# INCREASING TLB REACH USING TCAM CELLS

A Thesis

by

ANUJ KUMAR

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2004

Major Subject: Computer Engineering

# INCREASING TLB REACH USING TCAM CELLS

A Thesis

by

ANUJ KUMAR

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

| | |
|---|---|
| Rabi N. Mahapatra<br>(Chair of Committee) | Jianer Chen<br>(Member) |
| A.L.Narasimha Reddy<br>(Member) | Valerie E. Taylor<br>(Head of Department) |

December 2004

Major Subject: Computer Engineering

## ABSTRACT

Increasing TLB Reach Using TCAM Cells. (December 2004)

Anuj Kumar, B. Tech., Indian Institute of Technology, Kanpur, India

Chair of Advisory Committee: Dr. Rabi N. Mahapatra

We propose dynamic aggregation of virtual tags in TLB to increase its coverage and improve the overall miss ratio during address translation. Dynamic aggregation exploits both the spatial and temporal locality inherent in most application programs. To support dynamic aggregation, we introduce the use of ternary-CAM (TCAM) cells at the second-level TLB. The modified TLB architecture results in an increase of TLB reach without additional CAM entries. We also adopt bulk prefetching concurrently with aggregation technique to enhance the benefits due to spatial locality. The performance of the proposed TLB architecture is evaluated using SPEC2000 benchmarks concentrating on those that show high data TLB miss ratios. Simulation results indicate a reduction in miss ratios between 59% and 99.99% for all the considered bench-marks except for one benchmark, which has a reduction of 10%. We show that the L2 TLB when enhanced using TCAM cells is an attractive solution to high miss ratios exhibited by applications.

To My Parents

## ACKNOWLEDGMENTS

I would like to thank Dr. Rabi Mahapatra for all the help and guidance that he offered. I am grateful to him for letting me take up this interesting work.

I would like to thank Dr. Jianer Chen and Dr. Reddy for serving on my committee. In addition, I am grateful to both of them for the excellent courses offered by them. I would also like to thank Dr. Dmitri Loguinov for his technical guidance and financial support in the first year of my graduate program.

I have greatly enjoyed my time spent in the office with Vivek, Sai and Seraj. I thank them for all their help and for the numerous conversations we had together.

I am extremely grateful to all my friends who always were there for me and have made graduate life memorable. In particular, I thank Shravan, Rishi, Abhijit, Kaushik, and Abhiram.

# TABLE OF CONTENTS

# LIST OF TABLES

LIST OF FIGURES

# CHAPTER I

# INTRODUCTION

The increase in size and complexity of the application programs driven by the recent advancement in the DRAM technology has led to manifold increase in virtual address space. Therefore, a significant amount of memory spanning up to several pages is now required to store the much increased virtual to physical address translations. However, a small cache of recently used translation is stored in Translation Lookaside Buffer (TLB) to hide the memory latency. On every instruction fetch or data reference, TLB is looked up for address translation. If the mapping is not found then it results in a TLB miss upon which the miss handler looks up the page containing the translation in the main memory. The new mapping is then stored in the TLB for future references by replacing one of the previously stored mappings that is chosen based on a replacement policy.

The address translation is one of the most critical operations in modern processors. Previous studies show the most frequent kernel operation is TLB miss handling leading to significant processing time being spent to handle the TLB misses [1], [2], [3]. Due to the continuous increase in instruction level parallelism, clock frequency, and size of working set of applications, the impact of TLB performance on the overall application processing time will continue to grow.

The performance of any TLB design is evaluated in terms of two metrics - access time and miss ratio. Since the address translation is in the critical path of instruction fetch and data reference, the TLB look up needs to be completed as soon as possible. The TLB miss rate should be minimized because of severe miss penalty (30-50

The journal model is *IEEE Transactions on Automatic Control.*

clock cycles). This penalty is sure to increase further due to the ever-increasing gap between the processor and the memory access speed. Consistent efforts are being made to reduce the TLB miss rate. Two popular approaches to decrease TLB miss rate that have emerged in the research community are increasing TLB reach and prefetching. Use of superpages has been proposed to increase the TLB reach. The subblock TLB [4] and use of shadow memory [5], [6], [7] are some of the schemes that implement superpages. These schemes either place a considerable overhead on the memory management unit of the operating system or require significant architectural changes. Even though TLBs of most modern processors support multiple page sizes, the use of superpages in prevalent operating systems is rather limited due to their associated complexities. The subblock TLB further has a limitation of supporting only fixed sized superpages. Recency based prefetching [8] and distance prefetching [9] have been proposed to reduce miss rate but at the expense of complex prediction strategies. Furthermore, a TLB architecture should also support simultaneous multiple accesses to satisfy execution of multiple instructions in a single cycle to provide high throughput [10].

We show that majority of SPEC2000 benchmarks exhibit a considerable amount of spatial locality at the page level granularity and have a large scope of aggregation. Aggregation is the process of merging several virtual tags in TLB into a single entry thus freeing many TLB entries so that they can be filled up by other address translations. Based on these observations, we propose a modified TLB architecture that supports bulk prefetching and static aggregation similar to the concept of complete-subblock proposed in [4]. Further, we employ dynamic aggregation of virtual tags in the L2 TLB to increase TLB reach. To support dynamic aggregation, we introduce the use of TCAM cells [11] along with CAM cells at the second-level TLB. To the best of our knowledge, this is the first attempt to use TCAM cells in a TLB design to enhance TLB reach. We show that dynamic aggregation not only exploits spatial locality but also exploits temporal locality present in the applications. A significant reduction in the overall TLB miss rate is achieved without additional CAM overhead (i.e., no change in the number of virtual tags in the TLB). The aggregation technique proposed here is a hardware-controlled approach. Each virtual tag in TLB maps to a variable number of physical pages due to the use of TCAM cells. In addition, the TCAM cells allow mapping of larger sized pages (e.g., Itanium CPU provides ten different page sizes from 4KB to 256MB) in a single TLB entry. Thus, our proposed scheme provides the benefits of increasing the TLB reach as in the case of superpages but without most of the limitations and associated complexities.

The rest of the paper is organized as follows. Chapter II discusses related work. The details of experimental setup are presented in Chapter III. Chapter IV describes some of our observations regarding L2 TLB. Bulk prefetching and static aggregation strategies are described in Chapter V. Chapter VI introduces the dynamic aggregation process in detail. The timing analysis of the proposed architecture is given in Chapter VII. Chapter VIII describes the usage of TCAM cells in the TLB to support variable sized pages. Chapter IX concludes the paper with important results and a discussion on future research directions.

# CHAPTER II

# BACKGROUND

Over the past few years, lot of research has been done to improve TLB access time and miss rate. Most common solution to reduce the access time is to support multi-level TLB with a small L1 TLB followed by a larger L2 TLB [3], [12]. The second-level TLB is looked up only when there is a L1 TLB miss. Looking up a smaller TLB decreases the access time. Like the cache hierarchical structure, TLB hierarchy also follows the inclusion property (entries present in lower level is also present in higher level).

Two popular approaches to decrease TLB miss rate are increasing TLB reach and prefetching. TLB reach is the total physical memory mapped in the TLB that is equal to the product of number of entries in the TLB and the page size of each entry. The simple solutions to increase TLB reach are increase in page size and number of TLB entries but both of them are not practical. Increasing the page size worsens the page internal fragmentation and memory utilization and thus the overall performance of the system. Increase in size of TLB has not been able to keep pace with the increase of virtual address space due to constraints in chip area, access time and cycle time.

## A. Superpages

Use of superpages has been proposed to increase TLB reach. Superpage is a page in memory that is created by bringing several properly aligned pages that have continuous virtual address space together in the main memory. By doing so, several pages constituting the superpage will have a single TLB entry that frees up entries for other mappings thereby increasing TLB coverage. The issues involved in superpage management are allocation (copying or reservation based), fragmentation control, promotion, demotion and eviction of superpages [13]. Making best use of multiple sized pages is also a challenge. The benefit of increasing TLB reach can be eclipsed by increase in internal fragmentation and high disk traffic. Online superpage promotion [14] is a scheme for dynamically supporting superpage management by monitoring TLB misses to decide when to create a superpage. This scheme takes into account both the benefit of superpage promotion (reduction in TLB misses) and the cost (page copying). The analysis takes place during the TLB miss handling thereby increasing the cost of TLB miss. The paper proposes several promotion policies – OFFLINE, ONLINE, APPROX-ONLINE and ASAP. Supporting superpages places a significant overhead on the memory management unit of the operating system.

## B. Shadow Memory

Use of shadow memory [5], [6], [7] has been proposed to support superpages without much support from operating system. Many modern processors have 32-40 physical address bits, which can address four GB to one TB of actual memory. This greatly exceeds the amount of physical memory available. The scheme makes use of the unused physical address range to virtualize physical memory. Shadow memory is another address space between virtual and physical address space. TLB stores mapping of virtual address space and shadow address space. The scheme requires a secondary MMU and TLB in the main memory controller. This memory-controller TLB (MTLB) translates shadow address to physical addresses. The advantage here is it allows superpages to be created from discontinuous base physical pages by using continuous shadow pages without any modification to the processor MMU.

However, this scheme is also difficult to manage and requires many architectural changes. First, it requires MTLB (a new component) for implementation. The high-end machines that need most support of superpages use all addressable physical memory preventing the use of shadow address space. Use of MTLB can lead to imprecise exceptions. The memory issues (load and store instructions) that pass the TLB translation successfully can be identified as improper in the MTLB. In such a case, the CPU must back-off all the instructions that have been executed in advance. Operating system can use only rough reference information. Cache flush needs to be performed whenever pages are deallocated. The authors in [6] address these drawbacks by integrating partial sub-block TLB with shadow memory. The limitation of the new scheme is that maximum superpage size is fixed by the superpage factor of its partial sub-block TLB.

Fang *et.al.* [5] compares creating superpages via copying and remapping pages within the memory controller [15], [16], [17]. Remapping based promotion outperforms copying based promotion most of the time. Copying based promotion is slightly more effective on superscalar processors than single-issue processors.

## C. Prefetching Schemes

Another direction to decrease TLB miss rate is prediction and prefetching. Various prefetching techniques have been proposed mainly in context of caches and I/O. The study related to TLB is fairly recent. Prefetching can be classified based on the following two criterions:

- The schemes that predict strided reference patterns (sequential [18], [19] and arbitrary stride prefetching [20], [21]).
- The schemes that use much longer history to make predictions. (markov prefetching [22]).

