CHARACTERIZATION OF TLB AND PAGE ALLOCATION BEHAVIOR ON MODERN PROCESSORS

BY VISWANATHAN VAIDYANATHAN

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Computer Science

Written under the direction of

Dr.Abhishek Bhattacharjee

and approved by

_____

_____

_____

New Brunswick, New Jersey

October, 2012

ABSTRACT OF THE THESIS

Characterization of TLB and Page Allocation Behavior on Modern Processors

by Viswanathan Vaidyanathan

Thesis Director: Dr.Abhishek Bhattacharjee

Virtual memory support is prevalent in most modern processors and is facilitated through Translation Lookaside Buffers (TLBs) which play a major role in the overall system performance. TLB misses are costly since they require multiple high latency memory references to walk the page table and locate the desired Virtual Page Number (VPN) - Physical Page Number (PPN) mapping. This study improves TLB hit rates by taking advantage of any contiguity present in the pages allocated by the Operating System (OS). By contiguity we refer to cases where consecutive VPNs are mapped to consecutive PPNs. Traditionally, OSs use large or superpages to collapse hundreds of such contiguous entries, thereby using one TLB entry to represent them rather than hundreds of entries they would normally require. Unfortunately due to implementation complexities superpaging has not been universally successful in reducing TLB pressure. We show, however, that even without explicit superpaging, various OS virtual memory allocation activities lead to intermediate levels of contiguity that may be exploited to coalesce

TLB entries and significantly improve hit rates. We verify the presence of contiguity by running benchmarks on a real system and checking the page allocations of the OS. The OS page allocation schemes depend on memory pressure and memory defragmentation daemons. Further, we find an average contiguity of 30 pages over all the benchmarks and configurations with superpaging turned on and about 10 with superpaging turned off. To verify the performance of a Coalesced TLB we have implemented a fully associative TLB with variable size and Least Recently Used (LRU) replacement policy. Our results show an average hit rate improvement of 25% by adding an 8 - 16 entry fully associative Coalesced TLB. The Coalesced TLB further needs no complex hardware to implement, hence providing to a low cost means to reduce miss rates.

Acknowledgements

This study was possible due to the guidance provided by my advisor Dr. Abhishek Bhattacharjee. I would like to thank Dr. Richardo Bianchini and Dr. Thu Ngyuen for supervising my thesis defense. Further, I acknowledge the contributions of my colleague, Mr. Binh Pham, in helping me setup infrastructure and discussing research ideas for this study. I would like to express my gratitude to Mr. William Katsak for helping me understand the internals of Linux. Finally, I would like to thank the Department of Computer Science, Rutgers University for providing the necessary infrastructure to run my experiments on.

Dedication

I would like to dedicate this piece of work to my parents, who have been very supportive of my decision to pursue this field of study. I would also like to dedicate the major infrastructure work done in this study to the RUArch lab and hope it comes in handy for future experiments.

Table of Contents

Abstract ii

Acknowledgements iv

Dedication v

List of Figures ix

List of Tables xi

1. Introduction 1

2. Background 3

2.1. Translation Lookaside Buffers 3

2.2. Page tables 4

2.2.1. Page Table Walk 4

2.3. OS page allocation schemes 5

2.3.1. Basic page sizes 5

2.3.2. Transparent Hugepage support 6

2.3.3. VM Allocation - Buddy Allocator 6

2.3.4. Defragmentation 7

2.4. Related work 8

2.5. Our approach 9

3. Contiguity - Real System Experiments 10

3.1. System Configuration 10

3.1.1. Benchmarks 11

3.2. Methodology 11
3.2.1. Memhog 12
3.2.2. Benchmark Instrumentation 13
3.2.3. Results 13
3.2.4. Average Contiguity Results 14
3.2.5. Observations 17

4. Simulation Infrastructure 18
4.1. Simics 18
4.2. FeS2 18
4.2.1. Traditional FeS2-Simics Interaction 19
4.3. Trace Driven FeS2 19
4.3.1. Simics trace generation module 20
4.3.2. Trace Processor 21
4.3.3. Rewinds 21
4.3.4. Future 21

5. Coalesced TLB 22
5.1. Coalescing Operation 22
5.2. Lookup In Coalesced TLB 24
5.3. Benefits 24
5.4. Performance Evaluation 24
5.4.1. Experiment setup 24
5.4.2. Results: Recency plot 25
5.4.3. Recency Plots - THS On 26
5.4.4. Recency Plots - THS off 28
5.4.5. Observations 28
5.5. Conclusion 30

6. Discussions and Future 31

6.1. Replacement Policy 31

6.2. Prefetching from Cache line 31

6.3. Coalescing entries within TLB 31

6.4. Small fully associative buffers 32

References 33

List of Figures

2.1. Virtual address partitioned into range of bits used to perform the page walk 4

2.2. Page walk for x86_64 bit architectures 5

2.3. Working of a Buddy allocator 7

2.4. State of memory before defragmentation 8

2.5. Memory block movement 8

2.6. State of memory after defragmentation 8

3.1. Weighted average contiguity with no memory hog, Part 1 14

3.2. Weighted average contiguity with no memory hog, Part 2 14

3.3. Weighted average contiguity with 25% memory hog, Part 1 . . . 15

3.4. Weighted average contiguity with 25% memory hog, Part 2 . . . 15

3.5. Weighted average contiguity with 50% memory hog , Part 1 . . . 16

3.6. Weighted average contiguity with 50% memory hog , Part 2 . . . 16

5.1. Contents of a normal TLB 22

5.2. Contents of the corresponding Coalesced TLB 23

5.3. Recency plot of 473.astar with THS on 26

5.4. Recency plot of 401.bzip2 with THS on 26

5.5. Recency plot of 007.fasta_prot with THS on 26

5.6. Recency plot of 416.gamess with THS on 26

5.7. Recency plot of 459.GemsFDTD with THS on 26

5.8. Recency plot of 445.gobmk with THS on 26

5.9. Recency plot of 429.mcf with THS on 27

5.10. Recency plot of 001.mummer with THS on 27
5.11. Recency plot of 471.omnetpp with THS on 27
5.12. Recency plot of 453.povray with THS on 27
5.13. Recency plot of 458.sjeng with THS on 27
5.14. Recency plot of 002.tigr with THS on 27
5.15. Recency plot of 471.astar with THS off 28
5.16. Recency plot of 401.bzip2 with THS off 28
5.17. Recency plot of 007.fasta_prot with THS off 28
5.18. Recency plot of 416.gamess with THS off 28
5.19. Recency plot of 459.GemsFDTD with THS off 28
5.20. Recency plot of 445.gobmk with THS off 28
5.21. Recency plot of 429.mcf with THS off 29
5.22. Recency plot of 001.mumer with THS off 29
5.23. Recency plot of 471.povray with THS off 29
5.24. Recency plot of 473.omnetpp with THS off 29

List of Tables

3.1. System Configuration 11

3.2. Benchmarks used in the study 11

3.3. Benchmark Input 12

3.4. Various scenarios tested with the real system infrastructure 13

5.1. Mappings added to the Coalesced TLB in Figure 5.2. A study of the coalescing behavior. 23

Chapter 1
Introduction

Most modern processors employ virtual memory to automate the management of multiple layers of memory hierarchy. A key hardware structure to manage virtual memory is the Translation Lookaside Buffer (TLB). TLB performance is critical due to long miss penalites [13, 24, 36, 31]. Despite hardware and software enhancement studies such as size, associativity, multi-level hierarchies and prefetching, TLB misses can degrade performance by up to 50% [25, 32, 12, 11, 10, 9].

A key technique to improve TLB hit rates is superpaging. A number of past studies tried allocating contiguous physical pages to contiguous virtual pages, thereby constructing large pages [35]. The goal is to use one TLB entry to represent these pages rather than the hundreds of entries that would otherwise be required. Superpaging has a number of implementation complexities that have obstructed their widespread adoption. Among other problems, they require a huge level of OS mandated contiguity which can eventually lead to problems in page allocation leading to memory trashing.

