ABSTRACT

REZA, SAJJID. Reducing Migration-induced Misses in an Over-Subscribed Multiprocessor System. (Under the direction of Prof. Gregory T. Byrd.)

In a large multiprocessor server platform, using multicore chips, the scheduler often migrates a scheduling entity, i.e. a thread or process or virtual machine, in order to achieve better load balancing or ensure fairness among competing scheduling entities. Similar to a context switch, each such migration incurs overhead, from saving and restoring processor states and virtual machine control structure, extra Translation Look aside Buffer (TLB) misses and related page walks, cache misses, and interrupt rerouting. Such migration impact is likely to be more severe in virtualized environments, where high over-subscription of CPUs is very common for server consolidation workloads or virtual desktop infrastructure deployment causing frequent migrations and context switches. One way to mitigate such overhead would be to constrain a thread or virtual CPU to be executed on a specific CPU only. However, this approach will likely cause load imbalance, and would still suffer from context switch related misses. Furthermore, effectively determining and managing such constrain assignments of threads to CPUs would not be a trivial task and is unrealistic particularly in a large data center with thousands of servers running varieties of workloads. Alternatively, if we could effectively preserve important footprints (both cache and TLB) and reload them with migration of the process, we could avoid the expensive misses.

We characterized the effectiveness of saving and restoring TLB cache entries over a context switch/migration and we found that 60-100% of TLB cache misses could be avoided. We also present two predictors to select TLB entries that are most likely to be reused. After

comprehensive evolution, we found that both predictors could reduce storage footprints by 20-80% for most workloads while keeping the false negative rate very low.

We also characterized the effectiveness of preserving cache - in particular L2 where working set of a process/VM resides, in reducing the migration-induced cache misses. We propose three different MRU based schemes (local, regional and global) to select cache tags to be prefetched following a migration. After comprehensive evolution we found that the simple MRU (most recently used) based selection can provide similar benefit as more complex and costly (in terms of hardware implementation) global or regional MRU schemes and can provide performance benefits of 1.5% -27% reduction in CPI (cycles per instruction) following a migration. We also compared our migration prefetcher with standard hardware stream-based and stride-based prefetchers and found that our migration prefetcher performs almost always better. In some cases, combining the migration prefetcher with a standard hardware prefetcher can provide significant further improvements.

© Copyright 2012 by Sajjid Reza

All Rights Reserved

Reducing Migration-induced Misses in an Over-Subscribed Multiprocessor System

by
Sajjid Reza

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2012

APPROVED BY:

Prof. Gregory T. Byrd
Committee Chair

Prof. Eric Rotenberg

Prof. Tom M. Conte

Prof. Frank Mueller

DEDICATION

To my late parents, children and wife.

BIOGRAPHY

Sajjid Reza has been working at Intel Corporation since 1998. He is a member of the VMware Technology Lab team at Intel. His primary focus area is architectural performance analysis. He obtained his bachelor's degree in Computer Science from the University of Mississippi, Oxford, Mississippi in 1994 and his master's degree in electrical and computer science from Vanderbilt University, Nashville, Tennessee in 1997. He is married with three children and resides in Cary, NC.

ACKNOWLEDGMENTS

First and foremost, I thank Almighty Allah for granting me the favor of strength, wisdom, confidence and perseverance to carry out my studies.

I would like to express my deepest appreciation to my committee chair, Professor Greg Byrd. Without his continuous inspiration, guidance and persistent help this dissertation would not have seen the light of the day.

I would like to thank my committee members, Professor Thomas M. Conte, Professor Eric Rotenberg, and Professor Frank Mueller for their feedbacks and encouragement throughout the research for this dissertation.

I would like to specially remember my late parents for instilling in me the thirst of knowledge early on in my childhood. I would also like to express lots of thanks to my family for their long sacrifice and love.

Finally, I would also like to thank Intel Corporation for its financial support throughout the study.

TABLE OF CONTENTS

LIST OF TABLES	vii
LIST OF FIGURES	viii
CHAPTER 1: INTRODUCTION	1
1.1 Motivation	2
1.2 Contributions	7
1.3 Organization of the dissertation	8
CHAPTER 2: RELATED WORK.....	9
CHAPTER 3: CLASSIFICATION OF MIGRATION TYPES.....	14
CHAPTER 4: PRESERVING TRANSLATION LOOKASIDE BUFFER CACHE FOOTPRINTS	18
4.1 Save and Restore Method.....	19
4.2 Aging Based Predictor	22
4.3 Aged Plus Total Access.....	25
4.4 Simulation Results.....	26
4.4.1. <i>Aging based Predictor</i>	28
4.4.2 <i>Aging plus Total access Predictor</i>	38
4.5 Implementation Consideration	40
4.6 Summary	41
CHAPTER 5: REDUCING MIGRATION-INDUCED CACHE MISSES	44
5.1 Simulation Methodology.....	48
5.2 Efficiency Improvement.....	50
5.2.1 <i>Local MRU Selection</i>	51
5.2.2 <i>Global MRU Selection</i>	51
5.2.3 <i>Regional MRU Selection</i>	51
5. 3 Simulation Results.....	56
5.3.1 <i>Impact of L2 Cache Line State</i>	57

5.3.2 <i>CPI Improvement</i>	59
5.3.3 <i>Sensitivity to PMW Size</i>	62
5.4 Implementation Issues.....	64
5.5 Implementation Study	65
5.5.1 <i>Live-Migration</i>	68
5.5.2 <i>Delayed-Migration</i>	74
5.6 Comparison with Hardware Prefetcher	83
CHAPTER 6: SUMMARY AND FUTURE WORK	87
6.1 Summary	87
6.2 Future Work	88
6.2.1 <i>Reduce message traffic</i>	88
6.2.2 <i>Enhancing the OS/VMM scheduler</i>	89
6.2.3 <i>Extend migration prefetching to VM motion between servers</i>	89
6.2.4 <i>Extend migration prefetching to fault-tolerant computing</i>	89

LIST OF TABLES

Table 5.1: Typical L2 Cache Size of Intel Xeon Servers [26].....	45
Table 5.2: Decomposition of selection set and impact of cache-line state on usefulness	58