Prediction schemes have different degree of effectiveness for different reference patterns. The authors in [2], [23] consider prefetching TLB entries only for cold start. However, TLB misses have been seen to exhibit some regularity [8] thus detecting the pattern can significantly help in reducing TLB misses. Several prefetching schemes have been proposed such as recency based [8], and distance prefetching [9].

The prediction is based on TLB miss streams otherwise prediction would come in the critical path of TLB access. All the prefetched entries are stored in a prefetch buffer. At the time of a miss, this buffer is looked up in parallel with the TLB to find out if the entry has already been prefetched. Prediction accuracy is the metric used to compare the effectiveness of the prefetching techniques. Prediction accuracy is the fraction of misses for which the translations can be found in the prefetch buffer (due to accurate prediction i.e., the translation for the address causing the miss is prefetched before the miss actually occurs). Since prefetching causes additional memory traffic, one wants a high accuracy from any prefetching scheme. Mispredictions throw away prefetched entries from prefetch buffer (not from TLB) since an entry is removed from prefetch buffer and placed in TLB only upon a prefetch buffer hit.

Recency Prefetching is the first mechanism proposed exclusively for TLB. The idea behind the mechanism is that pages referenced at around the same time in the past will also be referenced at around the same time in the future. A LRU stack of page table entries is maintained to keep the information needed for prediction. Each page table entry has two pointers, previous and next, pointing to two page table entries that were evicted out just before and after this entry. These pointers are actually stored in the page tables. An entry is put on the top of the stack on being evicted out of the TLB and its next pointer is set to the previous entry that was removed (whose previous pointer is set to this entry). On a TLB miss, the prefetch mechanism fetches the entries corresponding to the next and previous pointers into the prefetch buffer expecting that these entries would also be needed. The advantage of this scheme is no additional storage cost on-chip since all the prediction information is kept in the page table. However, it comes at the cost of an increased page table size.

Distance prefetching is a mechanism that try to detect many patterns that recency and markov schemes can detect in addition to the regular strided reference behavior. The advantage is requirement of little space and no extra memory operations for prefetching. This mechanism works by tracking the differences between successive addresses (spatial separation or distance). Distance Prefetching maintains a table of distance values and each such value maps to few other distances that occurred immediately after this distance was encountered. On a TLB miss, the scheme indexes the table based on the current distance (difference in page number between current miss and previous miss) and prefetch the tags corresponding to all the distances that are mapped by current distance.

The prediction accuracy for all the above-mentioned techniques are compared in [9] over a diverse set of applications (SPEC 2000 benchmarks, MediaBench, Etch traces and Pointer Intensive benchmark suite). Distance prefetching was found to be more effective than the other schemes in majority of the applications.

## D. TCAM Cells

Normal memory devices access data according to the address of the memory location. This memory model is not suitable for lookup applications, which require searching capability based on content rather than address. Some of the examples of such applications include IP address lookup, TLB lookup, packet classification, pattern recognition, firewall implementation. In order to accomplish content-based lookup, they instead rely upon a suitable retrieval data structure. This may cause several memory accesses for a single lookup operation and can become a serious bottleneck in demanding lookup applications.

One of the solutions to address this bottleneck is to use memories with built in comparison capabilities known as Content Addressable Memories (CAMs). They bring down the complexity of content-based lookup operation to a single memory access and provide one lookup each memory cycle if implemented in pipelined fashion. However CAMs based solution run into problems if the lookup key is allowed to have variable length. The packet classification, access control list and address lookup in Internet exhibit such characteristics.

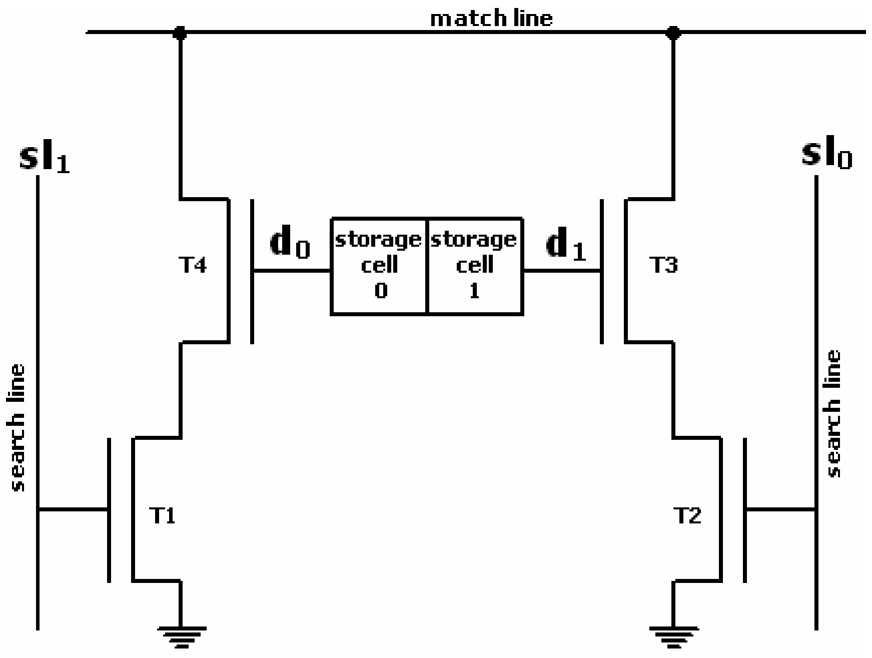


Fig. 1. TCAM cell.

For such applications, CAM architecture has been extended to store a “don’t care” state in addition to ‘0’ and ‘1’ state. These memories are called Ternary Content

Addressable Memory (TCAM) and they match both 0 and 1 if the don't care bit is set. TCAMs are increasingly being used in high performance backbone IP routers to achieve IP lookup at line speed typically 100 million lookups per second.

A NOR based TCAM cell is shown in Figure 1 [1]. It uses two SRAM based binary storage cell to store states 0, 1 and don't care. It has four transistor switches T1-T4 to assist comparison and includes a total of 16 transistors in comparison to a SRAM cell that is made of 6 transistors. Therefore, use of TCAM results in higher gate count and larger chip area. An array of TCAM cells are arranged to form a TCAM word. A TCAM word can be used to store a virtual tag in the TLB that assist in variable length matching. An array of TCAM word forms a TLB.

[1]Adapted from http://www.eecg.toronto.edu/ pagiamt/cam/camintro.html

# CHAPTER III

# EXPERIMENTAL SETUP

Some of the SPEC 2000 benchmarks exhibit high data TLB miss rate as pointed out in [24]. In this paper, we consider nine application benchmarks from SPEC2000 [25] to study the effectiveness of our approach. Five of them (galgel, twolf, vpr, lucas and mcf) show high data TLB miss rate while the other four (swim, bzip, gzip and crafty) belong to the set of benchmarks that show relatively lower miss rates. Three (galgel, lucas and swim) of these are floating-point benchmarks while the rest (twolf, vpr, mcf, gzip, bzip and crafty) are integer benchmarks. All the binaries are taken from [26]. These benchmark are compiled on an Alpha 21264 machine using Compaq's cc V5.9-008, cxx V6.2-024, f90 V5.3-915 and f77 V5.3-915. These benchmarks are run using SimpleScalar-3.0 toolset that supports Alpha architecture. The sim-cache component of the toolset which is a multi-level cache simulator has been used for functional (not cycle-by-cycle) simulations of instructions. Separate TLBs are considered for instructions and data references. However, only the data-TLB references are examined in this paper since the instruction-TLB miss rates for the SPEC2000 benchmarks are very low. We do not consider the effect of operating system (i.e., no interference from OS, no entries reserved for OS pages in TLB, no context switch). We are mainly interested in the characteristics of the applications involved with address translation in data TLB. We do not run the benchmarks to completion. Instead, we execute almost $2^{32}$ ($\sim$ 4.29 billion) instructions and take average of their outcomes. We believe that this many instructions should be sufficient to capture the true behavior of the applications. Throughout the paper, we consider fully associative TLB and least recently used (LRU) policy for replacement. The page size is fixed at 4KB for all the simulations. The term miss rate is defined as the ratio of total number of TLB miss to total number of data memory references. We represent total number of bits in virtual tag as W and |L2| to denote the size of the second-level TLB.

# CHAPTER IV

# OBSERVATIONS FROM TWO-LEVEL TLB

Two-level TLBs provide reduced access time as shown in [3], [24], [12] when compared to a single-level TLB architecture. Modern processors such as Itanium (32-entry L1, 96-entry L2), AMD Athlon (32-entry L1, 256-entry L2) and others provide multi-level TLB structures. Table I gives the overall miss rates in nine SPEC2000 benchmarks using a two-level TLB archi-tecture. We observed a high percentage of TLB misses for the first five benchmarks while the rest have a very low miss rate. In fact, four of the nine benchmarks show very poor L2 hit rates in spite of its large size. To improve the TLB hit rates, we examine properties related to TLB misses and virtual tag values to get an indication of the benefits that can be achieved by exploiting spatial locality available in the benchmarks.

## A. Will Bulk Prefetching Help?

We introduce a metric called *miss-distance* to examine if bulk prefetching can help in reducing overall TLB miss rates. Miss-distance is the difference between the tag value that results in a L2 TLB miss and the nearest tag value stored in L2 TLB. The following equation defines miss-distance:

$$|V_L - V_i| \; such\, that \; |V_L - V_i| \leq |V_L - V_j| \; \forall j, \; j \in \{1, ...., |L2|\}$$

where

$V_L$ = value of the looked up tag that resulted in a L2 TLB miss and

$V_i$= value of the tag stored in $i^{th}$ entry of L2 TLB.

Smaller miss-distance values indicate higher spatial locality present in the benchmarks.

Table I. Overall miss rate for two-level CAM based TLB with L1 = 32 and L2 = 256 entries.

| Benchmarks | L1 Miss Rate | L2 Hit Rate | Overall Miss Rate |
|---|---|---|---|
| GALGEL | 13.13% | 0.89% | 13.01% |
| TWOLF | 2.94% | 84.14% | 0.466% |
| MCF | 5.25% | 21.8% | 4.105% |
| VPR | 7.52% | 96.54% | 0.259% |
| LUCAS | 1.13% | 2.26% | 1.10% |
| SWIM | 0.14% | 11.68% | 0.126% |
| BZIP | 1.33% | 89.25% | 0.14% |
| GZIP | 0.53% | 98.6% | 0.0073% |
| CRAFTY | 2.49% | 98.27% | 0.043% |

Figure 2 shows the cumulative density distribution of miss-distances (i.e., $y$-axis shows the fraction of misses that have a miss-distance less than or equal to a given value on the $x$-axis). We measure miss-distance on each TLB miss up to a value of 32. All miss-distances larger than 32 are shown at the miss-distance value of 33.