This thesis is the first to show that regardless of superpaging, intermediate levels of contiguity are generated naturally and transparently by the OS due to the buddy allocator [27], memory defragmentation daemons [14] and system load. We show how to improve TLB hit rates by exploiting this contiguity. Two entries are contiguous if their VPNs and PPNs are contiguous and their permission bits are the same. Here are our major contributions

- Our work quantifies the amount of page contiguity present in workloads

when run on a real system. This study uses Spec CPU2006 [16] and BioBench [4] workloads. On an average we find a contiguity of 30 pages when superpaging is turned on and 10 pages when superpaging is turned off.

- Page allocation patterns vary due to memory pressure (also known as system load), superpaging and the defragmentation daemon. We determine the effect of these attributes on overall contiguity.

- Our second experiment exploits the contiguity mentioned above to create a Coalesced TLB. A large least recently used (LRU) list is used to characterize the performance of a range of fully associative TLBs. On an average close to 25% misses are eliminated with the addition of an 8 entry fully associative TLB. This is explained in detail in chapter 5.4.

The thesis is organized as follows. The second chapter gives some background on the topic. The next chapter talks about the real system experiments run for this study. Chapter 4 talks about the simulation infrastructure used to study the Coalesced TLB performance. As part of this study we have extended FeS2 [38], a cycle-accurate x86 simulator, to read from traces generated by Simics [39] to improve simulation run times. Chapter 5 introduces the Coalesced TLB and explains how it works. Performance improvement results are then discussed. The last chapter talks about some future extensions to the idea presented in this paper.

Chapter 2
Background

This chapter provides background on the various components involved in supporting virtual memory on modern systems. Later in the chapter, OS support for contiguous pages is discussed.

2.1 Translation Lookaside Buffers

Due to their performance criticality, a range of TLB design attributes such as size, placement, lookup, associativity and multiple hierarchies have been studied [25, 32, 12, 11, 10, 9]. Unfortunately a TLB is restricted by power and die area. Typical L1 data TLBs in Intel's recent processors are 4 way set associative and have from 32 - 64 entries. The corresponding L2 data TLBs are 4 way set associative and have 256 - 512 entries [26]. There are seperate L1 data and instruction TLBs while the L2 is unified. This study focuses on data TLB performance, since instruction TLBs already have high hit rates [10, 25]. TLB miss handling is accomplished through either hardware or software managed methods. Hardware managed methods involve walking the page table using hardware state machines. Some studies have shown the performance benefits of this approach [22] with miss latencies ranging from 10 to 50 cycles [23, 24]. Since this approach fixes the size of the page table, RISC architectures often use a software managed TLB [21, 36]. In this approach a TLB miss invokes an interrupt which causes the OS to walk a software managed page table to refill the TLB. Around 10 to 100 OS instructions need to be executed on every miss and this causes further misses in the memory

hierarchy. Studies have shown that software managed TLB misses take up to hundreds of cycles to get serviced [21, 22]. Hence improving TLB performance is critical.

2.2 Page tables

We now detail page walk mechanisms and associated acceleration hardware. Though different architectures approach this problem differently we focus here on x86 architectures since they are widespread and we use them for our studies.

2.2.1 Page Table Walk

X86 processors maintain the page table in a radix tree data structure, accessible by splitting the virtual address into the following fields as illustrated in Figure 2.1:

- Page Offset (bits 0 - 11)
- L1 Index/ Page Table Level (bits 12 - 20)
- L2 Index/ Page Directory Level (bits 21 - 29)
- L3 Index/ Page Directory Pointer (bits 30 - 38)
- L4 Index/ Page Map Level 4 (bits 39 - 47)
- Sign Extension (bits 48 - 63)

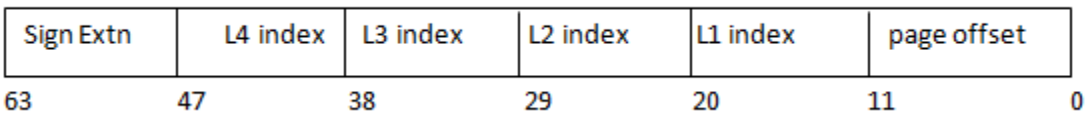

Figure 2.1: Virtual address partitioned into range of bits used to perform the page walk

.

Page tables are walked using hardware on a set of page tables known as the Page Map Level 4 (PML4), the Page Directory Pointer (PDP), the Page Directory (PD) and the Page Table (PT) . We will refer to these tables as the L4, L3, L2 and the L1 tables respectively. Figure 2.2 illustrates a page walk. Moving from L4 to L1, each level successively maps a smaller range of virtual addresses (VAs), with 512GB, 1GB, 2MB and 4KB being the respective sizes. The page walk iteratively uses the physical address of each level's base combined with the 9 index bits obtained from the VA to locate page table entries. These entries store the physical address of the next level. The L1 entry provides the final VA - PA address translation [6, 8].

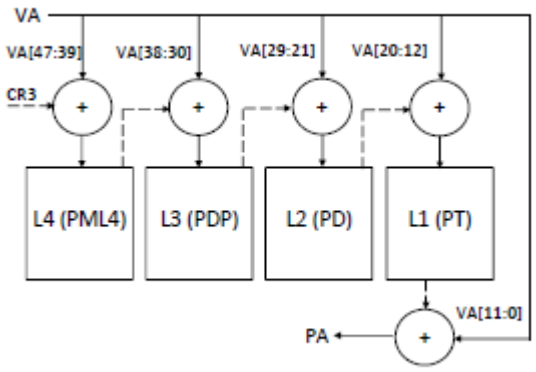

Figure 2.2: Page walk for x86_64 bit architectures

.

2.3 OS page allocation schemes

2.3.1 Basic page sizes

Linux distributions support 4KB, 2MB and 1GB pages. Of these the default is a 4KB page. Some applications use large contiguous chunks of memory and hence allocating a 2MB page can be beneficial. The first form of large page support, called Hugepages, allowed 2MB page allocations if a kernel parameter

nr_hugepages was set explicitly for processes [2]. Recent developments have introduced Transparent Hugepage Support (THS) [5] which allocates 2MB pages to processes without an administrator having to explicitly set them. 1GB pages are not yet supported by the Linux OS.

2.3.2 Transparent Hugepage support

Even though hugepages are advantageous, it requires an administrator to explicitly allocate them to a process. A more recent development, called THS [5], is instrumental in supporting hugepages seamlessly. Kernel *2.6.38* and above support this feature which can be turned on by setting */sys/kernel/mm/ transparent_hugepage/enabled* to *'always'*. When enabled, the OS tries to allocate a 2MB page if one is available. When it has no 2MB page available it attempts to defragment the memory, through a defrag daemon and allocates a new hugepage, if one is obtained. Otherwise the OS falls back to allocating 4KB pages.

2.3.3 VM Allocation - Buddy Allocator

Virtual memory page allocation is done through the Buddy allocator algorithm [27], which divides free memory into a partition that satisfies the requested size[1]. The most common form of buddy allocator is the binary buddy allocator where memory blocks have sizes proportional to powers of 2. Figure 2.3 shows how the buddy allocator works on a system with block sizes ranging from 4K - 32K.The colored portions represent allocated entries while the transparent portions represent free entries. Steps 1 - to end denote memory allocation requests to this chunk of memory starting with stage 1 showing the initial state, a free 32K chunk. The first memory request *Mem1* is for a 7K chunk and the buddy allocator provides this by splitting the free space into smaller chunks as shown by Steps 2.1 and 2.2. In step 2.1, the two 16K entries are buddies. In step 3 *Mem2* requires 13K,

which can be only satisfied by a chunk having 4 blocks. Since *Mem1* splits the 4 block chunk on the left side, its buddy is allocated for *Mem2*. In step 4 *Mem3* requires 4K and one of the 4K blocks satisfies this request. These block sizes make address computation simple since buddies are aligned on memory address boundaries, which are in turn powers of 2. The hardware sets the size of the smallest block based on the page size, which is typically 4KB in x86 processors [20]. Larger block sizes are power-of-two multiples of this size.

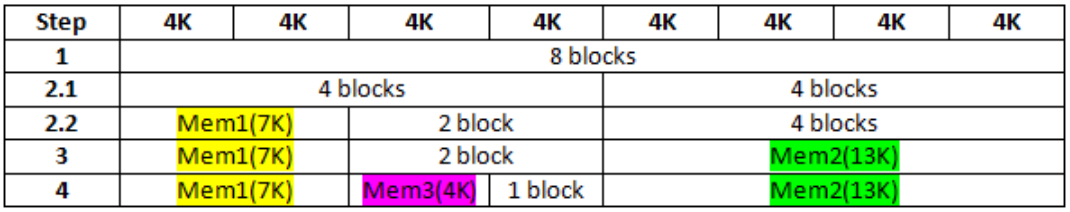

Figure 2.3: Working of a Buddy allocator

.