LIST OF FIGURES

Figure 1.1: Correlation of L2 MPI and CPU over-subscription ratio with Login VSI VDI workload	3
Figure 1.2: Correlation of LLC MPI, TLB MPI and Migration rate with CPU over-subscription ratio with Login VSI VDI workload	4
Figure 1.3: Impact of VCPU to Core subscription ratio on TLB, L2, and LLC miss latency ..	4
Figure 1.4: Migration-induced L2 misses for SpecCPU2006 C/C++ benchmarks with migrations at every 2 million instructions (after warm-up).	5
Figure 3.1: Local Migration.....	15
Figure 3.2: Remote Migration.....	15
Figure 3.3: Live and Delayed Local Migration	16
Figure 3.4: Live and Delayed Remote Migration	17
Figure 4.1: Snapshots of 1 st level TLB cache entries over 11 intervals of 1m memory instruction each using gcc trace.	22
Figure 4.2: Age based Predictor for save and restore	23
Figure 4.3: Age Plus Total Access based Predictor	25
Figure 4.4: Simple Save and Restore TLB method	26
Figure 4.5: TLB hit/miss info for Save and Restore TLB method	27
Figure 4.6: Entries predicted using different number of intervals/threshold pairs	28
Figure 4.7: Actually reused predicted 1 st level TLB cache entries using different number of intervals/threshold pairs	29
Figure 4.8: 1 st level TLB cache not selected for saving/restoring using different number of intervals/threshold pairs.	29
Figure 4.9: Actually reused not predicted 1 st level TLB cache entries using different number of intervals/threshold pairs.....	30
Figure 4.10: Actually reused predicted TLB entries selected by the Aging based predictor .	31
Figure 4.11: TLB hits for actually reused predicted entries	32
Figure 4.12: False negative predictions	33
Figure 4.13: TLB hits from false negative predicted entries' reuse	34
Figure 4.14: Prediction based on rejecting most aged entries.	35
Figure 4.15: Aging predictor entries for context switch window of 1m memory operations.	36
Figure 4.16: TLB hits for false negative predictions	37
Figure 4.17: Aging predictor entries for context switch window of 3m memory operations.	37
Figure 4.18: Aging predictor entries for context switch window of 6m memory operations.	38
Figure 4.19: Comparison of Aging only, aging ⁺ total access and simple save and restore	39
Figure 4.20: TLB hits comparison for Aging vs. Aging plus Predictor	39
Figure 5.1: Migration-induced L2 misses for SpecCPU2006 C/C++ benchmarks with migrations at every 2 million instructions (after warm-up).	45

Figure 5.2: Efficiency: Ratio of removed migration-induced L2 misses and the number of L2 entries saved and restored. For this figure, 100% of the L2 is saved and restored, and the post-migration window (PMW) of execution is 2 million instructions.	46
Figure 5.3: bzip2: 2MB LLC	52
Figure 5.4: bzip2: 4MB LLC	52
Figure 5.5: h264: 2MB LLC	52
Figure 5.6: h264: 4MB LLC	53
Figure 5.7: hmmer: 2MB LLC.....	53
Figure 5.8: hmmer: 4MB LLC.....	53
Figure 5.9: mcf: 2MB LLC	54
Figure 5.10: mcf: 4MB LLC	54
Figure 5.11: omnetpp: 2MB LLC	54
Figure 5.12: omnetpp: 4MB LLC	55
Figure 5.13: sjeng: 2MB LLC.....	55
Figure 5.14: sjeng: 4MB LLC.....	55
Figure 5.15: CPI reduction using Global MRU selection: 2MB L2.	60
Figure 5.16: CPI reduction using Local MRU selection: 2MB L2.....	61
Figure 5.17: CPI reduction using Global MRU selection: 4MB L2.	61
Figure 5.18: CPI reduction using Local MRU selection: 4MB L2.	62
Figure 5.19: CPI reduction for various PMW sizes (Local-MRU, 4MB L2).	63
Figure 5.20: Modeled system: collection of multicores with on-chip banked L2 and banked off-chip memory. Networks can be direct (NUMA) or indirect (UMA). This particular study uses two chips, with one CPU and L2 per chip, and two memory banks.	65
Figure 5.21: Triggering live and delayed migrations.....	67
Figure 5.22: Addition of Incoming and outgoing queue to L2	69
Figure 5.23: Cache-coherence Protocol modification to accommodate migration prefetch ..	70
Figure 5.24: Reduction of LLC misses in two-core migration experiment.	71
Figure 5.25: Reduction of CPI in two-core migration experiment.	72
Figure 5.26: Increase in message traffic (over all network links).....	73
Figure 5.27: Reserved directory memory for local L2 cache line storage.....	75
Figure 5.28: Reduction in Cache-Misses in PMW (will fix the x-axis legends)	77
Figure 5.29: Reduction in CPI in PMW.....	77
Figure 5.30: Increase in message traffic (over all network links).....	78
Figure 5.31:CPI reduction from Local MRU.....	80
Figure 5.32: CPI reduction from entire L2 cache set selection.....	81
Figure 5.33: Impact of L2 miss latency on migration prefetcher performance	82
Figure 5.34(a-f): Performance (CPI) comparison of migration only prefetch with/without hardware prefetchers	85

CHAPTER 1: INTRODUCTION

In today's computing platform, there are usually multiple execution contexts (thread/program/process/daemon/Virtual Machine) competing to use a processor at any time. The operating system (OS) or virtual machine monitor (VMM) scheduler commonly applies timesharing policy to context switch among multiple execution contexts. Context switching, however, incurs some direct overheads due to saving and restoring state of the execution context as well as indirect overheads from perturbations of cache footprints, Translation Lookaside Buffer (TLB) entries, branch prediction histories and prefetch history tables. With ever increasing number of cores per physical processor, the OS or VMM scheduler in multicore platforms often chooses to migrate a thread or process or virtual machine (VM) across cores/processors/NUMA nodes, either to achieve improved load balancing or increase fairness among the scheduling entities.

While migration helps in improving load balance across system resources, it also magnifies indirect overheads, since some footprints such as TLB entries, branch history, and prefetch history are permanently lost. A TLB entry miss could lead to an expensive page table walk. If the relevant contents of the page table are not present in the cache hierarchy, then the penalties are the worst, as they require accesses to memory. The cost of missing cache footprints at the destination L2 cache in particular (L1 could be quickly warmed up) would depend on the proximity between the source and destination CPUs and the time delay to the next schedule. If the source and destination CPUs reside in the same node with a shared last level cache, then the cost of refilling destination L2 cache with the important

memory footprints of the process might be smaller than had the source and destination CPUs resided in different nodes with no shared cache. Also, the longer the delay to the next scheduling of the process, the smaller the probability will be to find any cache footprints at the source without making expensive trips to memory.