For all the benchmarks except lucas, a high percentage of misses take place within a very small miss-distance. In fact, galgel and gzip have over 99% of their misses within a miss-distance of two. Moreover, the benchmarks twolf, swim, bzip and crafty show more than 92% of their misses within miss-distance of four. The curves for mcf and vpr show sharp rises up to a miss-distance of four after which the CDF increases gradually. These results signify a lot of spatial locality for most of the benchmarks when we look at data references at the page level granularity. Based on these observations, it seems that bulk prefetching will help reduce miss rates for a majority of the benchmarks.

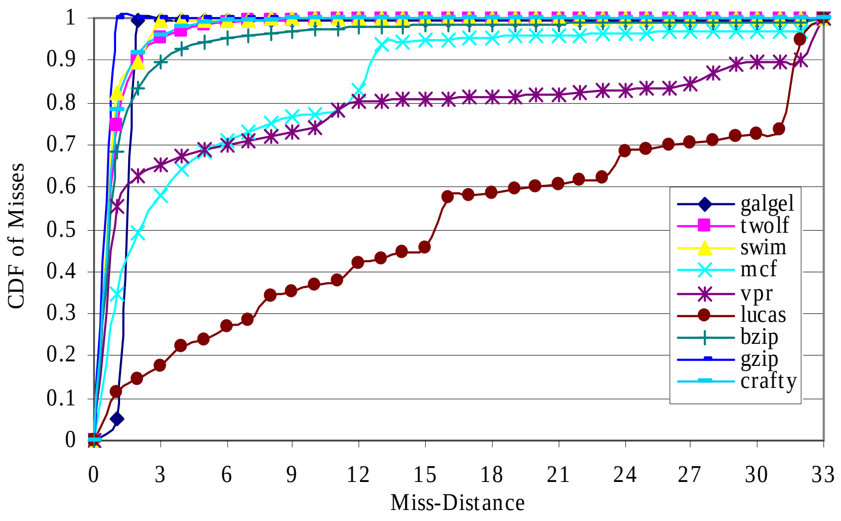


Fig. 2. Cumulative Density Function of misses for different miss-distances of two-level CAM based TLB with L1 = 32 and L2 = 256 entries.

## B. Will Aggregation Help?

We define *space-gain* as the total number of TLB entries that can be freed due to aggregation. We would like to aggregate virtual tags that differ in their least significant bits and map them to a single entry in the TLB. By doing so, many TLB entries can be freed to accommodate additional mappings and enhance the TLB coverage.

We measure the space-gain by counting the number of entries in the L2 TLB that have only their $Z$ least significant bits different. We explain the process of aggregation with an example having the value of $Z = 3$. Let (10010001, 01101000, 01101010, 00110110, 01110111, 00110100, 00110001, and 01101100) be the tag values in a TLB of size 8. We see that the tag values 01101000, 01101010, and 01101100 are the same value except for their 3 least significant bits. After aggregation, a single entry corresponds to these three tags thus freeing two TLB entries. Similar is the case with 00110110, 00110100 and 00110001 tag values. Overall, we can have a space-gain of four. Figure 3 presents the above example of aggregation with 3 bits.

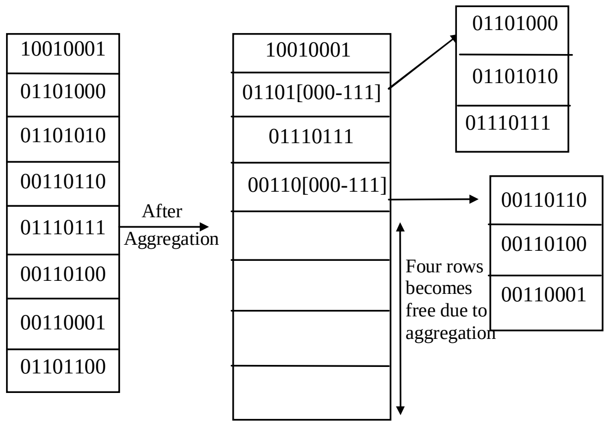
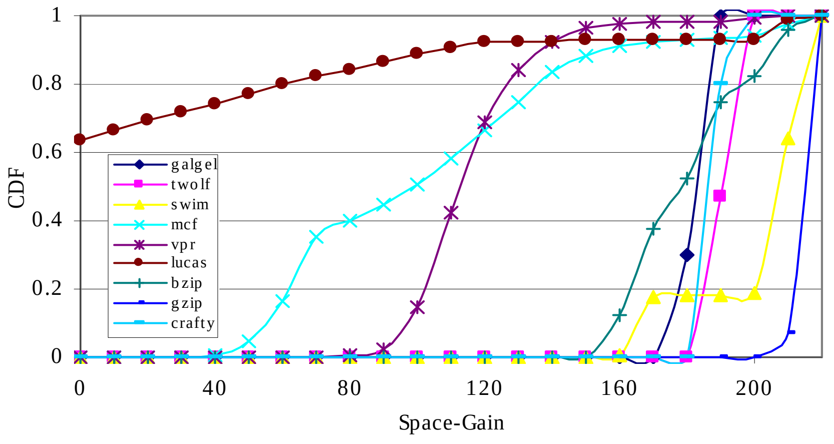
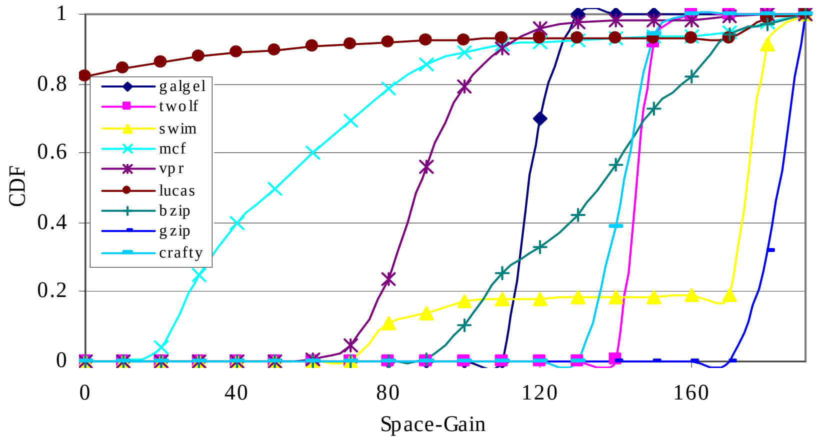


Fig. 3. Example of aggregation process.

Figure 4 shows the cumulative density distribution of space-gain (i.e., $y-$axis shows the fraction of time space-gain is less than or equal to the value on the $x-$axis). The CDF is plotted by measuring the space-gain after every 50 TLB miss. The L2 TLB used in the simulation contains 256 entries. Space-gain is maximized when all the entries in the L2 TLB can be aggregated. If this is the case, then we will require only 32 entries with each of these 32 entries mapping 8 translations. The rest of the TLB entries then become free. Therefore, the maximum space-gain that can be achieved in this example is 224. Figure 4(a) shows the CDF of space-gain for aggregation with $Z$ =3 bits. CDF plots for most of the benchmarks show a sharp rise indicating that a majority of the time the space-gain has a constant value indicated by the location of the steep rise. On an average, a space-gain of 210-220 entries is seen for swim and gzip while 180-190 entries is gained for galgel, twolf, bzip and crafty. Mcf and vpr show a moderate space-gain of 120-130 entries. Lucas does not seem to benefit much from aggregation. Figure 4(b) shows the CDF of space-gain for the aggregation with $Z$= 2 bits. The CDF plots follow the same trend as in Figure 4(a), except the maximum space-gain is 192. However, both cases show a large scope of aggregation for most of the benchmarks.



(a)



(b)

Fig. 4. Cumulative Density Function (CDF) of space-gain of two-level CAM based TLB with L1 = 32 and L2 = 256 entries. (a) *aggregation with Z = 3 bits.* (b) *aggregation with Z = 2 bits.*

Both bulk prefetching and aggregation techniques exploit spatial locality among the data references to a large extent that would surely help in reducing the overall TLB miss rate. In the next section, we propose the architectural changes needed in TLB to incorporate these schemes.

# CHAPTER V

# BULK PREFETCHING AND STATIC AGGREGATION

A TLB structure has two parts: virtual tags and data (physical addresses) with their attributes. Virtual tags are stored in the CAM array which we refer to as the *virtual tag table* (VTT) and the data portion is stored in SRAM which we refer to as the *physical tag table* (PTT). Based on the results presented in Chapter IV, we adopt the size of bulk prefetching to be four since all benchmarks except lucas show a very high percentage of misses with a miss-distance equal to or less than four. The four page table entries fetched from main-memory in response to a TLB miss are collectively referred as a *block*. One of the entries in a block contains the physical address corresponding to the virtual tag that resulted in a miss. The translations included in a block have virtual tag values such that they differ only in their two least significant bits. The remaining ($W - 2$) virtual tag bits common to all four translations represent the virtual tag for the entire block; we refer to it as *block tag* and store it as an entry in the VTT. The data portion corresponding to a block tag (physical memory addresses and attributes for four pages) is stored in consecutive entries in the PTT. We refer to this scheme as *static aggregation.*

Figure 5 shows the high level design of a CAM based L2 TLB that supports bulk prefetching and static aggregation. Each block tag in VTT maps to 4 entries in the PTT. Since a block is brought from memory to the TLB and evicted out of the TLB back to memory in a group, we have a single reference bit for the block in the VTT. The physical memory addresses for all the four pages in the block are stored sequentially in the PTT in ascending order of their virtual tag addresses. Each entry in the VTT contains a pointer to a PTT entry that stores the physical address of first translation of the block. We refer to this pointer as the *PTT block pointer.*

The PTT stores the physical memory address, valid bit and attribute bits for each page entry. On a TLB hit, the PTT block pointer and the two least significant bits of the looked up virtual address determine the location in the PTT containing the physical address. On a TLB miss, the least recently used block in the L2 TLB is replaced by the block that is fetched in response to the miss. We refer to this scheme as *L2_static_aggregation.*