2.3.4 Defragmentation

Memory gets fragmented over a period of time. A defragmentation daemon compacts such fragmented memory blocks into chunks of allocated blocks and thereby frees up a large chunk of contiguous memory blocks, which can then be used by the buddy allocator for hugepages. This is shown in in figures 2.4 - 2.6. The daemon builds up two lists of memory blocks as shown in figure 2.5 and moves them if sufficient space exists. The daemon runs this as two separate algorithms, one starting at the bottom of the memory range building a list of allocated blocks while the other starts at the top building a list of free blocks. To facilitate page movement, page migration code [15] is invoked. The defragmentation daemon can be enabled by setting the variable */sys/kernel/mm/transparent_hugepage/defrag* to *'always'* [3].

Figure 2.4: State of memory before defragmentation

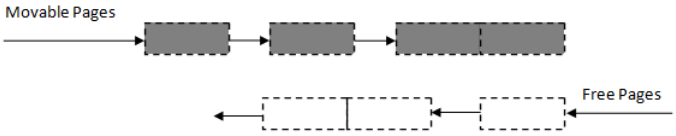

Figure 2.5: Memory block movement

Figure 2.6: State of memory after defragmentation

2.4 Related work

One design that exploits the predictable behavior of reservation-based physical memory allocators to improve TLB hit rates is SpecTLB [7]. SpecTLB provides speculative translations for TLB misses without consulting the page table by tracking partially filled large-page reservations. It claims to overlap an average of 57% of page walks with successful speculative execution. On a TLB miss SpecTLB checks if the virtual page that missed in the TLB was part of a large-page reservation.These mappings are however predictions and further does not guarantee if the page is still valid. Hence an explicit validation is required, which involves marking the page table. This amounts to modifying the page table. SpecTLB overcomes this with a heuristic reservation detection mechanism which requires alignment of bits (VA[20:12] = PA[20:12]). SpecTLB requires a special form of superpaging called reservation based superpaging and this is not

supported in commonly used OSs like Linux. Further not all systems have superpaging enabled hence reducing the usability of this design. SpecTLB also requires validation on hits, which involve page walks.Mis-speculations and alignment restrictions tend to further degrade the performance of this design. Section 2.5 introduces our approach.

2.5 Our approach

Our study uses intermediate levels of page contiguity present in applications to coalesce contiguous entries together. Two entries are contiguous if their VPNs and PPNs are consecutive. Our approach requires extra bits to hold the contiguity length and involves range checks, which can be performed through some combination logic. We first quantify the amount of page contiguity on real systems by varying OS parameters. We find sufficient contiguity (10 - 30 page contiguity) under various OS configurations. With this detail we determine performance numbers for a Coalesced TLB. We find an average of 25% miss eliminations over all the benchmarks using a 8 - 16 fully associative Coalesced TLB.

Chapter 3
Contiguity - Real System Experiments

Our goal is to exploit as much page contiguity as possible. Since a real system runs many processes, the number of superpages available for allocation tend to go down. As a first cut, we quantify the amount of contiguity present in benchmarks (table: 3.2) run under similar conditions. Even though simulations are used to substantiate architectural proposals they differ from a real system for the following reasons.

- These experiments collect page allocations over the whole life time of a benchmark unlike simulations, which capture phases of execution.
- When simulated the benchmark is the only program running. The effect of other processes is not obtained.
- The simulator uses a memory is not very fragmented since the OS has just booted up. The real system experiments are run on a system that has been running for days, thereby removing any such initial contiguity.

The following section shows the configuration used to run these experiments.

3.1 System Configuration

The system configuration mentioned in Table 3.1 was used to run these experiments. Fedora 15 is chosen due to its stable support for kernel 2.6.38, which supports THS. Since THS is one of the features being studied in this thesis, it was critical to run experiments on such a setup.

Operating System	Fedora 15
Kernel Version	Linux 2.6.38 and above
Memory	3GB
Processor	Intel(R) Core(TM) 2 Duo
Number of cores	2

Table 3.1: System Configuration

Benchmark Name	**Benchmark Suite**	**Benchmark Detail**
401.bzip2	CPU2006	Compression Algorithm
416.gamess	CPU2006	Quantum Chemistry
429.mcf	CPU2006	Combinatorial Optimization
433.milc	CPU2006	Quantum Chromodynamics
436.cactusADM	CPU2006	Physics / General relativity
445.gobmk	CPU2006	AI: Go
453.povray	CPU2006	Image Ray tracing
458.sjeng	CPU2006	AI: Chess
459.GemsFDTD	CPU2006	Computational Electromagnetics
471.omnetpp	CPU2006	Discrete Event simulation
473.astar	CPU2006	AI: A* algorithm
483.XalancBMK	CPU2006	XML processing
001.mummer	BioBench	Genome-level alignment
002.tigr	BioBench	Sequence assembly
007.fasta	BioBench	Sequence search

Table 3.2: Benchmarks used in the study

3.1.1 Benchmarks

Benchmarks mentioned in Table 3.2 are chosen from SPEC 2006 [16] and BioBench [4] suites. We ran some initial experiments to determine the D-TLB miss rates. These benchmarks had the highest D-TLB miss rates. Reference inputs used to run these benchmarks are listed in Table 3.3.

3.2 Methodology

To run real system experiments the following parameters are studied:

- THS: A kernel parameter that allocates superpages seamlessly.

Benchmark Name	Benchmark Input
401.bzip2	input.combined
416.gamess	triazolium.config
429.mcf	inp.in
433.milc	su3imp.in
436.cactusADM	benchADM.par
445.gobmk	trevord.tst
453.povray	SPEC-benchmark-ref.ini
458.sjeng	ref.txt
459.gemsFDTD	ref.in
471.omnetpp	omnetpp.ini
473.astar	BigLakes2048.cfg
483.xalancBMK	t5.xml, xalanc.xsl
001.mummer	hs_chrY.fa hs_chr17.fa
002.tigr	sitchensis.fa
007.fasta	-a -m 5 -O

Table 3.3: Benchmark Input

- Memory defragmentation daemon: A daemon that works in conjunction with THS to allocate super pages by compacting free pages.
- System load: The amount of memory usage. To create this a utility called Memhog [17] is used.

3.2.1 Memhog

Memhog is a utility that allocates a given amount of memory and holds it until it is explicitly terminated. This is used to load the real system when benchmarks are executed. We stress the memory at the following three levels.

- 0 % memory Hog.
- 25% memory Hog
- 50% memory Hog.

Memhog (in %)	THS state	Defrag state
0	On	On
25	On	On
50	On	On
0	Off	On
25	Off	On
50	Off	On
0	On	Off
25	On	Off
50	On	Off
0	Off	Off
25	Off	Off
50	Off	Off

Table 3.4: Various scenarios tested with the real system infrastructure

3.2.2 Benchmark Instrumentation

To collect statistics on OS page allocation, we created a utility called *dataCollect*. *dataCollect* reads through the page table of a process by reading the system files */proc/pid/pagemap* and */proc/pid/maps*. The VPN to PPN mapping is used to generate contiguity statistics. This utility is run throughout the lifetime of the benchmark by instrumenting each benchmark to call the *dataCollect* utility inside its main function. The utility is spawned periodically to collect page distribution over the whole run of the benchmark. Each benchmark is run twice and the results are averaged over both the runs.

By varying the amount of memory hog, toggling THS support and defragmentation daemon, the following twelve scenarios as shown in table 3.4 are run.

3.2.3 Results

The weighted average contiguity is calculated by counting each contiguous block of Virtual to Physical mapping exactly once, and weighting that by the length of its contiguity. The results are shown in Figures 3.1 - 3.6.