While some research has been done in estimating the impact/overhead of context switch from cache misses and branch miss-predictions, there are only few [13,14] articles on the characterization of the impact of misses due to a context switch coupled with migration.

In section 1.1 we motivate the problem by looking overall system behavior and at individual single-threaded benchmarks. Section 1.2 discusses the contributions of this thesis. Section 1.3 describes the organization of the thesis.

1.1 Motivation

Especially in a virtualization consolidation or Virtual Desktop Infrastructure (VDI) environment, there are usually many more virtual CPUs (VCPUs) than physical CPUs (cores) in the platform, leading to frequent migration of VMs or VCPUs among different cores/sockets. Figure 1.1 and 1.2 illustrate this trend using the Login VSI 3.0 VDI workload [1] running under VMware ESX server hypervisor on a two-socket system with 16 total physical cores. Each VDI user was configured as single VCPU VM and running Login VSI power user¹ profile. With increasing VCPU-to-core subscription ratio, both the migration rate and corresponding second-level (L2) cache misses per million instructions (MPI) and on-chip

---

¹ Power user profile – 8 simultaneous apps(WORD, OUTLOOK, PDF, FLASH, PPT, JAVA, ZIP,IE) with 40 sec idle time over 12 minute window

shared last-level cache (LLC) MPI increase dramatically, linking indirectly higher migration rate to higher cache misses. We also observe that the L2 cache misses are significantly higher (over 3-4x) than the LLC cache misses with higher migration rate. Moreover, L2 cache contains important footprints (both data and code) that are likely to be accessed and hit in immediate future references to memory by the owner thread or VM, as the principle of locality dictates. The CPU utilization is only at 73% with a VCPU-to-core over-subscription ratio of 4, and there is a lot more room left in term of CPU resources to go to an even higher over-subscription ratio, thus increasing the likelihood of incurring even higher migration rate and cache misses.

Figure 1.1: Correlation of L2 MPI and CPU over-subscription ratio with Login VSI VDI workload

Each migration incurs overhead, similar to a context switch, from saving and restoring processor states and virtual machine control structure (VMCS), extra TLB misses and related page walks, cache misses, and interrupt rerouting (for VMs). The indirect

overhead of TLB and cache misses for a migration could be potentially higher than for a context switch, because the migrated thread could begin execution in a totally different processor environment and cache hierarchy. When migration crosses chip/socket boundaries, even a large on-chip last-level cache (LLC) cannot avoid misses.

Figure 1.2: Correlation of LLC MPI, TLB MPI and Migration rate with CPU over-subscription ratio with Login VSI VDI workload

Figure 1.3: Impact of VCPU to Core subscription ratio on TLB, L2, and LLC miss latency

Memory and network bandwidth are also adversely affected by frequent migrations and the subsequent cache misses. Bandwidth is a global resource, and one CPU's misses can

slow down another's memory accesses. Figure 1.3 illustrates the impact of increasing memory bandwidth consumption from higher consolidations of VMs on the memory access latency. So, the cost of migration will increase at higher rate of consolidation or over-subscription in a system.

We have also investigated cache misses from a single migration using some benchmarks from SpecCP2006C/C++ [2]. We chose those benchmarks that perform computations typically observed in a VDI user session such as compression, multimedia streaming, display encoding, random memory accesses, etc. Figure 1.4 shows the number of migration-induced misses for 2MB and 4MB L2 for some of the SpecCPU2006 C/C++ benchmarks. (This models a worst-case scenario, in which none of the L2 data is available to the thread after the migration. Only misses that would not have missed with the original L2 are counted.)

Figure 1.4: Migration-induced L2 misses for SpecCPU2006 C/C++ benchmarks with migrations at every 2 million instructions (after warm-up).

All of the benchmarks suffer a large number of migration-induced L2 misses. In particular, mcf and bzip suffer the worst, because of their large working set sizes. The

remaining benchmarks had fewer than 6000 migration-induced misses. A few benchmarks (h264, hmmer, sjeng) did not experience larger migration-induced L2 misses with increasing L2 size, likely due to their smaller working set sizes.

In an equally subscribed or under-subscribed system, a user may apply strict processor/core affinity to avoid migrations altogether and thus not suffer any migration-induced cache or TLB misses. However, in highly over-subscribed systems, especially large-scale deployment environments such as data centers, such strict processor/core affinity enforcement over an entire set of processes or VCPUs is either impractical or not easily achieved, and thus frequent migrations are bound to occur, resulting in large migration-induced cache misses and TLB misses.

One easy solution to avoid the migration-induced TLB and Cache (L2 in particular) misses would be to save away the entire TLB and L2 cache in memory before migration at the source CPU and then restore it post-migration at the destination CPU. However, this is not very efficient due to the following observations:

- i. The size of L2 cache could be very large. The frequent migration would require large memory storage.
- ii. Not all entries (for both TLB and L2) may belong to the migrating process.
- iii. Not all entries are likely to be reused following a migration within a reasonable window of instructions.

Thus, a smaller subset of TLB and L2 caches with predictable reuse following a process migration is desirable. In this research, we focus on reducing *migration-induced*

misses for both L2 and TLB — accesses that would not have missed in these caches if the migration had not occurred. If we can reduce the dominating indirect overhead under different types of migration scenarios, such as local or remote migration (across nodes) or live or delayed migration, the scheduler has the flexibility to freely move threads where they are most beneficial to overall system throughput.

1.2 Contributions

Preserving TLB cache entries over context switch/migration

- We characterized the effectiveness of saving and restoring TLB cache entries over a context switch/migration in reducing TLB cache misses. We show that 60-100% of TLB cache misses could be avoided.
- We propose two predictors to select a subset of TLB entries as opposed to the entire TLB cache to be preserved over a context switch/migration with two goals in mind:
  - (1) -one to reduce the storage footprints as multiple threads/VMs will be doing it and
  - (2) to increase the effectiveness by predicting those TLB cache entries that are most likely to be hit following a context switch/migration. Both predictors show that we could reduce storage footprints by 20-80% for most workloads while keeping the false negative rate very low.

Reducing migration-induced cache misses

- We characterize the effectiveness of preserving cache - in particular L2 where working set of a process/VM resides, in reducing the migration-induced cache misses. We propose three different MRU based schemes (local, regional and global) and

compare them in terms of potential benefits from preservation over a migration over range of different sizes of cache lines set. The CPI (cycles per instruction) reduction varies by the workload and was as high as 27%. We also show that the simple MRU (most recently used) based selection can provide similar benefit as more complex and costly (in terms of hardware implementation) global or regional MRU based selection of cache lines for preservation.