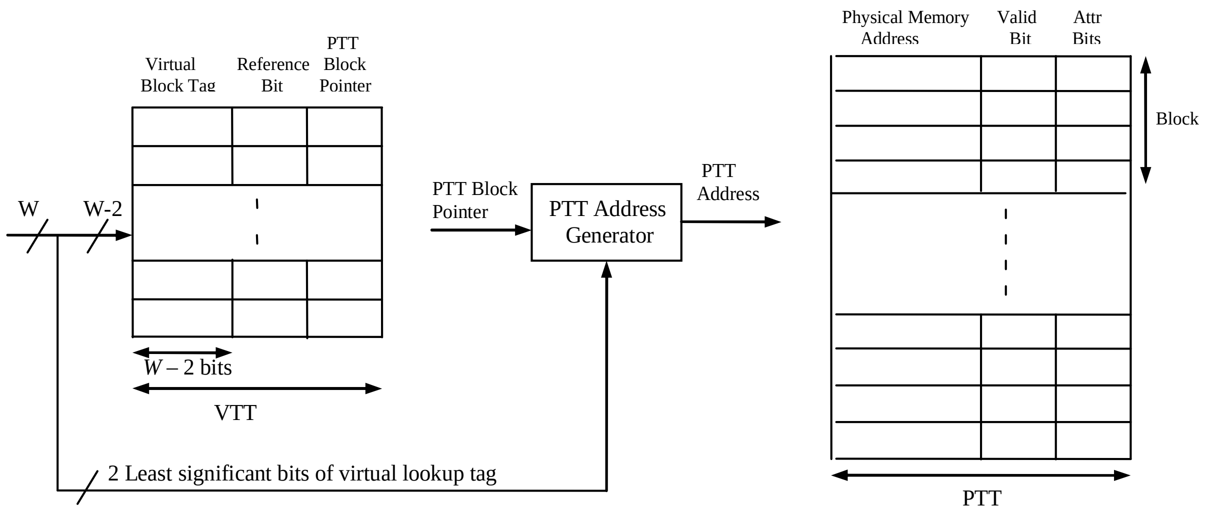


Fig. 5. High Level Design of CAM based L2 TLB showing VTT and PTT that supports aggregation with bulk prefetching of size four.

To improve the L1 hit rate, we can also incorporate the static aggregation scheme into the L1 TLB. In this case, both the L1 and L2 VTTs can share a common PTT. We refer this process as *L1_L2_static_aggregation.* The overall miss rate for both schemes remains the same due to the common PTT that holds all the translations. However, the L1 TLB reach is increased four times which leads to an increase in the L1 hit rate (decrease in L2 hit rate) and a reduction in the overall access time. Table II shows the overall miss rate for CAM based two-level TLB architecture with bulk prefetching and static aggregation. While we consider the size of VTT in L1 TLB to be 32 entries, the VTT size in the L2 TLB has been kept at 128 instead of 256 as in

Chapter IV. Even with half of the VTT size in the L2 TLB, we are able to reduce the miss rates significantly compared to the overall miss-rates listed in Table I.

Table II. Comparison of overall miss rate for two-level CAM based TLB with entries in L1 VTT = 32 and L2 VTT = 128 for the two schemes *L2_static_aggregation* and *L1_L2_static_aggregation*. The overall miss rate is same for both the schemes.

| Benchmarks | L2_static_aggregation | | L1_L2_static_aggregation | | Overall |
|---|---|---|---|---|---|
| | L1 Miss Rate | L2 Hit Rate | L1 Miss Rate | L2 Hit Rate | Miss Rate |
| GALGEL | 13.13% | 51.33% | 6.42% | 0.447% | 6.39% |
| TWOLF | 2.94% | 99.99% | 1.09% | 99.99% | 0.00000826% |
| MCF | 5.25% | 43.15% | 4.43% | 32.87% | 2.98% |
| VPR | 7.52% | 97.7% | 2.77% | 93.75% | 0.173% |
| LUCAS | 1.13% | 9.79% | 1.05% | 3.67% | 1.01% |
| SWIM | 0.14% | 71.67% | 0.04% | 0 | 0.04% |
| BZIP | 1.33% | 97.14% | 0.56% | 93.1% | 0.038% |
| GZIP | 0.53% | 99.65% | 0.00188% | 3.3% | 0.0018% |
| CRAFTY | 2.49% | 99.99% | 0.77% | 99.98% | 0.00000842% |

# CHAPTER VI

# DYNAMIC AGGREGATION

In this section, we show how to dynamically aggregate virtual block tags. Unlike static aggregation, each virtual block tag in the VTT maps to multiple blocks in the case of dynamic aggregation. To support dynamic aggregation, we employ TCAM cells along with CAM cells to store the virtual tag bits in the L2 VTT. A TCAM cell allows the storage of "don't care" state in addition to the 0 and 1 states. When the don't care bit is set, the tag bit becomes a wild card and matches on both 0 and 1. Thus, an entry in the VTT with don't care bits can match multiple look up tags. This way the TLB reach can be enhanced without increasing the VTT size. This benefit comes at the cost of additional PTT entries (SRAM space) and dynamic aggregation overhead.

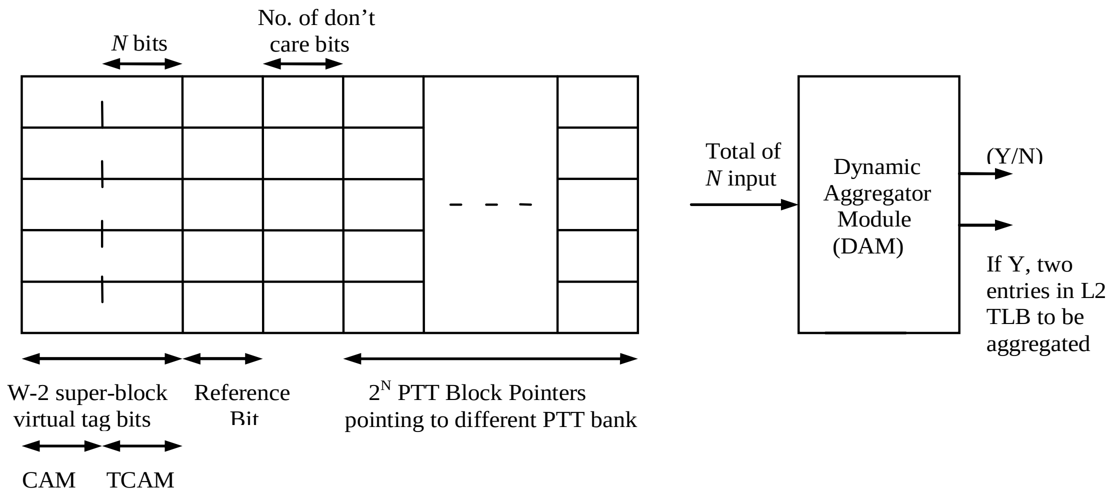

No. of don't care bits
N bits
- - -
Total of N input
Dynamic Aggregator Module (DAM)
(Y/N)
If Y, two entries in L2 TLB to be aggregated
W-2 super-block virtual tag bits
Reference Bit
2^N PTT Block Pointers pointing to different PTT bank
CAM
TCAM

Fig. 6. (a) Schematic of L2 VTT that supports dynamic aggregation by using TCAM cells along with CAM cells. Maximum number of dont care bits allowed is $N$ (i.e., $N$ TCAM cells per VTT entry). (b) The DAM module shown with input and output parameters.

Figure 6(a) shows one of the many possible designs to support dynamic aggregation using TCAM cells. We store only the ($W-$ 2) most significant bits of a virtual tag in the VTT. We use TCAM cells for storing the $N$ least significant bits out of these ($W-$ 2) bits. The rest of the virtual tag bits are stored using CAM cells. The extent of dynamic aggregation depends on the number of TCAM cells in the VTT. This way, a tag entry can map up to a maximum of $2^N$ blocks (i.e., $2^{N+2}$ physical page addresses as each block stores four of them). We refer to all the blocks that are mapped by a single entry in VTT as a *super-block* and we refer to its tag entry as a *super-block virtual tag*. Therefore, the number of physical page addresses in a super-block might be 4, 8, 16. . . .$2^{N+2}$ corresponding to 1, 2, 4,. . . , $2^N$ blocks. Rather than sequentially storing all the physical addresses belonging to a super-block, which would require extra overhead in terms of managing the PTT, we propose to use a bank of PTTs (total of $2^N$) each of size four times that of the VTT. Aggregation takes place between the virtual tags of two entries in the VTT having the same number of don't care bits with one TLB entry merging with the other. As was the case with static aggregation, we keep a single reference bit for the super-block. The number of translations fetched from main-memory on a TLB miss is always four; however, the number of translations that get replaced from L2 TLB may be 4, 8, 16 . . . $2^{N+2}$ depending on the super-block size. The VTT also stores the number of don't care bits for each entry as a separate field which is used by the dynamic aggregator module (DAM). DAM assists in searching for super-block virtual tags in the L2 VTT that are suitable for aggregation. In Section 6.2 we examine the details of DAM.

## A. L2 TLB LookUP

The second-level TLB is looked up only when there is a L1 TLB miss. The steps involved in a L2 TLB look up given in Figure 7.

1. $(W - 2)$ most significant bits of the looked up virtual tag are searched in the L2 VTT.
2. Bits (2 to 4) are passed as input to the PTT bank selector and the mux module.
3. Bits (0 and 1) are passed as input to the PTT address generator.
4. The mux selects the PTT block pointer pointing to the block containing the physical address of the required page.
5. The PTT address generator provides the exact PTT address where the physical address of the looked up tag is stored.
6. The physical address corresponding to the virtual address is obtained from the PTT bank.

The steps 1 to 3 occur simultaneously. Steps 2 to 4 are carried out on a TLB hit. The only extra step involved in a look up that supports dynamic aggregation is the multiplexer stage in Step 2. This was not present in static aggregation as there was only one PTT block pointer.

## B. TCAM Enabled Dynamic Aggregation

Dynamic aggregation takes place in response to the following events:

- Bulk prefetching after a TLB miss, and
- Replacement of a tag from the L1 TLB.