3.2.4 Average Contiguity Results

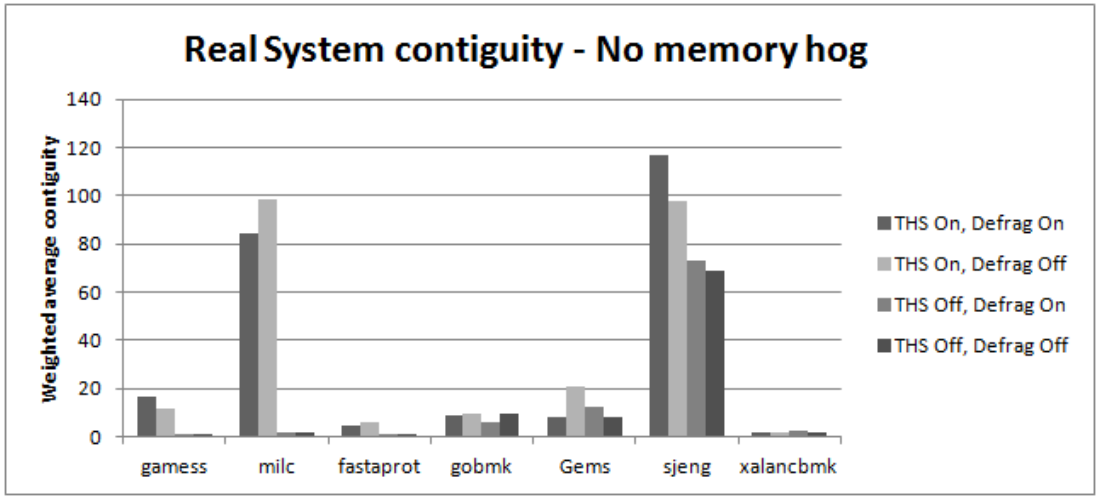

Figure 3.1: Weighted average contiguity with no memory hog, Part 1

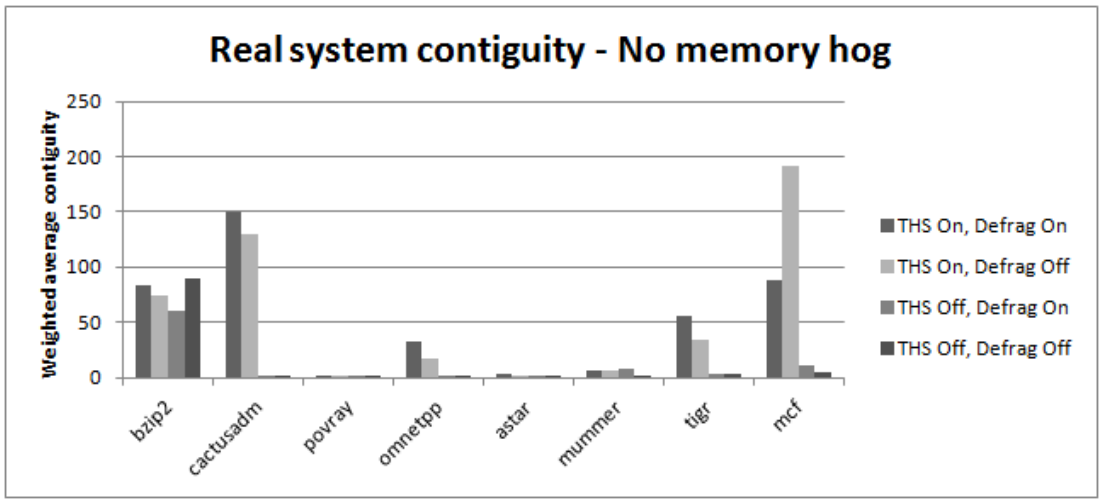

Figure 3.2: Weighted average contiguity with no memory hog, Part 2

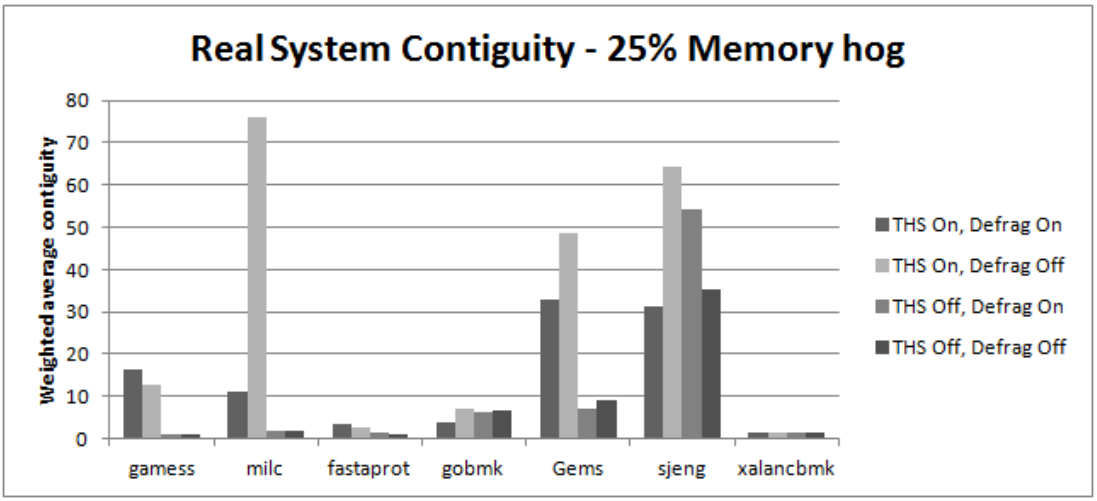

Figure 3.3: Weighted average contiguity with 25% memory hog, Part 1

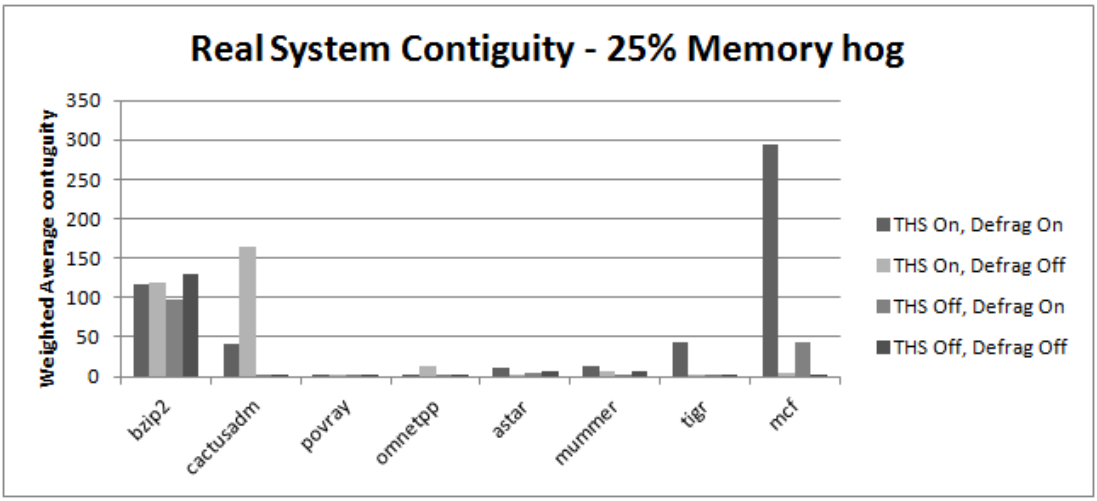

Figure 3.4: Weighted average contiguity with 25% memory hog, Part 2

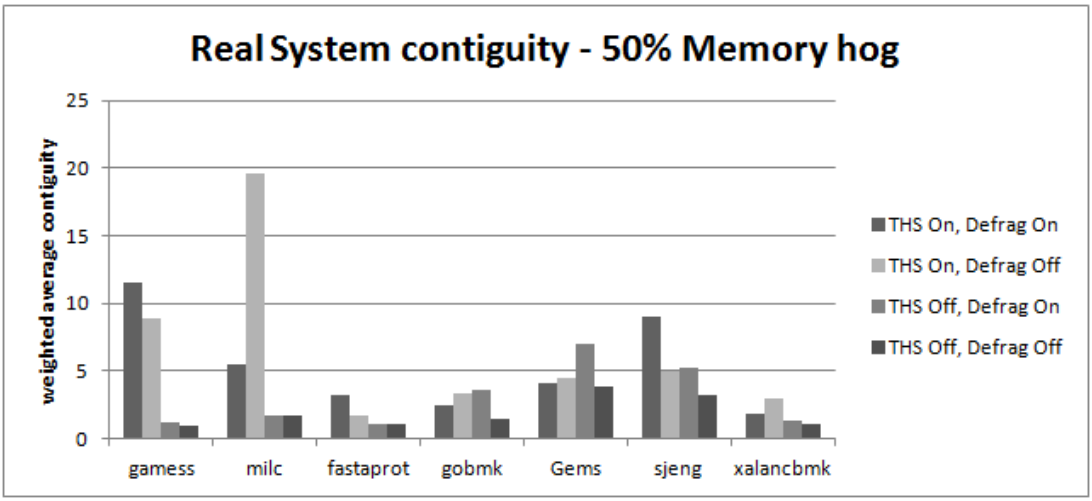

Figure 3.5: Weighted average contiguity with 50% memory hog , Part 1

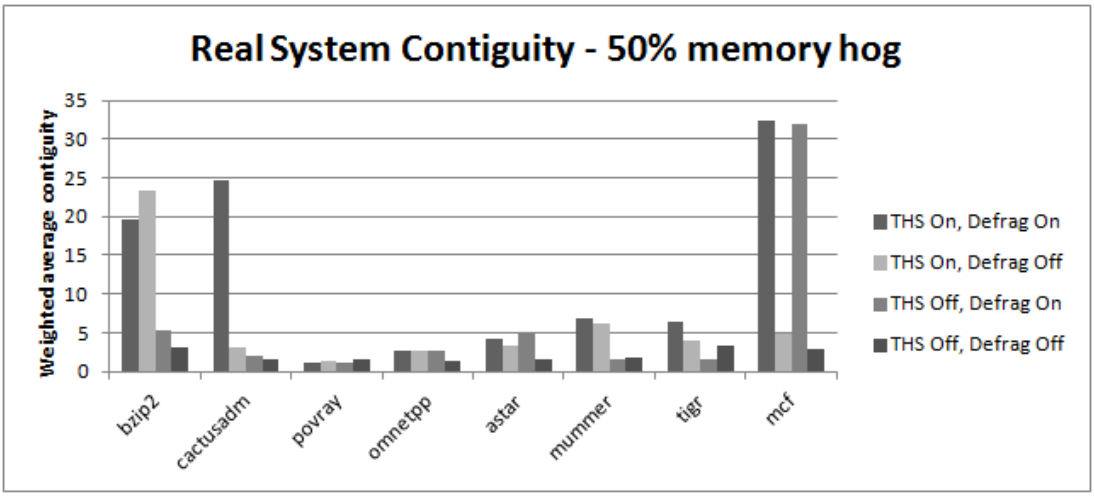

Figure 3.6: Weighted average contiguity with 50% memory hog , Part 2

3.2.5 Observations

The results in Figures 3.1 - 3.6 show contiguity exists across all the configurations. 401.bzip2 is a benchmark that exhibits a lot of contiguity in all scenarios. Since 401.bzip2 is a compression algorithm it could be using large data structures which in turn could be allocated in contiguous chunks of memory. On the other hand the BioBench workload 007.fasta is a sequence searching utility and hence might access random address patterns that are not contiguous. This behavior is evident from the results which show low overall contiguity.

Turning on THS increases the average page contiguity from 10 to 30. This is because THS transparently allocates larger pages if possible and most benchmarks that require data spanning page boundaries use this. Further since modern Linux kernels have this feature enabled by default, a future micro-architectural design involving this feature should have usability. Turning off this feature still gets 10 pages of contiguity. Hence under both these scenarios a micro-architectural design involving contiguity stands to gain.

Lower levels of memory hog, upto 25%, make for great contiguity. In some benchmarks a memory hog of 25% gets better results than no memory hog. Some examples are 416.gamess, 459.GemsFDTD and 471.astar. This is due to the combined effect of the defragmentation daemon and system load. Nevertheless both 0% and 25% memory hog produce an average contiguity of around 28.1 and 29.4 pages respectively . When the memory hog is increased to 50% the average page contiguity dips to 5.4 since the defragmentation daemon finds fewer free blocks to compact. A configuration with THS on, defragmentation daemon on and 25% memory hog has the highest average contiguity of 45.8 while a configuration with THS off, defragmentation daemon off and 50% memory hog has the lowest contiguity of 2.11. As we can see even the worst case has sufficient contiguity that can be exploited with a Coalesced TLB, discussed in Chapter 5 . To test this out infrastructure listed in the next chapter is used.

Chapter 4
Simulation Infrastructure

Given the amount of contiguity exhibited by the benchmarks, a Coalesced TLB would benefit from it. We used traces collected from Simics [39] to evaluate its performance. The following sections introduce the simulators used.

4.1 Simics

Windriver Simics [39] is a full system functional simulator that supports popular micro-architectures like x86, SPARC etc. Simics provides functional correctness of a program running on it by ensuring no wrong paths (mis-predicted paths) are simulated. As a result Simics can be used to verify micro-architectural state. Simics allows users to extend components like TLBs, trace generators among many other modules. Even though Simics is comprehensive, it remains a functional simulator and hence does not have a notion of time.This requires a more cycle accurate simulator to model pipeline stage delays. Many such simulators are present for popular architectures [30, 38] . We used one such simulator called FeS2 [38].

4.2 FeS2

