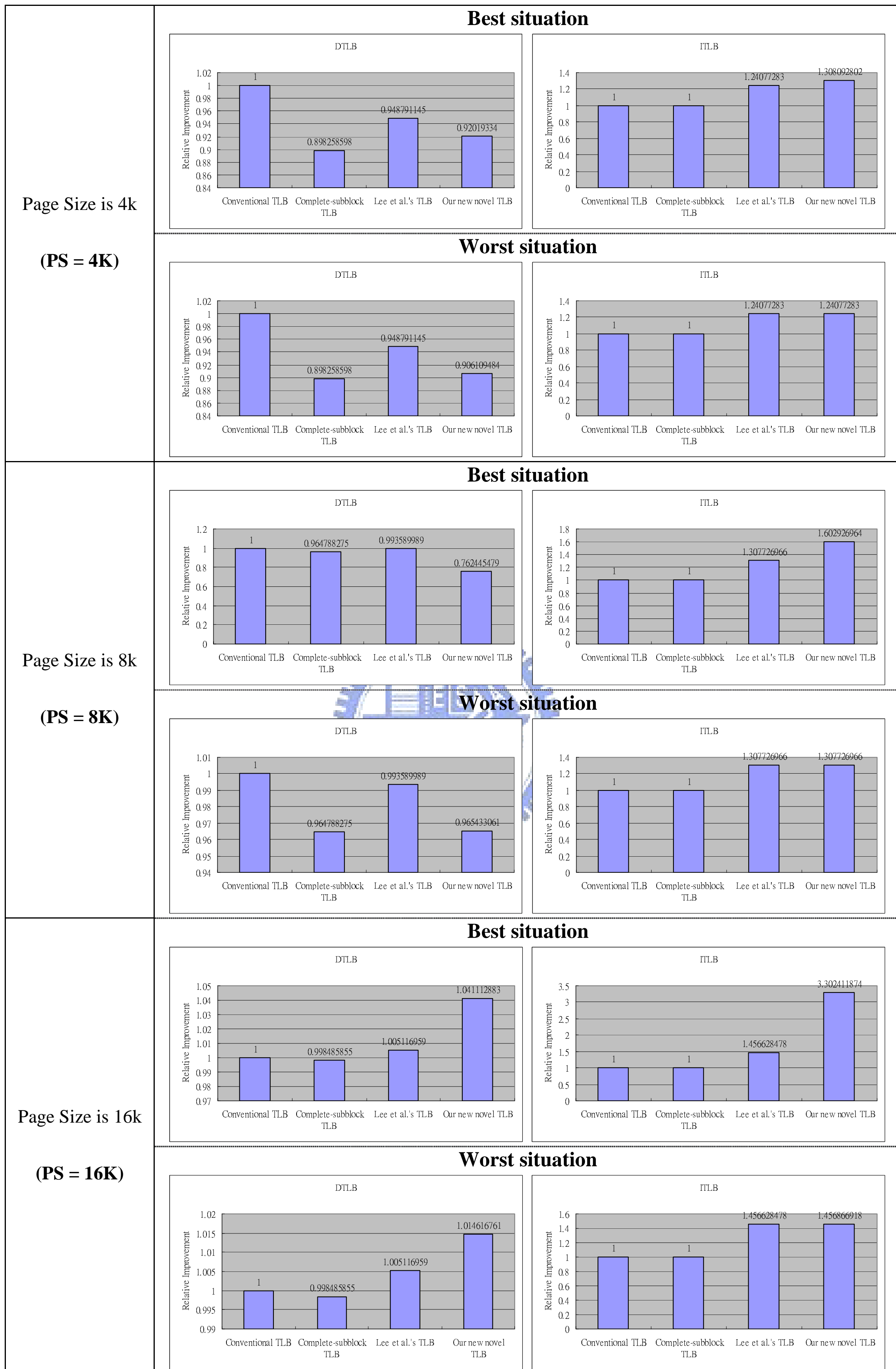


Figure 29: The relative improvement of miss rate in *equake*.