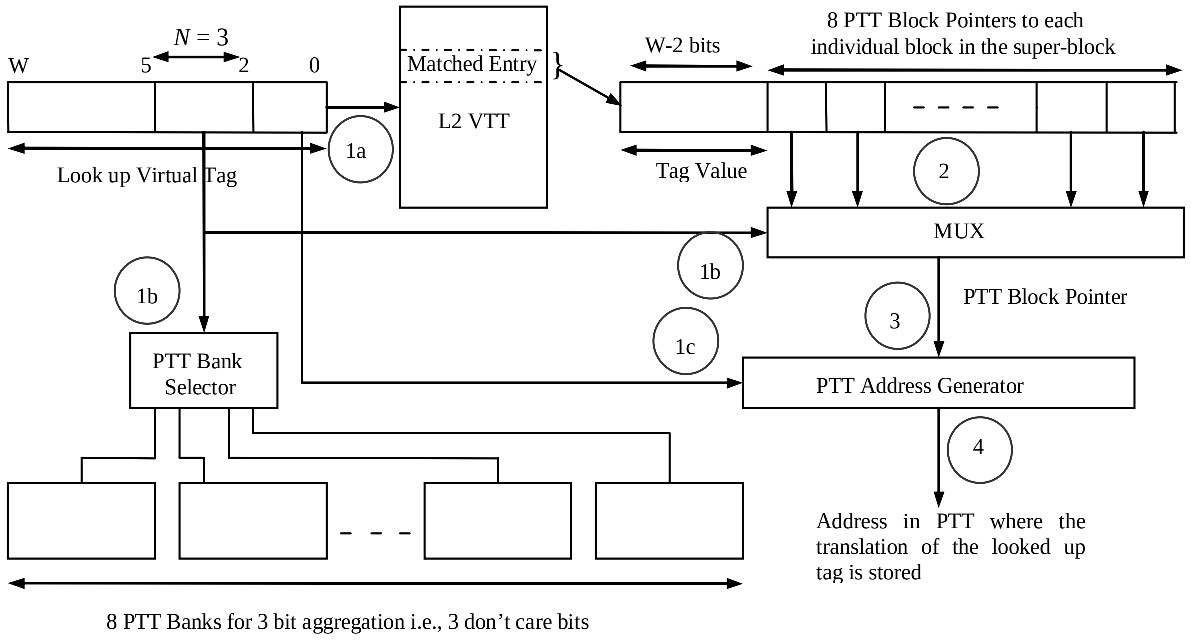

N = 3
W
5
2
0
Matched Entry
L2 VTT
W-2 bits
8 PTT Block Pointers to each individual block in the super-block
1a
Look up Virtual Tag
Tag Value
2
MUX
1b
1b
PTT Block Pointer
3
1c
PTT Bank Selector
PTT Address Generator
4
Address in PTT where the translation of the looked up tag is stored
8 PTT Banks for 3 bit aggregation i.e., 3 don't care bits

Fig. 7. Steps involved during look up in L2 TLB that supports dynamic aggregation. Number of dont care bits allowed here is 3 so that one virtual tag in L2 VTT can map to a maximum of 32 translations.

The ($W$− 2) most significant bits of an incoming block from memory or a L1 TLB replaced block are looked up in the L2 TLB. We refer to this look up as a *missed look-up* and differentiate it from the normal look up coming from processor that we discussed in Section 5. For a missed look-up, the bit next to the most significant don't care bit in a super-block virtual tag is set as don't care and later restored back to its original state of 0 or 1 after a missed look-up is served. If there is an entry with no don't care bits, the least significant bit among the $N$ TCAM bits is set as don't care. This logic can be implemented in TCAM cells. The number of don't care bit field in the VTT as shown in Figure 6 is not changed during the missed look-up. The matched entries due to the missed look-up become input to the DAM. Some of the properties regarding missed look-up include:

*Lemma 1*: Maximum number of entries that matches on a missed look-up is equal to $N$.

Notations

Let the $j^{th}$ virtual tag in L2 TLB be represented by $\mathrm{B}_j = [b_j^{W-1}\ b_j^{W-2} \ldots . b_j^{N+1} \ldots . b_j^2 b_j^1\ b_j^0]$ where bits $b_j^{W-1} \ldots\ b_j^{N+2}$ are store using CAM cells, $b_j^{N+1} \ldots\ b_j^2$ are store using TCAM cells and the last two bits $b_j^1 \quad b_j^0$ are not stored in TLB. Let the lookup tag be $\mathrm{L} = [l^{W-1} l^{W-2} \ldots l^{N+1} \ldots . l^2 \quad l^1 \quad l^0]$.

Background

- The last two bits are not considered in lookup.
- The bits ($N$+1)$\ldots$.2 are stored using TCAM cells. They can be set as don't care.
- Entry with all the TCAM cells set as don't care do not participate in dynamic aggregation.

- Aggregation of an entry with zero don't care to one don't care is always due to missed lookup of block prefetched tag and happens as soon as (at the earliest possible instant) the second entry involved in aggregation is fetched from memory. Therefore, we cannot have two entries in the TLB for which all the bits except bit 2 are same.

- Aggregation of an entry from one don't care to more (aggregation always happen in steps of one) is due to missed lookup of L1 replaced block tag. Two rows (having at least one don't care bits) can be further aggregated (even though present in the TLB at the same time) only when one of the two tags gets replaced from L1 TLB (not at the earliest possible instant).

First, we look at the missed lookup due to L1 replaced block tag. The following reasoning can be easily extended to missed lookup due to block-prefetched tag.

Step 1. For matching to take place with any entry (say $i$): following bits have to be same -

$b_i^{W-1} = l^{W-1}$, $b_i^{W-2} = l^{W-2} \ldots b_i^{N+2} = l^{N+2}$

Step 2.

Maximum numbers of matches occur if one of the entries has all the $N$ bits same as that of lookup bits (i.e., $b_i^{N+1} = l^{N+1}, \ldots.., b_i^3 = l^3$, $b_i^2 = l^2$) or the only bit that may not be same is the least significant bit that can be a don't care bit (i.e., $b_i^2 = *$). Both the cases cannot occur together. Remaining $(N-1)$ matched entries are such that:

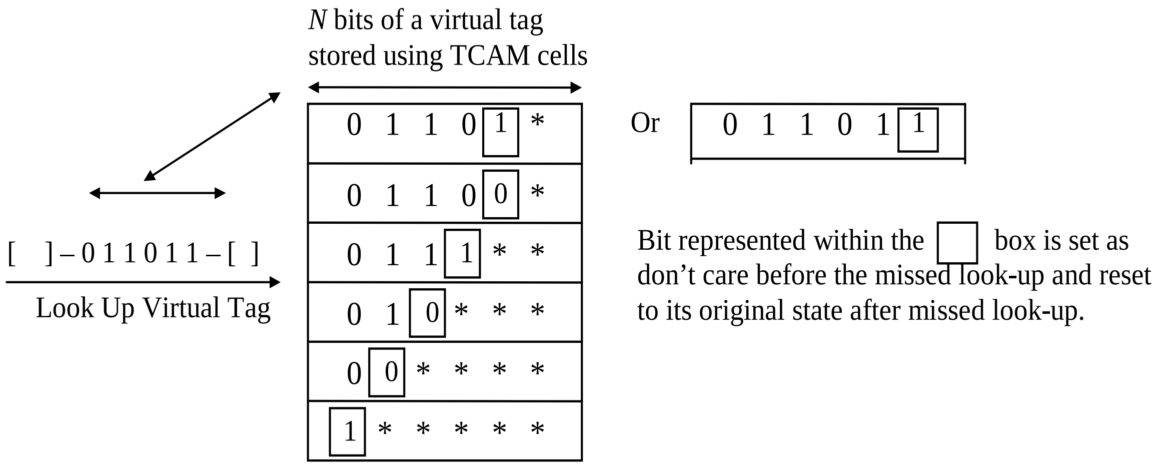
For $k = 2$ to $N$

The $k^{th}$ bit, $b^k = \sim \; l^k$ and all the bits less significant to the $k^{th}$ as don't care. For each k, we have one entry.

Therefore, maximum possible match on a missed lookup due to L1 replaced block tag is $N$. No other entry is possible since these $N$ entries cover all possible combination of $N$ bits.

The only difference between missed lookup of L1 replaced block tag and block prefetched tag is Step 2. As opposed to two possible matches in previous case ($b_i^2 = l^2$or $b_i^2 = *$), we have only one possible case with $b_i^2 =\sim l^2$.

The above proof is illustrated using an example in Figure 8. Only the possible matched entries and the $N$ bits in those entries are given in the figure. The $N$= 6 bits of the lookup tag are 011011. Two entries corresponding to Step 2.1 are 01101* and 011011. Any one of them can be present but not both. The remaining possible entries corresponding to Step 2.2 are 01100*, 0111**, 010***, 00****, and 1*****.



N bits of a virtual tag stored using TCAM cells
0 1 1 0 1 *
0 1 1 0 0 *
0 1 1 1 * *
0 1 0 * * *
0 0 * * * *
1 * * * * *
Or
0 1 1 0 1 1
[ ] – 0 1 1 0 1 1 – [ ]
Look Up Virtual Tag
Bit represented within the box is set as don't care before the missed look-up and reset to its original state after missed look-up.

Fig. 8. Example of missed look-up showing maximum of $N$ possible matches.

*Lemma 2*: Due to the inclusion property, at least one match will always be found in L2 TLB for a missed look-up of L1 TLB replaced block tag. However, for a bulk prefetched block, the missed look-up might not result in any match.

*Lemma 3*: Out of a maximum $N$ possible matches, at most two matches will have the same number of don't care bits. If we have two such matched entries,

they become candidates for dynamic aggregation. Moreover, these two entries will have the minimum number of don't care bits among all the matched entries. Bulk prefetched block will aggregate only with blocks that have zero don't care bits. On the other hand, the L1 TLB replaced tag can aggregate with another entry that has same number of don't care bits as the entry containing the replaced tag.
Proof of Lemma 3 follows from the proof of lemma 1.
The various steps involved in dynamic aggregation are:

1. The matched super-block tags from a missed look-up become the input to DAM.

2. The DAM decides on aggregation by finding two matched entries having the same number of don't care bits. Figure 9 shows examples of missed look-ups and the resulting matched entries. The examples also show if the matched entries result in aggregation. Out of the five examples shown in the figure, the first three missed look-ups are due to an incoming block tag from memory because of a TLB miss while the last two are due to the replaced block tag from the L1 TLB.