FeS2 is a cycle accurate, x86 multiprocessor simulator designed to work with Simics and was chosen since it works with x86 architectures. FeS2 decodes an x86 instruction into micro instructions and schedules them for execution out of

order [38, 28]. Some other benefits are its support for complex memory subsystems like Ruby [19] and a variety of branch predictors. Ruby could be extended to model interconnection networks [18] and power modules. Support for the 64 bit architecture x86_64 was recently included. Since the study deals with 64 bit systems, FeS2 was chosen as the timing accurate simulator to be used in this study.

4.2.1 Traditional FeS2-Simics Interaction

In a traditional Simics-FeS2 simulator setup, FeS2 reads instructions from Simics and executes them in its pipeline. The two simulators perform a consistency check on every instruction commit by comparing their physical registers. In case of a mismatch, FeS2 copies the state information from Simics and continues to simulate the next instruction. FeS2 also relies on Simics to provide it memory translation and paging information since Simics simulates the OS. This setup can take weeks to simulate benchmarks like 436.cactusADM. To improve simulation run times we added trace driven support to FeS2. To decouple the two simulators, the trace must contain enough information to replicate the architectural state. This is explained in the next section.

4.3 Trace Driven FeS2

Since we are interested in the benchmark phases that stress the micro-architecture the most, a huge number of instructions are skipped during every simulation. This leads to an overall run time of a number of weeks. To speed up the process we extended FeS2 to read from a trace generated by Simics.

4.3.1 Simics trace generation module

Simics provides a trace generation module that generates the following information.

- Instructions executed
- Data accesses and the translation information.
- Exceptions
- OS instructions and their memory accesses.

The Simics development SDK [33] has APIs to extract important architectural details like register values, instruction disassembly and page walk information. We extend the trace generation module as follows.

- An initial snapshot of the system is dumped into the trace file at the start. This includes
  - Architecture type.
  - Physical registers (based on the architecture type)
  - XMM registers, if supported.
  - FPU status and FPU registers.
  - Register mapping information.
- The disassembled version of the instruction is stored.
- Registers modified by that instruction are stored along with their updated values.

We have created a new processor in FeS2 to read these traces called the Trace Processor.

4.3.2 Trace Processor

The trace processor reads inputs from a trace file and sends them to the fetch stage of the FeS2 pipeline. Based on the fetch width of the FeS2 processor, that many instructions are read from the trace file. Since Simics is decoupled from FeS2, memory translations and physical contents are stored in two maps. These maps are updated when reading the data access lines from the trace. As the instructions move through the pipeline, effective addresses are calculated from FeS2's internal register state and the resulting address is looked up in one of the maps to obtain its physical content. Since FeS2 supports exception handling and branch prediction modules the trace needs to be rewound when an exception or a misprediction occurs. This is explained in detail in the next section.

4.3.3 Rewinds

A rewind updates the instruction pointer (IP) to reload a previously executed instruction. To be able to rewind, the processor stores file pointers and register state information for every macro instruction in the pipeline. During a rewind the register state is set back to the state when the instruction being rewound was first fetched. During a rewind the buffers holding state information and the FeS2 pipeline are flushed.