- We implemented the proposed MRU based -local and global schemes to preserve cache lines under all possible types of migrations. We actually preserve only cache-line tags rather than the data and thus avoid cache-coherency issues. The preserved tags are used to prefetch the data at appropriate time. Our migration prefetcher performs almost always better than typical hardware stream-based or stride-based prefetchers. In some cases, combining the migration prefetcher with a standard hardware prefetcher can provide significant further improvements.

1.3 Organization of the dissertation

The dissertation is organized as follows. Chapter 2 explains the related work. Chapter 3 discusses classification of migration types. Chapter 4 discusses the preservation of TLB cache. Chapter 5 explains the methodologies with implementation details and related results to reduce migration-induced cache misses. Lastly, Chapter 6 offers a summary and some discussion of future work.

CHAPTER 2: RELATED WORK

A lot of research has been done in understanding the overhead of context switching in a computer system [3, 4, 5, 6, 7]. Context switching introduces two types of overheads in computation. Direct overhead is the cost involved in saving and restoring processor registers, flushing processor pipelines and TLBs, and executing the operating system scheduler to de-schedule the current task, select the next task to run, and update relevant task structures so the actual computation begins again. Indirect overhead or performance impact of context switching comes from increased cache misses (both data and instruction) and TLB misses (both data and instruction), as well as inaccuracies introduced into branch predictions and prefetches.

The migration of a process/thread from one CPU to a different CPU further magnifies the indirect overhead incurred in a context switch. The cache and TLB footprints that were left at the source CPU are not available at the destination CPU, so the computation will incur extra misses if any references to those footprints are made. (We call these “migration-induced” misses.) The cache footprints may be partially available at the destination CPU, if there is any shared cache involved and migration happens only within the cores sharing the cache.

The migration rate is likely to increase in a virtualization server environment, where logical CPUs are often highly oversubscribed by VCPUs for applications such as server consolidation and virtual desktops infrastructure deployment. For example, the vConsolidate benchmark [8] developed by Intel to measure the performance of a system running consolidated workloads has six consolidated stack units (CSU) consisting of a total of 48

VCPUs running in a two-socket server with a total 16 logical cores [9]. VMmark benchmark developed by VMware likewise has also high over-subscription of VCPUs. According to VMware Inc.'s website [10], server consolidation ratios commonly exceed 10 virtual machines per physical processor. Even though cost is the focus in server consolidation, processor designers are continuously adding features such as Nested Paging (NP) in case of AMD [11] and Extended Page Table (EPT) for Intel [12] to improve the virtualization performance.

Li et al. [3] measured the indirect cost of context switch using a synthetic workload. They observed that the indirect cost of context switch varies with different workloads with different memory access behavior and for different architectures. David et al. [4] measured both direct and indirect overhead of context switches by modifying the Linux kernel. They noticed that secondary overhead was higher than direct overhead over a set of workloads that are compute and memory-intensive, involving no I/O.

Some research publications have focused on reducing the direct overhead of context switch. Jaaskelainen et al. [13] proposed co-operative multithreading so context switches occur in places that require a minimal amount of processor state to be saved. Ditzel [14] described how SPARC^{TM} Version 9 has improved context switching direct overhead by saving and restoring fewer registers during a context switch. It efficiently tracks both integer and floating pointer registers if they are modified recently; if some registers were not written into recently then there is no need to save them.

Several studies [3, 5] looked into cache perturbation effects from context switch. Liu et al. [6] studied further how cache misses due to cache perturbations from context switches are influenced by cache parameters and application behaviors.

A common scheme deployed in reducing the overhead incurred in a context switch is to save the related context of a thread or process and later restore the same upon resumption of execution of the same thread or process. Lee et al. [15] partitioned a large branch predictor table into several small ones. A new partition is selected on a context switch, and the partition is filled with branch history previously saved in memory. Parischa and Veidenbaum [16] also presented several schemes to save and restore the branch predictor state on context switches to improve prediction accuracy for Embedded Processors such as Intel Xscale; they showed that a 1-6% improvement in prediction rate can be achieved by saving less than 128 bits for a 1K entry hybrid predictor. Dhodapkar and Smith [17] proposed saving the most significant bits of counters in branch predictor tables on a context switch to improve accuracy. Their scheme used a co-designed Virtual Machine Monitor (VMM) for saving and restoring branch predictor contents in a transparent manner to all conventional software.

Bradford et al. [7] proposed prefetching data for a thread to be executed soon on a context switch by means of saving and restoring prefetch control data through hardware/software collaboration.

Hanyu et al. [18] proposed restoring a program's locality by prefetching into the L2 cache the data a program was using before it was swapped out. A Global History List is used to

record a process' L2 read accesses in LRU order. These accesses are saved along with the process' context when the process is swapped out and loaded to guide prefetching.

Abhik et al. [19] proposed pushing select cache blocks with a live task migration in embedded execution environments. A limited set of region registers is filled by the software identifying the cache blocks to be transferred.

Constantinou et al. [20] considered the influence on the performance of a single thread of the following migration and multi-core parameters: frequency of migration, core warm-up modes, subset of resources that are warmed-up, number of cores, and cache hierarchy organization and showed that the performance losses due to activity migration for a multi-core can be minimized by remembering the predictor state between migrations. However, the study lacks any actual implementation analysis.

Brown et al. [21] is the most closely related to this work. They propose specialized hardware in each core to capture and summarize a thread's working set, and the summary is used to prefetch data into the destination CPU's local cache hierarchy. However, they consider the case of very small threads (100's of instructions) migrating within a single multi-core/many-core processor. Their aim is to eliminate L1 misses during the first instructions executed after migration; the shared LLC will hold most data that will be accessed later in the execution.

In our case, we are migrating heavyweight processes that will run for millions of instructions, and the migration will often cross chip boundaries, where a shared LLC will not help. To be effective, the number of misses eliminated must be higher, because the benefits

are amortized over a longer execution time, so it is appropriate to migrate a larger portion of the working set residing in L2 instead; it does not require any core modifications. We also considered the impact of different types of migrations: they can be live or delayed, as well as within a local node or to a remote node.