3. If aggregation is possible, one of the two entries to be aggregated (preferably one that is less recently used among the two) is merged with the other one. The PTT block pointers from the merging entry are copied to the other entry involved in aggregation. The merging entry is reset and becomes the least recently used entry in the L2 TLB. Figure 10 illustrates an example. R1 and R2 are two entries selected by the DAM for aggregation. Both the entries have one don't care bit i.e., they map to total of eight translations. After merging, R1 has two don't care bits. The tag value after aggregation is 1000101-1** and maps to sixteen translations. R2 is reset and no longer maps to any translations. The figure shows how two PTT block pointers are copied from R2 to R1. As an

| | | |
|---|---|---|
| R1: | 0111011-10* | [1904-1911] |
| R2: | 1001101-10* | [2480-2487] |
| R3: | 0110110-000 | [1728-1731] |
| R4: | 0011111-010 | [1000-1003] |
| R5: | 0111011-11* | [1912-1919] |
| R6: | 0111011-0** | [1888-1903] |
| R7: | 1001100-*** | [2432-2463] |
| R8: | 1001101-0** | [2464-2479] |
| R9: | 0111010-*** | [1856-1887] |
| R10: | 0011111-00* | [0992-0999] |
| R11: | 0110100-000 | [1664-1667] |
| R12: | 0110110-01* | [1736-1743] |

Missed Look Up Tag:
1) From Memory: 1001111-010 [2536-2539]
   No Match. Therefore **no aggregation**
2) From Memory: 0011111-011 [1004-1007]
   No. of Match = 2
   Matched Tag #1 = 0011111-010 (R4) (No. of don't care bits = 0)
   Matched Tag #2 = 0011111-00* (R10) (No. of don't care bits = 1)
   Missed Look Up Tag **aggregates** with Matched Tag #1.
   Final Value of Matches Tag#1 = 0011111-01* [1000-1007]
3) From Memory: 1001101-110 [2488-2491]
   No. of Match = 2
   Matched Tag#1 = 1001101-10* (R2) (No. of don't care bits = 1)
   Matched Tag#2 = 1001101-0** (R8) (No. of don't care bits = 2)
   **No aggregation** since no. of don't care bits does not match.

- Tag = 1001100-*** (R7) would also result in a match but since we do not aggregate further than 3 don't care bits, this tag is not involved in dynamic aggregation.

4) From L1 TLB replacement: 0111011-101 [1908]
   No. of Match = 3
   Matched Tag#1 = 0111011-10* (R1) (No. of don't care bits = 1)
   Matched Tag#2 = 0111011-11* (R5) (No. of don't care bits = 1)
   Matched Tag#3 = 0111011-0** (R6) (No. of don't care bits = 2)
   Matched Tag#1 and #2 **aggregates** to 0111011-1** [1904-1919]
5) From L1 TLB replacement: 0110110-000 [1728]
   No. of Match = 2
   Matched Tag#1 = 0110110-000 (R3) (No. of don't care bits = 0)
   Matched Tag#2 = 0110110-01* (R12) (No. of don't care bits = 1)
   **No aggregation** since no. of don't care bits does not match.

Fig. 9. Missed Look-Up Examples. Entries in L2 VTT is 12. $W = 12$. Only $W$-2 = 10 bits are stored in the VTT. Maximum number of dont care bits allowed due to dynamic aggregation is 3 i.e., a tag in VTT can map to a maximum of 32 entries (e.g., R7). Five examples of missed look-up are given. Three of them are look up due to block coming from memory while rests are due to L1 TLB replaced blocks..

application continues to run, R1 might aggregate with another entry say R3 that has a virtual tag of 1000101-0**. R3 (not shown in the figure) will have four pointers pointing to the first four PTT banks. After merging R1 and R3, the merged entry will have eight pointers pointing to all the PTT banks and will map to a total of 32 translations. The merged entry will have tag as 1000101-*** and will no longer participate in the process of dynamic aggregation as it has reached the maximum allowed don't care bits.

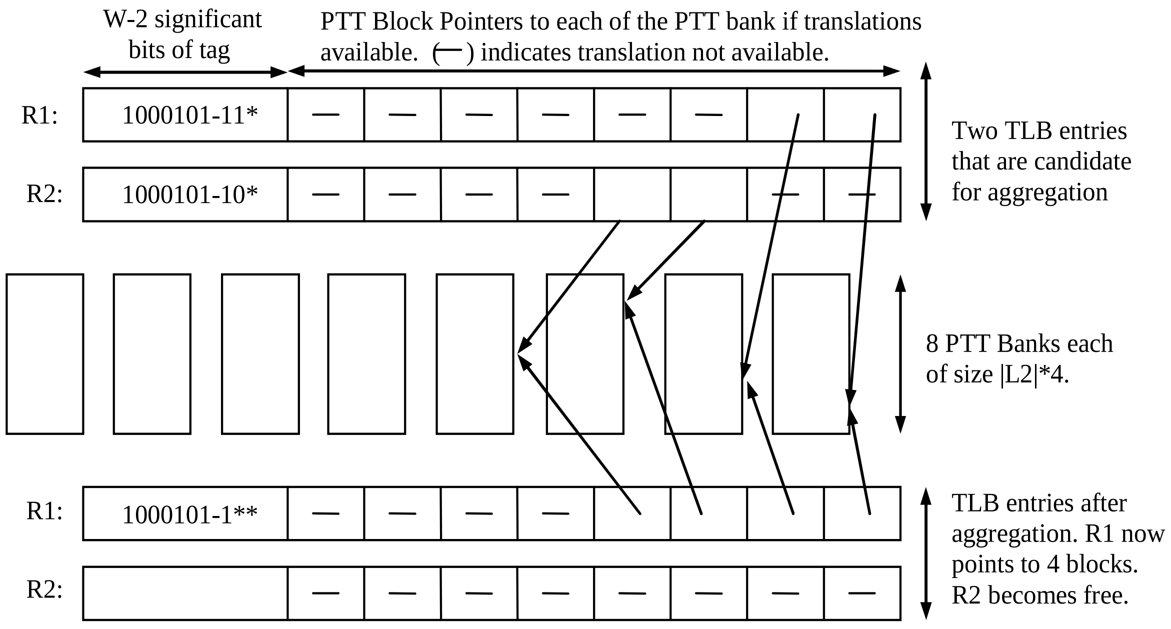

W-2 significant bits of tag
PTT Block Pointers to each of the PTT bank if translations available. (—) indicates translation not available.
R1:
1000101-11*
R2:
1000101-10*
Two TLB entries that are candidate for aggregation
8 PTT Banks each of size |L2|*4.
R1:
1000101-1**
R2:
TLB entries after aggregation. R1 now points to 4 blocks. R2 becomes free.

Fig. 10. Process of merging during dynamic aggregation. ($W$ 2) significant bits are stored in L2 VTT. Maximum number of dont care bits allowed are 3 so a total of 8 PTT banks.

## C. Effect on Overall Miss Rate

For simulations, we considered the sizes of the L1 and L2 VTT as 32 and 128 entries respectively. The size of bulk prefetching is set to four. The maximum number of don't care bits allowed in the simulation is three. Therefore, an entry can map to a maximum of 32 translations. Table III shows the effect of dynamic aggregation on the overall miss rate. For galgel, bzip, mcf and vpr miss rate is reduced by 99.86%, 96.31%, 65.77%, 38.77% respectively compared to a TLB architecture with only static aggregation. As a result the overall miss rates for galgel and bzip become negligible. Twolf, swim, gzip and crafty do not have much scope for further reduction as they have negligible miss rate after static aggregation. Lucas benefit very little from dynamic aggregation. The overall miss rates for galgel and bzip are almost negligible.

Table III. Overall Miss Rate with TCAM enabled L2 TLB architecture supporting dynamic aggregation. Sizes of L1 and L2 VTT are 32 and 128 respectively. The maximum number of TCAM bits for each virtual tag is three (i.e., N = 3).

| Benchmarks | L1 Miss Rate | L2 Hit Rate | Overall Miss Rate |
|---|---|---|---|
| GALGEL | 6.42% | 99.99% | 0.00863% |
| TWOLF | 1.09% | 99.99% | 0.00000826% |
| MCF | 4.43% | 76.98% | 1.02% |
| VPR | 2.77% | 96.17% | 0.106% |
| LUCAS | 1.05% | 6.56% | 0.981% |
| SWIM | 0.04% | 8.89% | 0.036% |
| BZIP | 0.56% | 99.74% | 0.0014% |
| GZIP | 0.00188% | 6.4% | 0.00176% |
| CRAFTY | 0.77% | 99.98% | 0.00000842% |

## D. Temporal Property of Dynamic Aggregation

Next we look at the accuracy of bulk prefetching (i.e., out of four translations brought together from memory how many of them are looked up before they are replaced). We expected to see a significant increase in accuracy due to the improvement in miss rate. However, looking at Figure 8, we see there is not much difference in accuracy between static and dynamic aggregation. The first four rows correspond to percentage of blocks replaced from L2 TLB with only one, two, three and all the four translations having been looked up. It is important to note that total tags replaced from L2 TLB are not same. Table IV gives the total number of block tags replaced from L2 TLB during simulation period. In fact, the values in Table VI and Table VII are also percentage of the value in Table V. Twolf is not considered in this experiment as the number of miss is negligible after static aggregation only and all the entries of L2 TLB is not used up; therefore no replacement takes place. Two columns (static and dynamic) for each benchmark denote the prefetch accuracy (in percentage) with static and dynamic aggregation respectively.

Table IV. Prefetch Accuracy for two designs: static aggregation and dynamic aggregation.

| | MCF | | GALGEL | | SWIM | | VPR | | LUCAS | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Static | Dynamic | Static | Dynamic | Static | Dynamic | Static | Dynamic | Static | Dynamic |
| 1 | 71.6 | 79.2 | 1.18 | 96.2 | 16.4 | 13.64 | 65.65 | 62.64 | 94.72 | 94.45 |
| 2 | 17.97 | 8.9 | 94.1 | 1.785 | 19.1 | 15.76 | 25.3 | 26.44 | 3.11 | 3.36 |
| 3 | 6.1 | 4.1 | 4.66 | 0.545 | 0.72 | 0.207 | 6.7 | 7.6 | 0.013 | 0.004 |
| 4 | 4.33 | 7.75 | 0.067 | 1.466 | 63.75 | 70.38 | 2.33 | 3.32 | 2.15 | 2.19 |

Table V. Total Number of L2 TLB replacement for two designs: static and dynamic aggregation

| | Static | Dynamic |
|---|---|---|
| MCF | 5.68 * $10^7$ | 1.94* $10^7$ |
| GALGEL | 1.145*$10^8$ | 1.54*$10^5$ |
| SWIM | 5.58*$10^5$ | 5.07*$10^5$ |
| VPR | 3.12*$10^6$ | 1.90*$10^6$ |
| LUCAS | 9.17*$10^6$ | 8.89*$10^6$ |

We use the term *block occupancy* to refer to the duration between a block being fetched from main-memory and the block being evicted out of L2 TLB. Using SimpleScalar, we measured this in terms of the total number of data-TLB look ups occurring in that duration. We noticed a considerable difference in the block occupancy between static and dynamic aggregation. Table VI gives the percentage of blocks with different block occupancy ranges for mcf, galgel and swim. We consider only these benchmarks as they benefit more from the temporal property of dynamic aggregation.