4.3.4 Future

This setup is tested to work with multi-fetch processors on a simple memory hierarchy. In the future this infrastructure can be extended to work with more complex memory systems, having variable delays and port contention built into them. This infrastructure could be used to test the performance of a Coalesced TLB mentioned in chapter 5.

Chapter 5

Coalesced TLB

A Coalesced TLB coalesces contiguous entries in the TLB within one entry. This can be extended from a normal TLB by adding some bits to each entry to denote the length of contiguity. Figures 5.1 and 5.2 show how the same content is stored on the two TLBs. The contiguous entries are coalesced to a single entry (refer to the first two entries in 5.2). The entry with VPN=0x501021 is not coalesced with the entry having VPN=0x501022 since their PPN's are not contiguous.

Virtual Page Number	Physical Page Number
0x501023	0x1254
0x44531F1	0x88522
0x501022	0x1253
0x501021	0x6652
0x501024	0x1255
0x44531F2	0x88523

Figure 5.1: Contents of a normal TLB

5.1 Coalescing Operation

A Coalesced TLB entry contains the following fields.

- Base VPN
- Base PPN

Virtual Page Number	Physical Page Number	Compression Length
0x501022	0x1253	3
0x44531F1	0x88522	2
0x501021	0x6652	1

Figure 5.2: Contents of the corresponding Coalesced TLB

- Coalesced length
- Flags (similar to the normal TLB)

When two entries are contiguous, the lesser of the two is stored as the base VPN-PPN mapping and the coalesced length is increased by one. We have not restricted the number of entries that can be coalesced. Further coalescing occurs between two VPNs irrespective of whether the newly looked up VPN is lesser or greater than the other. To illustrate this assume the following two addresses are added to the Coalesced TLB depicted in Figure 5.2. After adding the first mapping coalescing cannot be performed as the mapping $0x44531F3 \rightarrow 0x88524$ is missing. When the second mapping is looked up, entries with VPN $0x44531F1$ and $0x44531F4$ are coalesced along with the incoming VPN $0x44531F3$ to create one entry of length 4.

S.No	VPN	PPN
1	0x44531F4	0x88525
2	0x44531F3	0x88524

Table 5.1: Mappings added to the Coalesced TLB in Figure 5.2. A study of the coalescing behavior.

5.2 Lookup In Coalesced TLB

The Coalesced TLB has a slightly different lookup function when compared to the normal TLB. Since the VPN corresponds to the least VPN in that range, the lookup function performs a check in the range defined by $BaseVPN \leftrightarrow BaseVPN + CoalescedLength$. If the VPN is within that range, the PPN is similarly checked before tagging it as a hit. On a miss, the Coalesced TLB evicts an entry based on its replacement policy. Some future studies could involve modifying the replacement scheme to only evict entries with unit Coalesced length. This is discussed in Section 6.1.

5.3 Benefits

To understand the benefits of such an architecture, one can look at the effective capacity of a Coalesced TLB. If n were the average coalesced length of an entry in the TLB, a similarly sized Coalesced TLB would hold $n * X$ worth of entries, where X is the total number of entries on the normal TLB. This reduces miss rates without having to introduce complicated hardware. The experiments run in chapter 3 show an average contiguity of 30 pages with THS turned on and 10 pages with THS turned off.

5.4 Performance Evaluation

This section quantifies the performance gain with a Coalesced TLB.

5.4.1 Experiment setup

To carry out this experiment, phases of the benchmark that stress the micro-architecture are selected using a utility called Simpoints [37]. Simics traces from ten such simpoints are collected for each benchmark. The data references of

these simpoints are used to populate two fully associative TLBs with varying sizes and LRU replacement policy. The first TLB is a baseline that exactly resembles a normal TLB. The second TLB contains coalesced entries used to characterize performance benefits for a range of fully associative TLBs. The following subsection mentions how performance gains are obtained.

5.4.2 Results: Recency plot

The performance improvement metric is obtained as follows.

- On a hit in both the TLBs, the depth of that entry from the head of LRU list is noted.
- Frequency counts are obtained for each depth.
- Cumulative distribution functions (CDF) are calculated for these frequency counts.
- With the X axis denoting the depth of the hit, the CDF denotes the total number of hits to expect with a fully associative TLB of that size.
- The normal TLB's CDF is subtracted from the Coalesced TLB's CDF and these values are divided by the total number of misses at each depth in the Normal TLB.
- This indicates the performance gain with a Coalesced TLB.

The following subsections show the recency plots with THS turned on and off.