CHAPTER 3: CLASSIFICATION OF MIGRATION TYPES

A migration of a process or Virtual CPU (VCPUs) of a virtual machine from one physical core to another core by the operation system (OS) or virtual machine monitor (VMM) in a multicore/multi-processor server platform can be classified based on the following two factors: (1)*Destination* – Is the newly selected core of execution for the process or VCPU within the local node or a remote node of the server platform. (2) *Timing* - Does the migration occur while the process is still in active execution in one of the physical cores or while the process is currently not in execution?

We define two types of migrations based on the *destination* factor.

Local migration – When a process or a VCPU migrates to a physical core within the current node or socket of the system as shown Figure 3.1. For such migration, the last level share cache (LLC) could help to reduce the impact of migration-induced cache misses. This reduction is possible only if there are not enough concurrent competing processes or VCPUs for the shared cache such that the working set residing in the L2 of the source core is pushed down to the shared LLC still remains in LLC or at-least part of it when the process resumes execution at the new core sharing the LLC within the local node. However, if the over-subscription is too high, then chances for the shared LLC to provide still some of the working set data of the migrated process or VCPU will be diminished since they are likely to be evicted by the competing processes or VCPUs.

Figure 3.1: Local Migration

Remote Migration - When a process or a VCPU migrates to a physical core belonging to a different node or socket of the system as shown in Figure 3.2. In such migration, migration-induced cache misses at the destination cannot be avoided and will either be satisfied by the cache hierarchy of the source core if the data is still to be found there or by the memory.

Figure 3.2: Remote Migration

We also classify both local and remote migration further based on the *timing* factor.

Live/Online Migration- a process or VCPU is actively executing on a core and the OS or VMM migrates it to different core, where it resumes execution immediately, as shown in Figure 3.3. In such a migration, the cache hierarchy at the source is likely to retain some of the data footprints belonging to the working set of the process or VCPU, and the opportunity may still exist to prefetch the needed cache lines into the destination core early to help reduce migration-induced cache misses.

Figure 3.3: Live and Delayed Local Migration

Delayed/Offline Migration- the OS or VMM migrates a process or VCPU to a different core while the process or VCPU is in idle or wait state. The OS or VMM may have *a priori* knowledge (i.e. processor queue length is too high) that the process or VCPU currently executing at a core will be migrated to a different one in near future when it goes to

either idle or wait state. The *a priori* knowledge of OS or VMM of such migration presents an opportunity to capture some info about the data footprints of the working set of the process or VCPU at the moment it is de-scheduled from the current core, and this info can be later used at the destination core to prefetch data likely to be consumed when the process or VCU resumes execution at the new destination.

Figure 3.4: Live and Delayed Remote Migration

Local migrations are likely to occur more frequently than remote migration (shown in Figure 3.4), as the OS or VMM are likely to give higher priority in trying first to keep the processes or VCPUs within a node in to order avoid complete loss of cache footprints. We address all of above migration types in our implementation study.

CHAPTER 4: PRESERVING TRANSLATION LOOKASIDE BUFFER CACHE FOOTPRINTS

One component of indirect overhead incurred either in a context switch or a migration in a multi-core server platform is due to page walks needed to fetch missing TLB entries when the process or a VCPU resumes execution. Even though TLB cache misses are not as frequent as L2 and LLC misses, still there were large number of TLB cache misses observed in an oversubscribed virtualization environment as shown in Figure 1.2 in Chapter 1

We have studied the preservation of the Translation Lookaside Buffer (TLB) cache entries over a migration/context switch event, to be consumed by the same process or VCPU later at a different physical core (migration) or at the same core (context switch). We focused on two key goals. The first goal is to illuminate TLB cache misses by preserving TLB entries, and the second is to include only those TLB cache entries in the preservation set that will be reused by the process or VCPU following a context switch or migration. The second goal recognizes that unused entries in the preservation set would increase the memory required for preservation, since there will be many processes or VCPUs in a large multicore server platform, and also would increase memory bandwidth pressure. The locality principle was the motivating factor in our consideration of the preservation methods, such as simple save and restore of TLB entries and more efficient prediction based preservation of TLB cache entries.

TLB cache entries and TLB caches have been saved and restored on a context switch in some computer platforms [14] where TLBs are software managed or in some VMM

implementations. But, we have not seen any characterization study of the benefits/efficiency of such saving and restoring of TLB entries. Furthermore, how the context switch intervals affect such scheme is also not known. When context switch is also accompanied by migration of the thread, TLB miss penalty could potentially increase significantly if the migration happens to a new socket/package which is not likely to have any relevant page table footprints in its cache hierarchy. We studied the impacts of saving and restoring TLB entries, particularly in the 1st level TLB cache, and how the context switch interval affects the accuracy/benefits of the scheme. We also build a predictor to predict the entries in the 1st level TLB cache that are likely to be used in the future, and only the predicted entries are saved and restored on a context switch (migration). We compare the performance benefits of saving and restoring TLB entries with and without the predictor.

4.1 Save and Restore Method

We performed memory trace driven simulation to study the impact of saving and restoring TLB entries at context switch (migration). First, we have used the PIN tool [25] to collect several memory load/store address traces of size 16 million from the SPEC CPU2006 [2] benchmarks. Each trace is started at a randomly selected memory instruction after the workload has been warmed up with a sufficient number of instructions (a few hundred million). Each trace was then fed through a two-level TLB cache simulator in which we simulate a context switch/migration at several randomly selected points in the trace.

We have tried several TLB cache configurations by varying both size and ways. We chose several context switch intervals consisting of 1, 3 and 6 million memory instructions.

Depending on the workloads, the instruction mix could contain anywhere from 10% to 40% memory instructions. So, our context switch interval with 1 million memory instructions could be anywhere from 2.5 million to 10 million instructions. Likewise, with 3 and 6 million memory instructions, a context switch interval will contain 7.5-30 million and 15-60 million instructions, respectively.

At each context switch point we record the level1 TLB cache content. We estimate the effect of simple save and restore technique on 1st level TLB cache entries by flushing the 2nd level TLB cache and monitoring TLB accesses, hit and misses at both levels of TLB cache hierarchy for a context switch interval. Without save and restore feature, context switch with migration would cause all TLB cache entries of the execution context to be lost for future execution. We then compute the savings from the reuse of restored TLB entries and note whether the saved entries were resident in the TLB cache hierarchy when they were first re-accessed after the context switch.