Next, we measure how many times a block is replaced from L1 TLB without being replaced from L2 TLB, which we refer to as *L1-block-replacement-count (LBRC)*. When a block is fetched from memory, it is placed in the L1 as well as the L2 TLB. After some time, it is removed from L1 due to the LRU policy but remains in L2 until it is referenced again, upon which it is placed back into L1. Every time this happens, LBRC for the block is incremented. This process continues until the block gets evicted out from the L2 TLB.

Table VII shows the percentage of blocks with various LBRC ranges. We see

Table VI. Percentage of blocks with different ranges of block occupancy.

| Block | MCF | | GALGEL | | SWIM | |
|---|---|---|---|---|---|---|
| Occupancy | Static | Dynamic | Static | Dynamic | Static | Dynamic |
| $< 1000$ | 44.25 | 40.57 | 0.35 | 0 | 27.37 | 0 |
| $< 10000$ | 52.02 | 41.11 | 99.15 | 1.4 | 0.089 | 18.14 |
| $< 50000$ | 2.475 | 10.26 | 0.29 | 95.7 | 0.537 | 0.42 |
| $< 100000$ | 0.904 | 2.16 | 0.016 | 0.1 | 0.586 | 0.33 |
| $< 1000000$ | 0.308 | 5.08 | 0.185 | 0.17 | 71.4 | 6.83 |
| $> 1000000$ | 0.032 | 0.807 | 0.002 | 2.6 | 0.015 | 74.3 |

a higher percentage of blocks having larger LBRC when using dynamic aggregation versus static aggregation for three benchmarks. These behaviors contribute to the reduction in overall miss rate. The increase in LBRC is attributed to both benchmark characteristics and the increase in block occupancy.

From the above study, we see many applications look up the same pages repeatedly with different time-gaps in between references. If the gap is very small, the mapping for the page will remain in L1 TLB; however if it is too large, it will be replaced from L2 TLB. If the gap is between these two extremes, the LBRC will depend on the block's occupancy. As the block occupancy on average is greater in the case of dynamic aggregation, the LBRC increases result in a better hit rate.

Thus, we see that TCAM enabled dynamic aggregation not only helps in exploiting spatial locality by storing more translations for the same size of L2 VTT but also helps in exploiting temporal locality by increasing the block occupancy.

Table VII. Percentage of blocks with various ranges of LBRC.

| LBRC | MCF | | GALGEL | | SWIM | |
|---|---|---|---|---|---|---|
| | Static | Dynamic | Static | Dynamic | Static | Dynamic |
| 1 | 72.01 | 76.16 | 99.956 | 97.3 | 100 | 96.74 |
| 2-5 | 26.9 | 18.1 | 0.043 | 0.40 | 0 | 2.91 |
| 6-10 | 0.986 | 1.95 | 0 | 0.025 | 0 | 0.0002 |
| 11-50 | 0.075 | 2.18 | 0 | 0.368 | 0 | 0.342 |
| 51-100 | 0.00083 | 0.92 | 0 | 0.014 | 0 | 0 |
| 101-500 | 0.0027 | 0.68 | 0.00035 | 0.737 | 0 | 0 |
| 501-1000 | 0 | 0.044 | 0.00024 | 0.28 | 0 | 0 |
| $>$ 1000 | 0 | 0.015 | 0 | 0.83 | 0 | 0 |

# CHAPTER VII

# TIMING ANALYSIS

## A. Average Cost of Translation

In this section, we compare the average cost of translations between a CAM based two-level TLB and two-level TLB as presented in this paper. The overall cost of translation for any two-level TLB architecture is given by:

$$L1\_AccessTime + L1\_MissRate \times [L2\_AccessTime + L2\_MissRate \times MissPenalty]$$

The L1 TLB for both cases are identical in terms of access time and miss rate. Both are of the same size and implemented using CAM cells. In this section, we do not include static aggregation in the TLB so that the miss rate is same for both cases.

We assume that the access time for the L2 VTT for both cases is almost the same due to the following reasons. Arsovski *et.al.* [27] show TCAM access time to be almost the same as CAM access time. We are using only $N$ TCAM cells per virtual tag in the L2 VTT; the remaining bits are stored using CAM cells. Further, the size of TCAM supported L2 VTT is half the size of the CAM based L2 VTT for all the simulations. Therefore, the only variable parameters involved in cost of translation for the two schemes are the L2 TLB miss rate and their respective miss penalties.

Therefore, the ratio of the variable parameters determining the cost of translation between the two schemes is given by:

$$L2\_MissRate\;Ratio \times Overhead\;Ratio$$

where

$$L2\_MissRate\;Ratio = \frac{L2\_MissRate\;(TCAM\;along\;with\;CAM\;based\;L2\;TLB)}{L2\_MissRate\;(CAM\;based\;L2\;TLB)}$$

and

$$Overhead\ Ratio = \frac{Total\ Penalty\ (bulk\ prefetching\ and\ dynamic\ aggregation)}{Penalty\ (fetching\ one\ entry)}$$

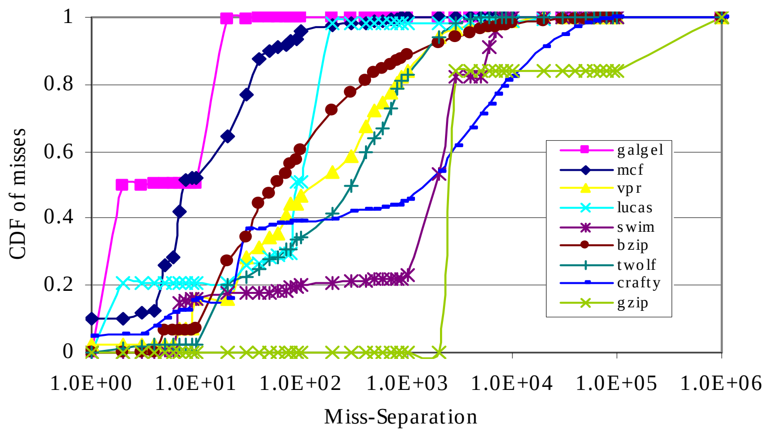
From Table VIII, we see that for all the benchmarks except lucas the ratio of the variable factor is less than one. Therefore, the average cost of translation is lower for our scheme in all these benchmarks. The reduction in miss rate is able to hide the overhead of bulk prefetching and dynamic aggregation.

Table VIII. Ratio of the variable parameters determining cost of translating a virtual address between the two schemes for various overhead ratios.

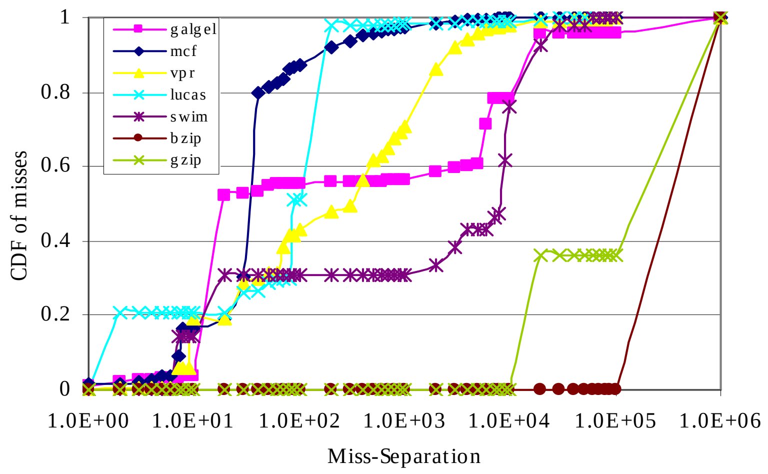
| Benchmarks | L2 MissRate | OVERHEAD RATIO | | | | |
|---|---|---|---|---|---|---|
| | Ratio | 1.1 | 1.2 | 1.3 | 1.4 | 1.5 |
| GALGEL | 0.000663 | 0.00073 | 0.00080 | 0.00086 | 0.00093 | 0.000995 |
| TWOLF | 0.0000177 | 0.0000195 | 0.0000213 | 0.000023 | 0.000248 | 0.0000265 |
| MCF | 0.2484 | 0.273 | 0.298 | 0.323 | 0.348 | 0.373 |
| VPR | 0.2581 | 0.284 | 0.31 | 0.3356 | 0.3614 | 0.387 |
| LUCAS | 0.8821 | 0.977 | 1.06 | 1.15 | 1.24 | 1.33 |
| SWIM | 0.2911 | 0.32 | 0.349 | 0.378 | 0.407 | 0.436 |
| BZIP | 0.0098 | 0.01077 | 0.0117 | 0.0127 | 0.0137 | 0.0147 |
| GZIP | 0.237 | 0.261 | 0.284 | 0.308 | 0.332 | 0.355 |
| CRAFTY | 0.00019 | 0.00022 | 0.00023 | 0.00025 | 0.00027 | 0.00029 |

## B. Prefetching Window

*Miss-separation* has been defined as the number of instructions executed between two consecutive TLB misses [24], and it provides an insight into the timing constraints under which prefetching mechanisms need to operate. If this time interval is too small, prefetching based on complex prediction schemes may not be able to hide the miss latency. Figure 11(a) gives the CDF of temporal miss-separation (i.e., y-axis shows the fraction of misses that are separated by at most a given value in x-axis) for the two-level CAM based TLB. For galgel and mcf, the prefetching windows for more than half of the total misses are 5 and 8 instructions respectively. These prefetching windows increase to 20 and 50 respectively when using dynamic aggregation as shown in Figure 11(b). The frequency of prefetching in galgel is low as the miss-rate is negligible due to dynamic aggregation as compared to the CAM-based two level TLB. The CDF plot is flat between 20 and 1000 signifying that the remaining 50% of misses have miss-separation of more than 1000 instructions. It was observed that CDF curves for all the benchmarks shifted to higher values of miss-separations. Due to dynamic aggregation, all the misses in gzip and bzip occur with miss-separation larger than 10,000 and 100,000 respectively resulting in a flat curve. The CDF plot for twolf and crafty are not shown in Figure 11(b) as there was no replacement of tags from L2 TLB for these two benchmarks (i.e., some of the 128 entries in L2 TLB went unutilized). Thus, the increased miss-separations provide a larger window for bulk prefetching. Moreover, the overhead of prefetching four entries on each TLB miss can be reduced by using *Clustered Page Tables* as introduced in [28].