5.4.3 Recency Plots - THS On

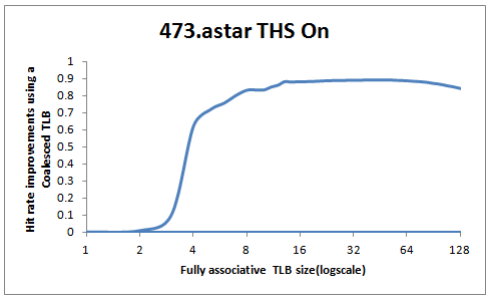

Figure 5.3: Recency plot of 473.astar with THS on

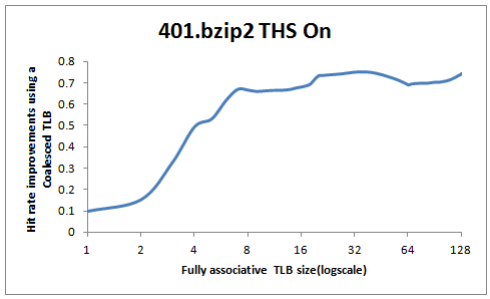

Figure 5.4: Recency plot of 401.bzip2 with THS on

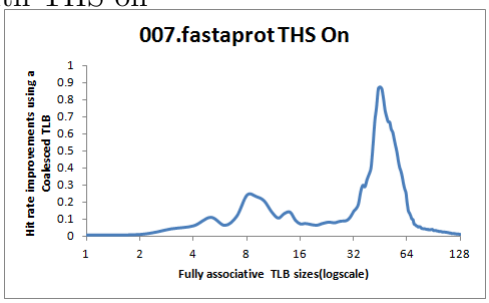

Figure 5.5: Recency plot of 007.fasta_prot with THS on

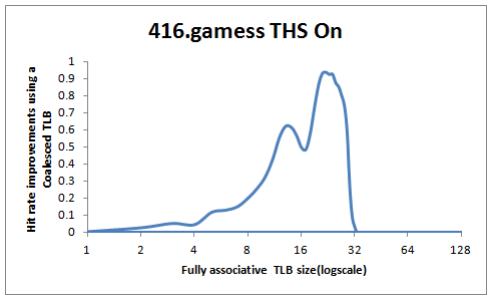

Figure 5.6: Recency plot of 416.gamess with THS on

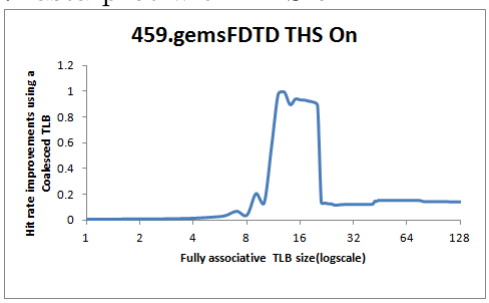

Figure 5.7: Recency plot of 459.GemsFDTD with THS on

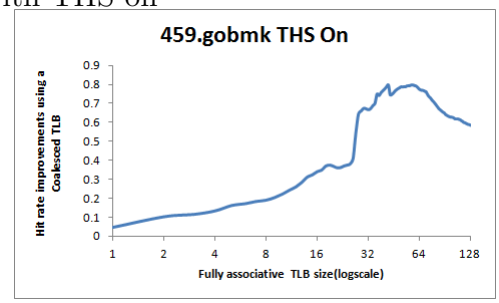

Figure 5.8: Recency plot of 445.gobmk with THS on

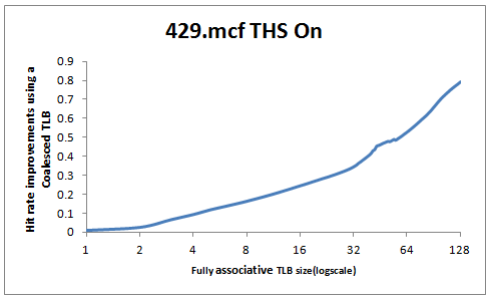

Figure 5.9: Recency plot of 429.mcf with THS on

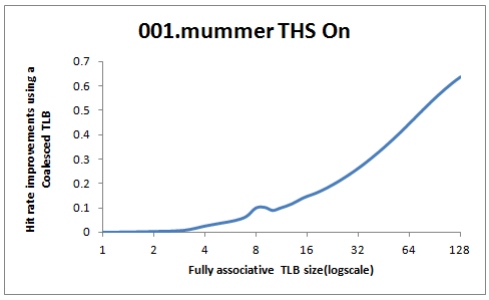

Figure 5.10: Recency plot of 001.mummer with THS on

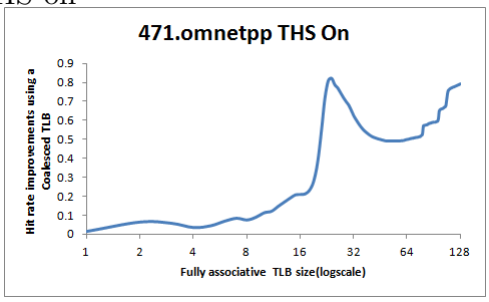

Figure 5.11: Recency plot of 471.omnetpp with THS on

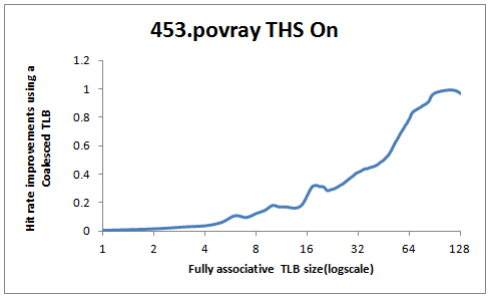

Figure 5.12: Recency plot of 453.povray with THS on

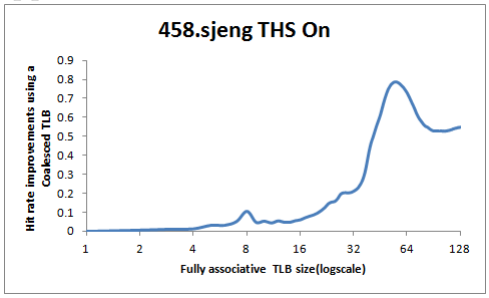

Figure 5.13: Recency plot of 458.sjeng with THS on

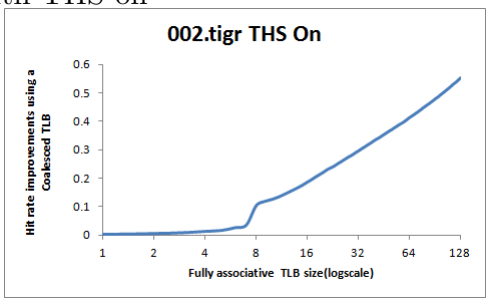

Figure 5.14: Recency plot of 002.tigr with THS on

5.4.4 Recency Plots - THS off

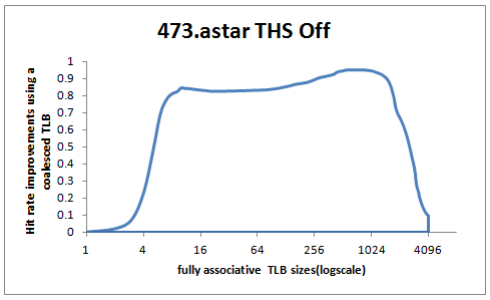

Figure 5.15: Recency plot of 471.astar with THS off

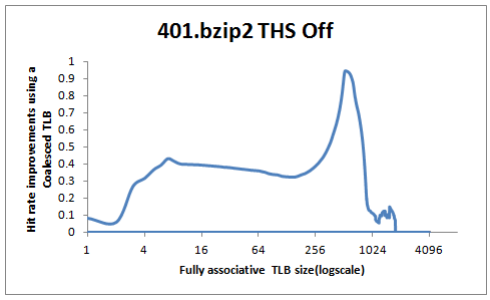

Figure 5.16: Recency plot of 401.bzip2 with THS off

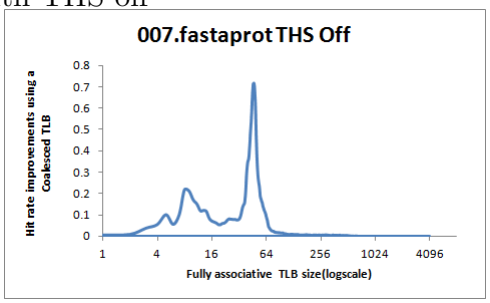

Figure 5.17: Recency plot of 007.fasta_prot with THS off

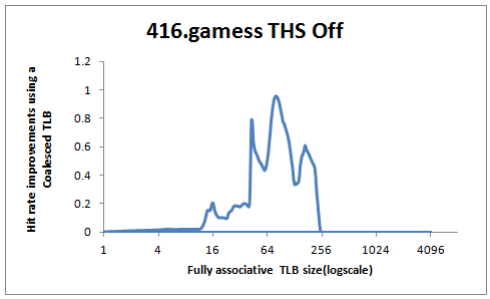

Figure 5.18: Recency plot of 416.gamess with THS off

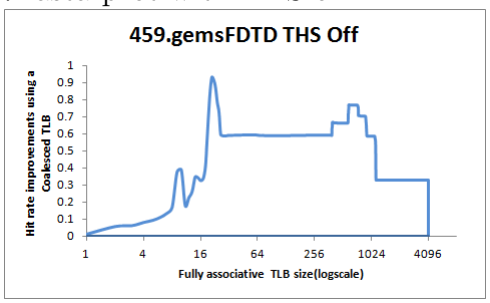

Figure 5.19: Recency plot of 459.GemsFDTD with THS off

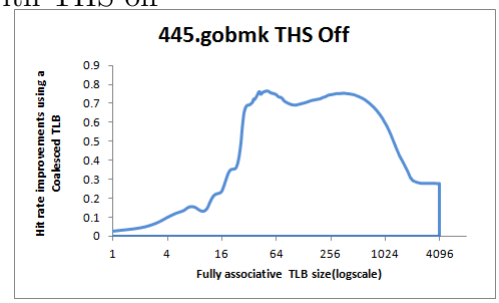

Figure 5.20: Recency plot of 445.gobmk with THS off

5.4.5 Observations

The recency plots denote hit rate improvement with a Coalesced TLB. It can be seen from Figures 5.3 - 5.14 that the plots increase around 8 - 16 entries and go down to zero around 256 - 512 entries. This is due to a low number of pages

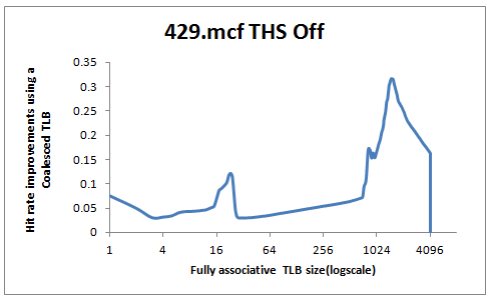

Figure 5.21: Recency plot of 429.mcf with THS off

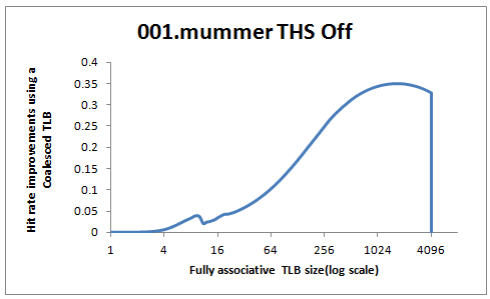

Figure 5.22: Recency plot of 001.mumer with THS off

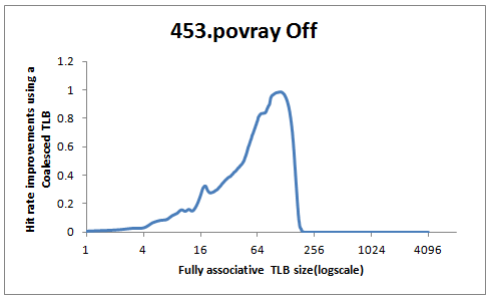

Figure 5.23: Recency plot of 471.povray with THS off

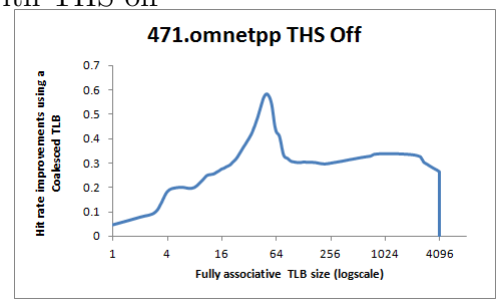

Figure 5.24: Recency plot of 473.omnetpp with THS off

being used with THS turned on (250 - 500 pages). Having less than 8 entries doesn't gain much as there are very few entries to hold both coalesced and non-coalesced entries. Future studies could study the performance gains of a TLB holding coalesced entries alone. With 8 - 16 entries miss elimination ranges from 10% (figure 5.5) to 90% (figure 5.3).