Saving and restoring entire set of valid entries in 1st level TLB cache, though simpler than having to use a more heavy weight predictor to smartly pick entries that will actually be reused soon, could be inefficient at times, depending on the workload. For example, if only a few of the entries are likely to be reused in future, or even if they are reused in future but not before they were already completely replaced by other newer entries from both TLB cache hierarchies (providing no cost savings), then saving entire TLB cache would be ineffective. Since we only take the snapshot of 1st level TLB cache at the time an execution context was de-scheduled, no previous history or temporal locality of entries present in the captured

snapshot is available. Thus, it lacks the ability to judge how recently each entry was accessed or to predict the likelihood of each entry being reused in the near future to provide any benefit of saving and restoring these TLB entries. Experimental data provided later also supports this. We have shown some snapshots of some 1st level TLB cache entries over several successive intervals of 1 million memory operations each using gcc trace in Figure 4.1. We see that pages such as 838a, 8488, 84d6, 84d7, and others that are accessed over more than one interval in a row also accessed soon in near future. On the other hand pages such as 8445, 8309, 8559 were not accessed over several intervals in a row, and also were very lightly accessed, and thus do not appear soon in future.

Therefore, we can further improve efficiency of TLB footprint reuse on a migration by tracking of hot and/or recent entries in the first level TLB cache and only selectively saving and restoring such entries predicted with highest reuse probability. So, based on the observations above we have looked to build predictors to pick TLB entries to save and restore with some confidence, as described next.

4.2 Aging Based Predictor

We break a context switch interval into several smaller sampling windows. A sampling window size is determined by the number of memory instructions in the window. A bit vector with size equal to the number of sampling windows in a context switch interval is maintained per TLB entry in the 1st level TLB cache. At the end of a sampling window, the bit vector is right shifted, and the most significant bit is set to 1 if the corresponding TLB entry is accessed in the just completed sampling window.

Figure 4.1: Snapshots of 1st level TLB cache entries over 11 intervals of 1m memory instruction each using gcc trace.

At the time of context switch the bit vector for each TLB entry is compared against a threshold value. If a vector value is larger than the threshold, then the entry is predicted to be reused in the near future and thus selected for save and restore. We reset the bit vector value upon replacement.

The Prediction scheme is further explained in Figure 4.2. In this case, we have a sampling window of 1K memory instructions and a context switch interval consisting of

three such sampling windows. At the end of the context switch, only two TLB entries (containing page addresses 0xaaaa and 0xdfdf) will be selected for save and restore if a threshold value of 1 is used. This prediction scheme excludes a TLB entry that has only been accessed way back in the past determined by the threshold values. We experimented with several sampling windows and threshold values for a given context switch interval size to get the best prediction.

Figure 4.2: Age based Predictor for save and restore

A global counter is used to keep track of timestamp. Timestamp is increased by 1 at the end of a sampling window and a new window is started. If a TLB entry is evicted from both TLB caches, it is stored into the page table with the current timestamp and its corresponding bit vector. If the entry is referenced again in near future, we shift its bit vector by the difference of the current global timestamp counter and the stored timestamp. Alternatively, we can assume that it has been too long since it was last referenced and always choose to reset the bit vector if we do not store the timestamp in page table.

The operating system or VMM scheduler could de-schedule the current execution context at any point of a sampling window, and not necessarily at the end of a sampling window. When a context switch happens in the middle of a sample window, we could either decide to update the bit vectors, as if the current window has ended, or use the value updated at the last sampling window. We compare the predictor performance using both choices.

When this execution context is re-scheduled later on a different hardware location (CPU/core), we refill the 1st level TLB cache with TLB entries that were predicted and saved previously at the time it was de-scheduled. We would avoid very expensive page table walks to access these TLB entries, as long as they are not replaced by a new entry before they are referenced. In case of 2-level TLB cache, even if such entries are moved to 2nd level TLB and are still present at 2nd level TLB when they are referenced, we would still avoid expensive page table walks involving trips to memory and coherent traffic.

4.3 Aged Plus Total Access

Instead of keeping a bit vector that monitors whether the TLB entry is only touched during a sampling interval, we keep a counter to keep track of total accesses during a context switch interval. We have sampling interval as defined before but we only shift the counter by one at the end of a sampling window. So, if a TLB entry was heavily accessed early in the context switch interval, but not accessed later at all, it may still have a chance of being selected, as opposed to entries that were lightly accessed early. At a context switch, if the counter value is larger than a threshold, then the entry is predicted to be used in the near future and is thus selected for saving and restoring at the next scheduled opportunity. So, in Figure 4.3, we will have selected only the TLB entries corresponding to addresses 0xbbbb and 0xdfdf if threshold used was 32.

Figure 4.3: Age Plus Total Access based Predictor

4.4 Simulation Results

We evaluated simple save and restore method with a 1st level 256 entry 4 way and 2nd level 512 entry 4 way TLB cache hierarchy on a subset of benchmark traces from SPEC CPU2006 [2]. We chose the benchmarks (C/C++) for which we had compiling tools available. We chose random context switch/migration points in the memory trace simulation. From the point of context switch/migration, we continued simulation for another 3 million memory operations, and the results are shown in Figure 4.4. Except for omnetpp, 60-100% of the L1 TLB cache entries selected were reused within the 3 million memory instruction window following the context switch/migration. Of the entries actually reused, most of them (~100%) were found in TLB cache hierarchy on their first access following the context switch/migration, except in the case of mcf. The benchmark mcf has a very large working set, and thus some entries saved were already replaced by newer entries from TLB cache hierarchy and therefore were not found when reaccessed for the first time.

Figure 4.4: Simple Save and Restore TLB method

Figure 4.5: TLB hit/miss info for Save and Restore TLB method

Just by checking the valid bit of the TLB entry, as opposed to selecting all 256 1st level TLB cache entries, we were able to reduce storage footprints for saving TLB entries for h264ref and omnetpp benchmarks. In a system with hardware simultaneous multithreading (SMT) support, further reduction would be possible by also checking the thread id bits along with the valid bit.

We also realized from Figure 4.4 that, except for bzip and mcf, not all entries selected were actually reused. If we could effectively predict entries that will be actually reused following context switch/migration, we could then reduce the footprint by selecting/saving only those entries. From Figure 4.4 we see that we can reduce the footprint by 40%, 29%, 14%, 34% and 97% for hammer, gcc, gobmk, h264ref and omnetpp respectively. We evaluate the two predictors we have discussed earlier in the next sections.