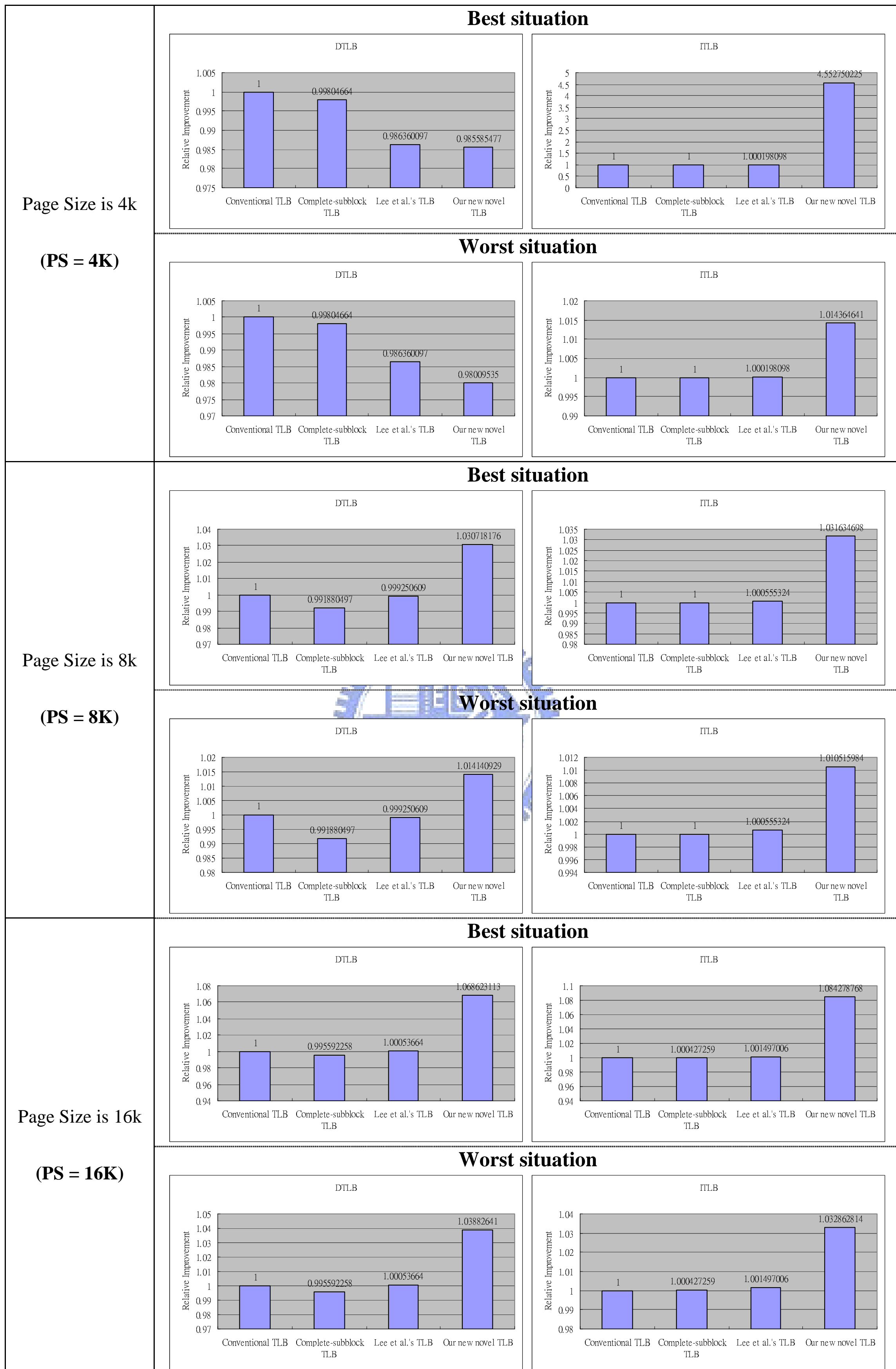


Figure 30: The relative improvement of miss rate in *mgrid*.

Reference

[1] Advanced Micro Devices, Inc., *Software Optimization Guide for AMD Athlon™ 64 and AMD Opteron™ Processors*, November 2004.

[2] D. Burger and T.M. Austin, “The SimpleScalar Tool Set Version 2.0,” *Technical Report 1342*, Computer Sciences Department, University of Wisconsin, Madison, WI, 1997.

[3] R. Case and A. Padegs. *Architecture of the IBM System/370*, ch 51, pp.830-855, McGraw-Hill Book Company, New York, 1982.

[4] David Channon and David Koch, “Performance Analysis of Re-configurable Partitioned TLBs,” in *Proceedings of the 30th Hawaii Int’l Conf. on System Sciences*, Vol. 5, pp.168-177, 1995.

[5] J. B. Chen, A. Borg, and N. P. Jouppi. “A simulation Based Study of TLB Performance.” Pages 114-123, 1992.