The recency plots with THS turned off, mentioned in Figures 5.15- 5.24 show benefits over a wider range. This is because the number of pages being used across the whole run of the trace is much higher than in the previous case. Povray, Figure 5.23 is an exception to this rule and uses the same number of pages as its THS off counterpart. Even though benefits can be seen over a wide range of fully associative TLB sizes, it is not practical to use a large fully associative TLB. Since 8 - 16 entries show considerable gains they can be used to improve performance.

5.5 Conclusion

This study shows performance improvement of 10% - 90% using a 8 - 16 entry fully associative Coalesced TLB. Though a Coalesced TLB requires range checks on a lookup it can be implemented using simple combinational logic. The Coalescing operation must be taken off the critical path by performing it in the background. The above results show this to be a low cost approach to obtain better TLB hit rates. The next chapter talks about some future studies related to the Coalesced TLB.

Chapter 6
Discussions and Future

We have shown performance improvement with the Coalesced TLB using very little hardware changes. This is however a basic design and needs to be studied further by testing set indexing, replacement policies and sizes. We list some future studies here.

6.1 Replacement Policy

If an LRU replacement policy is used in the Coalesced TLB, a simple modification choosing non contiguous entries for eviction could be tested. This policy must be restricted to a certain depth from the head of the LRU so as to not evict a unit length entry that was recently accessed.

6.2 Prefetching from Cache line

If the TLB is set indexed, the number of entries that can be coalesced is restricted to the number of blocks within that line. In such a scenario we could prefetch the whole cache line into a coalesced entry if the former has contiguous VPNs and PPNs.

6.3 Coalescing entries within TLB

The experiments in section 5.4 show large number of pages being contiguous, sometimes leading up to 1K pages. Hence on a set indexed TLB there is scope

to coalesce entries among sets. This scheme could work on top of the prefetching mechanism discussed above.

6.4 Small fully associative buffers

Another way of improving hit rates could be to add a small fully associative buffer next to the Coalesced TLB. This buffer could be at most 16 entries, which experiments in section 5.4 have shown to improve hit rates by 30% on an average. By varying inclusion patterns (with the traditional L1 and L2 TLBs) we can study various designs.

References

[1] Buddy memory allocation: Wikipedia page.

[2] Hugepage-linux documentation.

[3] Transparent hugepage support. January 2011.

[4] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce Jacob, Chau-Wen Tseng, and Donald Yeung. BioBench: A benchmark suite of bioinformatics applications. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2005), pp. 2-9*, pages 2–9, Austin, Texas, March 2005. IEEE.

[5] Andrea Arcangeli. Transparent hugepage support. *presented at the KVM Forum 2010*, Boston, MA, Aug. 2010.

[6] Thomas W. Barr, Alan L. Scott, and Scott Rixner. Translation caching: Skip, don't walk (the page table). In *Proceedings of the 37th International Simposium on Computer Architecture (ISCA XXXVII, Saint-Malo, France, June 2010)*, pages 1–12, Saint-Malo, France, June 2010. ACM.

[7] Thomas W. Barr, Alan L. Scott, and Scott Rixner. SpecTLB: A mechanism for speculative address translation. In *Proceedings of the 37th International Simposium on Computer Architecture (ISCA XXXVIII, San Jose, USA, June 2011)*, pages 307–317, San Jose, USA, June 2011. ACM.

[8] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional pagewalks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, pages 26–35, Seattle, WA, March 2008. ACM.

[9] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared last-level TLBs for chip multiprocessors. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture*, pages 1–12. ACM, 2011.

[10] Abhishek Bhattacharjee and Margaret Martonosi. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. In *Proceedings of THE 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–12, San Antonio, TX, USA, February 2009. ACM.

[11] Abhishek Bhattacharjee and Margaret Martonosi. Inter-core cooperative TLB prefetchers for chip multiprocessors. In *Proceedings of 15th Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS XV)*, pages 1–12, Pittsburgh, Pennsylvania, USA, March 2010. ACM.

[12] J.Bradley Chen, Anita Borg, and Norman P.Jouppi. A simulation based study of TLB performance. In *WRL Research report 91/2*, pages 1–32, Palo Alto, CA, May 1992.

[13] Douglas W. Clark and Joel S.Emer. Performance of the VAX-11/780 translation buffers: Simulation and measurement. In *ACM Transactions on Computer Systems*, volume 8, pages 31–62, Hudson,MA, February 1985. ACM.

[14] Jonathon Corbet. Memory compaction.

[15] Jonathon Corbet. Page migration: Linux article.

[16] Standard Performance Evaluation Corporation. SPEC CPU2006. Accessed May 2012.

[17] Alex Feinberg. Memhog: Github.

[18] GEM5. Garnett - interconnection network. Accessed November 2011.

[19] GEM5. Gem5 - ruby. Accessed October 2011.

[20] Intel. Intel 64 and ia-32 architectures software developers manual. volume 1. Intel, May 2012.

[21] Bruce L Jacob and Trevor N Mudge. Software-managed address translation. In *Proceedings of the Third International Symposium on High Performance Computer Architecture, (HPCA III)*, San Antonio, TX, February 1997. ACM.

[22] Bruce L Jacob and Trevor N Mudge. A look at several memory management units: TLB-refill, and page table organizations. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 1–12, San Jose, CA, October 1998. ACM.

[23] Bruce L Jacob and Trevor N Mudge. Virtual memory in contemporary microprocesors. 1998.

[24] Gokul B Kandiraju and Anand Sivasubramaniam. Characterizing the d-TLB behavior of SPEC CPU2000 benchmarks. *Private communication.*

[25] Gokul B Kandiraju and Anand Sivasubramaniam. Going the distance for TLB prefetching: An application-driven study. *Private communication.*

[26] David Kanter. Inside nehalem: Intel's future processor and system. Accessed May 2011.

[27] Kenneth C. Knowlton. A fast storage allocator. In *Communications of the ACM*, volume 8, pages 623–624, New York,NY, October 1965. ACM.

[28] Milo M. K. Martin1, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, , and David A. Wood. Multifacets general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News(CAN 2005)*, September, 2005.

[29] Collin McCurdy, Alan L.Cox, and Jeffrey Vetter. Investigating the TLB behavior of high-end scientific applications on commodity microprocessors. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 135–144. ACM, 2008.

[30] University of Wisconsin-Madison. Multifacet gems. Accessed October 2011.

[31] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta. The impact of architectural trends on operating system performance. In *ACM Transactions on Computer Systems*, pages 1–14, Stanford,CA, 1995. ACM.

[32] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenstrom. Recency-based TLB preloading. In *Proceedings of the Twenty Seventh International Conference on Computer Architecture (ISCA XVII)*, pages 117–127. ACM, 2000.

[33] Virtutech Simics. Simics programming guide. 2007.

[34] Shekhar Srikantaiah, Mahmut Kandemir, and Mary Jane Irwin. Adaptive set-pinning: Managing shared caches in chip multiprocessors. In *Proceedings of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008)*, pages 135–144, Seattle,WA, USA, February 2008. ACM.

[35] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 1–14. ACM, October 1994.

[36] Richard Uhlig, David Nagle, Tim Stanley, Trevor Mudge, Stuart Sechrest, and Richard Brown. Design tradeoffs for software managed TLBs. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 175–205, San Diego,CA, May 1993. ACM.

[37] San Diego University of Califronia. Simpoint. Accessed May 2012.

[38] Urbana-Champaign University of Illinois. Fes2 - a full-system event driven simulator for x86. Accessed March 2012.

[39] Windriver. Windriver Simics. Accessed May 2012.