(a)



(b)

Fig. 11. Cumulative Density Function of miss-separation. (a) two-level CAM based TLB. (b) two-level TLB proposed in this paper supporting dynamic aggregation.

# CHAPTER VIII

# SUPPORT FOR VARIABLE SIZED PAGES

Some operating systems use different page sizes (e.g., 64KB, 256KB or 1MB sized pages) for certain page mappings such as kernel data structures and frame buffers. Most modern processor implementations include support for multiple page sizes in their TLBs. TCAM based implementation of TLBs can also support variable page sizes with a single VTT entry by varying the number of don't care bits. Typical TLB implementations that support multiple page sizes require that the pages are always a power-of-two multiple of base page size and both the virtual and physical base addresses are aligned to the page size. A TCAM based implementation of TLBs can support page sizes of any arbitrary multiple of base page size by appropriately setting the don't care bits. This enables TCAM based implementations to adjust to the dynamic behavior of applications and the system environment thus providing better performance without increasing operating system complexity and associated overhead of supporting superpages. We intend to evaluate effectiveness of TCAM based implementation of variable page sizes against other such implementations in subsequent work.

# CHAPTER IX

# CONCLUSION AND FUTURE WORK

We proposed a two-level TLB architecture that improves the TLB reach with the same number of virtual tag stored in the TLB. Application programs seem to benefit significantly from bulk prefetching and static aggregation. Using TCAM cells in the L2 TLB further improves the hit rate due to dynamic aggregation. Support for large sized pages can be easily accommodated due to the use of TCAM cells. Table IX summarizes the results showing the reduction in miss rates due to the various schemes proposed in this paper compared to the existing prevalent 2-level TLB. It is noteworthy to mention that four out of nine benchmarks achieve more than 99% reduction in their miss rates.

As future work, we would like to investigate the impact of the proposed design on chip area and power consumption. We would like to extend the timing analysis using the CACTI model. We intend to study the effectiveness of the proposed scheme under multi-programming, multi-threading and run-time environments that frequently change virtual to physical memory mappings (such as dynamic linking, and run-time environments like Java).

Table IX. Overall Miss Rates for all the three schemes: (1) CAM based two-level TLB with 32 and 256 entries. (2) CAM based two-level TLB that supports bulk prefetching and static aggregation with 32 and 128 entries. (3) 2-level TLB with CAM based L1 and CAM+TCAM based L2 that supports dynamic aggregation in addition to bulk prefetching and static aggregation with 32 and 128 entries. Also shown in the table is the percentage miss reduction of scheme (2) and (3) over scheme (1).

| Benchmarks | CAM based 2-level TLB L1=32,L2= 256 | Bulk Prefetching + Static Aggregation L1 =32, L2 = 128 | | Bulk Prefetching + Static + Dynamic Aggregation (CAM + TCAM based L2 TLB) L1 = 32, L2 = 128 | |
|---|---|---|---|---|---|
| | Overall Miss Rate | Overall Miss Rate | Miss Reduction | Overall Miss Rate | Miss Reduction |
| GALGEL | 13.01% | 6.39% | 50.88% | 0.00863% | 99.93% |
| TWOLF | 0.466% | 0.00000826% | 99.99% | 0.00000826% | 99.99% |
| MCF | 4.105% | 2.98% | 27.4% | 1.02% | 75.15% |
| VPR | 0.259% | 0.173% | 33.2% | 0.106% | 59.07% |
| LUCAS | 1.10% | 1.01% | 8.18% | 0.981% | 10.81% |
| SWIM | 0.126% | 0.04% | 68.25% | 0.036% | 71.4% |
| BZIP | 0.14% | 0.038% | 72.8% | 0.0014% | 99% |
| GZIP | 0.0073% | 0.0018% | 75.3% | 0.00176% | 75.89% |
| CRAFTY | 0.043% | 0.00000842 | 99.98% | 0.00000842 | 99.98% |

## REFERENCES

[1] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska, "The interaction of architecture and operating system design," in *International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS*, Santa Clara, CA (USA), 1991, pp. 108–120.

[2] K. Bala, M. F. Kaashoek, and W. E. Weihl, "Software prefetching and caching for translation lookaside buffers," in *Proc. of the Usenix Symposium on Operating Systems Design and Implementation*, Monterey, California, Nov 1994, pp. 243–253.

[3] J. B. Chen, A. Borg, and N. P. Jouppi, "A simulation based study of TLB performance," in *Proc. of the 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, 1992, pp. 114–123.

[4] M. Talluri and M. D. Hill, "Surpassing the tlb performance of superpages with less operating system support," in *Proc. of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, California, USA, December 1994, vol. 28, pp. 171–182.

[5] Z. Fang, L. Zhang, J. B. Carter, W. C. Hsieh, and S. A. McKee, "Reevaluating online superpage promotion with hardware support," in *International Symposium on High Performance for Computer Architecture*, Nuevo Leone, Mexico, 2001, pp. 63–72.

[6] C. H. Park, J. Chung, B. H. Seong, Y. Roh, and D. Park, "Boosting superpage utilization with the shadow memory and the partial-subblock tlb," in *Proc. of*

*the 14th International Conference on Supercomputing*, Santa Fe, New Mexico, USA, 2000, pp. 187–195, ACM Press.

[7] M. Swanson, L. Stoller, and J. Carter, "Increasing tlb reach using superpages backed by shadow memory," in *Proc. of the 25th Annual International Symposium on Computer Architecture*, Barcelona, Spain, 1998, pp. 204–213, IEEE Computer Society.

[8] A. Saulsbury, F. Dahlgren, and P. Stenstrom, "Recency-based tlb preloading," in *Proc. of the 27th annual International Symposium on Computer Architecture*, Vancouver, Canada, 2000, vol. 28, pp. 117–127, ACM Press.

[9] G. B. Kandiraju and A. Sivasubramaniam, "Going the distance for tlb prefetching: an application-driven study," in *Proc. of the 29th Annual International Symposium on Computer Architecture*, Anchorage, Alaska, 2002, pp. 195–206, IEEE Computer Society.

[10] T. Austin and G. S. Sohi, "High-bandwidth address translation for multiple-issue processors," in *Proc. of the 23rd Annual International Symposium on Computer Architecture*, Pennsylvania, USA, 1996, pp. 158–167.

[11] J.P. Wade and C.G. Sodini, "A ternary content addressable search engine," *IEEE Journal of Solid-State Circuits*, August 1989.

[12] Y. A. Khalidi, M. Talluri, M. N. Nelson, and D. Williams, "Virtual memory support for multiple page sizes," in *Proc. of the Fourth IEEE Workshop on Workstation Operating Systems (WWOS)*, Napa, California, USA, Oct 1993, pp. 104–109.

[13] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating

system support for superpages," in *Symposium on Operating Systems Design and Implementation OSDI*, Boston, USA, 2002, pp. 89–104, ACM Press.

[14] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad, "Reducing TLB and memory overhead using online superpage promotion," in *Proc. of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, 1995, pp. 176–187.

[15] J. B. Carter, W. C. Hsieh, L. Stoller, M. R. Swanson, L. Zhang, E. Brunvand, A. Davis, C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama, "Impulse: Building a smarter memory controller," in *International Symposium on High Performance for Computer Architecture*, Orlando, FL, USA, 1999, pp. 70–79.

[16] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The stanford flash multiprocessor," in *Proc. of the 25th Annual International Symposium on Computer Architecture*, Barcelona, Spain, 1998, pp. 485–496, ACM Press.

[17] M. Oskin, F. T. Chong, and T. Sherwood, "Active pages: A computation model for intelligent memory," in *Proc. of the 25th annual international symposium on Computer architecture*, Barcelona, Spain, 1998, pp. 192–203.

[18] S. P. VanderWiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Computing Surveys*, vol. 32, no. 2, pp. 174–199, 2000.

[19] F. Dahlgren, M. Dubois, and P. Stenstrom, "Fixed and adaptive sequential prefetching in shared memory multiprocessor," in *International Conference on Parallel Processing*, Syracuse, New York, USA, Aug 1993, pp. 56–63.

[20] T. Chen and J. Baer, "Effective hardware based data prefetching for high performance processors," *IEEE Transactions on Computers*, pp. 609–623, May 1995.

[21] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *Proc. of the 25th Annual International Symposium on Microarchitecture*, Portland, Oregon, United States, 1992, pp. 102–110, IEEE Computer Society Press.

[22] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 121–133, 1999.

[23] J. S. Park and G. S. Ahn, "A software-controlled prefetching mechanism for software-managed tlbs," *Microprocessors and Microprogramming*, vol. 41, no. 2, pp. 121–136, 1995.

[24] G. B. Kandiraju and A. Sivasubramaniam, "Characterizing the d-tlb behavior of spec cpu2000 benchmarks," in *Proc. of the International Conference on Measurement and Modeling of Computer Systems SIGMETRICS*, Marina Del Rey, California, 2002, pp. 129–139, ACM Press.

[25] S.P.E.C Corporation, http://www.spec.org.

[26] D. Burger and T. Austin, SimpleScalar Toolset, Version 3.0, http://www.simplescalar.com.

[27] I. Arsovski, T. Chandler, and A. Sheikholeslami, "A ternary content-addressable memory (tcam) based on 4t static storage and including a current-race sensing scheme," *IEEE Journal Of Solid-State Circuits*, vol. 38, no. 1, pp. 121–133, January 2003.

[28] M. Talluri, M. D. Hill, and Y. A. Khalidi, “A new page table for 64-bit address spaces,” in *Symposium on Operating Systems Principles*, Colorado, USA, 1995, pp. 184–200.

# VITA

Anuj Kumar was born in Madhubani, India on the 14th of October, 1977. He completed his Bachelor of Technology degree in electrical engineering from the Indian Institute of Technology, Kanpur, India in May 1999. He subsequently worked for three years as a software engineer before starting his graduate studies as a computer engineering major at Texas A&M University in the fall of 2002. His research interests are in field of networking and computer architecture. He can be reached at the following email address: kumar.anuj@gmail.com

Permanent Address:

F-27, Sector 3,

Dhurwa,

Ranchi 834004,

Jharkand,

India.

The typist for this thesis was Anuj Kumar.