4.4.1. Aging based Predictor

We have evaluated the performance of aging based predictor for saving and restoring TLB entries in two different ways. In the first approach, we allow the predictor to look in the past beyond current context switch interval to build predictions, and then measure its performance by continuing to monitoring 3 and 6 million memory instructions from the point of the context switch.

First, we find a suitable number of intervals in the past to look and a threshold to use to find the best prediction. We used an interval size consisting of 1 million memory instructions. We have displayed our search results in Figures 4.6-4.9.

Figure 4.6: Entries predicted using different number of intervals/threshold pairs

Figure 4.7: Actually reused predicted 1st level TLB cache entries using different number of intervals/threshold pairs

Figure 4.8: 1st level TLB cache not selected for saving/restoring using different number of intervals/threshold pairs.

Figure 4.9: Actually reused not predicted 1st level TLB cache entries using different number of intervals/threshold pairs

We see that for interval/threshold pair of either (4, 0) or (5, 1), we achieved the best performance in terms of the number of entries predicted that were actually reused, while at the same time keeping false negatives (i.e. entries not predicted that actually wind up being reused) to be low. If we use higher threshold values, we predict fewer entries and our false negatives also increase. We also observed that the benchmark h264ref has high false negative rate, regardless of interval/threshold pair values; this is due to the streaming nature of its data accesses. We have used interval/threshold pair of (4, 0) for the rest of study of the predictor. We showed the results of the predictions in Figure 4.10-4.13.

Figure 4.10: Actually reused predicted TLB entries selected by the Aging based predictor

We see that the predicted entries' actual reuse increases with the size of context switch interval. There are no differences between simple save and restore method and this predictor scheme for the benchmarks bzip and mcf, as noted earlier, due to their large working sets that seem to not fit in TLB cache hierarchy used. However, the predictor selected considerably fewer entries than the simple save and restore method in case of gcc, h264ref, and omnetpp. Especially for omnetpp, we were able to predict the only entry out of 40 entries in 1st level TLB that was actually reused. From Figure 4.11, we see actually reused predicted entries were also mostly found in TLB cache hierarchy. For gobmk, TLB hit

rate decreases with increasing context switch interval, because the number of actual reused predicted entries goes down significantly as well.

Figure 4.11: TLB hits for actually reused predicted entries

We also need to evaluate the false negative predictions of the method as well, shown in Figure 4.12. For omnetpp and h264ref benchmarks, there are no false negative predictions and thus we have performed better in this for these two benchmarks than the simple save and restore method. However, we do have false negatives for hammer, gcc and gobmk benchmarks. It is worst for hammer as pointed out earlier due to its heavy streaming data accesses; we lose out opportunity to save a lot of clock cycles by not being able to predict

significant number of entries that actually get reused, and the opportunity lost increases with higher context switch interval.

Figure 4.12: False negative predictions

We also suffer lost opportunity with gcc, though somewhat less. Figure 4.13 also shows that most of the false negative predicted entries were found in the TLB cache hierarchy, which further raises the importance to understand these false negative predictions and the prediction to be useful overall.

Figure 4.13: TLB hits from false negative predicted entries' reuse

We have also experimented with different interval sizes other than 1 million memory instructions interval used above. We did not see any improvements with interval sizes such 500k and 2 million memory instructions.

In the second approach of the aging based predictor, we limit our past history to the size of a chosen context interval, and we try to reject those entries that were seen only at the very early stages of the current context switch interval. The motivation is from the fact that some of the aged TLB entries could be from a different thread or execution context. We experimented with dividing a context switch interval into smaller several intervals. For

example, a context switch interval of 1 million memory instructions, we experimented with 100k, 200k, 250k, and 500k memory instruction smaller sampling intervals. The number of bits required would be number of sampling intervals that fits in a context switch interval.

Figure 4.14: Prediction based on rejecting most aged entries.

We found 250k, 500k and 2m memory instruction sampling intervals to perform best with context switch interval sizes of 1m, 3m and 6m memory instructions, respectively.

In Figure 4.15 we see the prediction accuracy for context switch interval of 1 million memory instructions. As before, for h264ref and omnetpp the predictor performs well, reducing the storage footprints significantly (by 46% and 97%) and the false negative predictions are also negligent. For, hmmer, gcc and gobmk, while the number of predicted entries were much smaller than the total entries selected by the simple save and restore method, the true prediction percentages were not high enough, meaning we could have

reduced the storage footprints more if we had better prediction. Moreover, false negatives were also high; for example, gcc has 21.88% of the total entries as false negative entries. There are no differences for bzip and mcf benchmarks from prediction.

Figure 4.15: Aging predictor entries for context switch window of 1m memory operations

Except for mcf and omnetpp, false negative predictions were always found on the TLB cache hierarchies as evident from Figure 4.16. The prediction improves with higher context switch intervals as shown in Figures 4.17 and 4.18 for context switch intervals consisting of 3m and 6m memory instructions respectively.

With higher context switch window (6M) as shown in Figure 4.18, the predictor accuracy improved for h264ref, hmmer, and gcc. We still have some false negative predictions for hmmer and gcc. We will see next if we were able to improve upon these false negative predictions using the Aging plus total access predictor method.

Figure 4.16: TLB hits for false negative predictions

Figure 4.17: Aging predictor entries for context switch window of 3m memory operations

Figure 4.18: Aging predictor entries for context switch window of 6m memory operations

4.4.2 Aging plus Total access Predictor

We have tried to evaluate different sample interval sizes and threshold values for the counter used. From our early evaluation, it was found that a 500K interval with a threshold value of 2 for a 6-bit counter produces the best predictions. We have captured the predictor performance for only a 6M memory instruction window following a context switch/migration event. The prediction accuracy and TLB hits from this method is shown in Figures 4.19 and 4.20 respectively. We compared this predictor's performance with previous aging-only save and restore predictor, and we see that it performs similar to aging-only predictor, except for hmmer and gcc. For hmmer and gcc, it reduces false negatives but it also increases the storage footprints by a predicting larger set of entries.