[6] Chang-Jiu Chen, Wei-Min Cheng, Chi-Wen Chang, Wen-Chiuan Liao, “A Novel TLB Architecture To Reduce Miss Rate In Context Switching,” *in the Proceedings of 2004 Taipei International Computer Symposium*, Taipei, 2004.

[7] D. W. Clark and J. S. Emer. “Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement.” *ACM Trans. On Computer Systems* Vol. 3, No. 1, pp.31-62, February 1985.

[8] Zhen Fang, Lixin Zhang, John B. Carter, Wilson C. Hsieh, and Sally A. Mckee, “Reevaluating Online Superpage Promotion with Hardware Support,” *in Proceedings of the 7th Int’l Symp. on High-Performance Computer Architecture*, pp.63-72,2001.

[9] Erin Farquhar and Philip Bunce, *The MIPS Programmer's Handbook*, Morgan Kaufmann, San Francisco, CA, 1994.

[10] Michael J. Flynn, Computer Architecture – *Pipelined and Parallel Processor Design*, ch. 5.16, pp.323-325, Jones and Bartlett Publishers, Boston, 1995.

[11] Intel Corp., *IA-32 Intel® Architecture-Software Developer's Manual Vol. 3-System Programming Guide*, 2004.

[12] Intel Corp., *Intel® Itanium® 2 Processor Reference Manual – For Software Development and Optimization*, May 2004.

[13] Intel Corp., *Pentium® Pro Family Developer's Manual Vol. 3 – Operating System Writer's Guide*, Dec. 1995.

[14] B.L. Jacob and T.N. Mudge. "Virtual memory: Issues of implementation." *IEEE Computer*, Vol.31, No. 6, pp.33-43, June 1998.

[15] Gokul B. Kandiraju and Anand Sivasubramaniam, "Going Distance for TLB Prefetching: An Application-driven Study," in *Proceedings of the 29th Annual Int'l Symp. on Computer Architecture*, 2002.

[16] Gokul B. Kandiraju , Anand Sivasubramaniam, Characterizing the *d*-TLB behavior of SPEC CPU2000 benchmarks, *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, June 15-19, 2002, Marina Del Rey, California.

[17] Jung-Hoon Lee, Jang-Soo Lee, She-Woong Jeong, and Shin-Dug Kim, "A Banked-Promotion TLB For High Performance and Low Power," in *Proceedings of the 2001 Int'l Conf. on Computer Design*, pp.118-123, 2001.

[18] Jung-Hoon Lee, Jang-Soo Lee, and Shin-Dug Kim, "A dynamic TLB management structure to support different page sizes," in *Proceedings of the Second IEEE Asia-Pacific Conference on ASICs*, pp.299-302,2000.

[19] Ashley Saulsbury, Fredrik Dahlgren, Per Stenstrom, "Recency-Based TLB Preloading," in *Proceedings of the 27th Int'l Symp. on Computer Architecture*, pp.117-127, 2000.

[20] Naohiko Shimizu, and Ken Takatori, “A Transparent Linux Super Page Kernel for Alpha, Sparc64 and IA32 – Reducing TLB Misses of Applications,” *ACM SIGARCH Computer Architecture News*, Vol. 31 Issue 1, March 2003.

[21] SimpleScalar LLC, http://www.simplescalar.com/

[22] SPARC International Inc. The SPARC Architecture Manual Version 8, pp.241-260, 1992.

[23] Sun Microstsems, Inc., *UltraSPARC® III Cu User Manual Version 2.2.1*, Jan. 2004.

[24] Mark Swanson, Leigh Stoller, and John Carter, “Increasing TLB Reach Using Superpages Backed by Shadow Memory,” in *Proceedings of the 25th Annual Int’l Symp. On Computer Architecture*, pp.204-213, 1998.

[25] M. Talluri. *Use of Superpages and Subblocking in the Address Translation Hierarchy*. Ph.D. thesis, Dep. Of CS, University of Wisconsin at Madison, 1995.

[26] M. Talluri and M. Hill. “Surpassing the TLB Performance of Superpages with Less Operating System Support,” In *Proceedings of the Sixth Int’l Conference on Architectural Support for Programming Languages and Operating Systems*, pp.171–182, 1994.

[27] M. Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. “Tradeoffs in Supporting Two Page Sizes,” In *Proceedings of the 19th Annual Int’l Symp. on Computer Architecture*, pp.415-424, May 1992.

[28] Joel M. Tendler, Steve Dodson, Steve Fields, Hung Le, abd Balaram Sinharoy, *IBM@server POWER4 System Microsrchitecture*. IBM Server Group, October 2001.