Figure 4.19: Comparison of Aging only, aging⁺ total access and simple save and restore

Figure 4.20: TLB hits comparison for Aging vs. Aging plus Predictor

4.5 Implementation Consideration

Since the majority of the indirect overhead incurred following a migration (context switch) is due to extra cache misses, as opposed to TLB misses, we have not attempted a full system implementation study of the above methods of preserving TLB footprints over a context switch or migration. Nonetheless, we would like to point out the key aspects that any implementation would need to provide. We would need to provide hardware support to read and write a TLB entry from the TLB cache hierarchy. We would also need to maintain couple of queues. The first queue is filled with TLB entries to be preserved. Once the queue is full, we need to package all the TLB entries and send a message either –

- i) To the local directory to be stored at a reserved memory indexed by the process or VCPU identifier. We could use part of the region of memory [17] that is concealed from typical software and used by the Virtual Machine Monitor (VMM) as software cache to store the selected TLB footprints. Each footprint can be accessed using the process id or virtual CPU ID. The storage could use a bitmask equal to the total number of TLB cache entries of the TLB cache we are interested to save and restore on context switch and or following by migration. The directory eventually will respond to a new message seeking to receive the preserved TLB entries identified by the supplied process/VCPU identifier contained in the message.
- ii) To the destination processor directly for preservation. .

The second queue would be filled with TLB entries contained in a special message arriving either from a source processor (in case of live migration) or a directory (in case of delayed migration).

Thus, we would need to augment messaging protocol with additional message types to transfer TLB entries. There also has to be consideration on how to deal with TLB shoot-down, in case of a page is remapped by the VMM or OS. The benefit (cycles saved) is expected to outweigh the cost (cycles) involved in saving and restoring TLB entries. This is because, if we assume we can issue a TLB read or write on every clock cycle, the total cost of save and restore would be equal to $((\text{read} + \text{write operation}) * \text{the maximum entries selected})$ or $(2 * \text{maximum selected entries cycles})$. The potential savings could be very significant because each TLB miss could trigger a very expensive page table walk that would take few hundred cycles when the context switch is coupled with migration to a core with less chances of having any of necessary page table entries cached in its cache hierarchy. The page table walks on a virtualized platform with nested (AMD [11]) or extended (Intel [12]) page table support would need an extra hop and thus would increase the cost even more.

4.6 Summary

Each saved entry that is reused following a migration saves a processor core from taking an expensive page table walk. A page walk could mean potentially up to 4 expensive trips (100 or more cycles each) to system memory and up to 5 trips in hardware virtualization environment with nested or extended page table support. With increasing number of cores per processor, and high oversubscription level of virtual CPUs to logical processor cores in

virtualization environment, the storage footprint requirements could be very large for a system with a large number of virtual CPUs. We have seen that relatively simple predictors can reduce the storage footprint by 20-30%. The use of predictors would be even more important in heterogeneous computing platforms. With the migration rate already in the thousands/logical processor/sec in small multi-core platforms (8 cores or higher total), it will likely to get only worse with larger multi-core platforms that will soon appear. Therefore, reducing the impact of migration from TLB misses would enable operating system and virtual machine monitor schedulers to aggressively schedule the threads or VMs and improve overall resource usage.

We have seen from the experimental data that the simple save and restore method, where it picks any valid entries in the TLB cache, is not always effective. Often there are 30% or more entries that are not reused in the context switch intervals we have experimented with. The predictors proposed removed most of those entries, but there are instances where they either produced some false positive entries or expensive false negative entries. The design methodologies for the predictor proposed remains valid even if we consider different context switch intervals, either smaller or larger. We have limited our study to only memory traces. However, a cycle accurate simulation with a real operating system scheduler to trigger a context switch or migration would allow a study of the performance impact of preserving TLB footprints utilizing the smart predictions described.

There are other research areas that can be pursued in the future that are beyond the scope of this dissertation. For example, one could tie the hot TLB entries identified through

the predictors with the prefetch table entries, and selectively save only those prefetch table entries that are likely to be used soon following a context switch or migration. Similarly, one could also use the predicted TLB entries to select branch prediction table entries for saving and restoring on a context switch/migration.

CHAPTER 5: REDUCING MIGRATION-INDUCED CACHE MISSES

One easy solution to avoid migration-induced cache misses would be to save away the entire cache in memory before migration at the source CPU, and then restore it post-migration at the destination CPU. Assuming an inclusive cache-hierarchy, the L2 cache is usually designed to be large enough to hold the working sets of the targeted workloads for the system with either a 2-level (L1-L2) or a 3-level (L1-L2-L3) cache hierarchy. The L3 cache, if it exists, is typically shared across multiple physical cores of a node within server system. Although large L3 cache could help reduce L2 miss penalty it does remove entirely all L2 misses for highly over-subscribed platforms, as shown in figure 1.3 (chapter 1), and these L2 misses still incur large miss latency, as shown figure 1.x (chapter 1). Thus, if we target to save away the entire L2 cache content of a process prior to migration while the footprint is still hot, and subsequently restore the data at the destination L2, we could eliminate the migration-induced cache misses.

This is an expensive solution, however. First, L2 cache could be very large, as shown in Table 5.1. Second, some cache lines are essentially dead — they are not accessed again after the migration, but they have not yet aged out of the cache. This is evident when we compare migration-induced misses for 2MB and 4MB L2 cases, as shown in Figure 5.1. For several benchmarks (bzip2, h264, hmmer, and sjeng), saving and restoring twice as much data has little or no positive effect.

Table 5.1: Typical L2 Cache Size of Intel Xeon Servers [26]

<i>Processor Family</i>	<i>L2 Cache Size</i>
Intel® Xeon® Processor 3040	2MB
Intel® Xeon® Processor 3070	4MB
Intel® Xeon® Processor X7550	2MB
Intel® Xeon® Processor E5205	6MB
Intel® Xeon® Processor X5667	1MB
Intel® Xeon® Processor E7540	1.5MB

Figure 5.1: Migration-induced L2 misses for SpecCPU2006 C/C++ benchmarks with migrations at every 2 million instructions (after warm-up).

To capture this tradeoff between cost and benefit, we define efficiency metric to measure how many of the saved cache lines actually help in reducing post-migration cache misses in a post-migration window (PMW) of execution:

$$Efficiency = \frac{\text{reduced L2 misses in PMW}}{\text{L2 cache lines saved pre-migration}}$$

The ideal efficiency would be 100%, since every cache line saved would be accessed after migration, eliminating a migration-induced miss. Reduction in efficiency comes from two sources: useless cache lines (described above), and restored cache lines that are pushed out of the cache before being touched again (due to limited capacity).

Figure 5.2: Efficiency: Ratio of removed migration-induced L2 misses and the number of L2 entries saved and restored. For this figure, 100% of the L2 is saved and restored, and the post-migration window (PMW) of execution is 2 million instructions.

Figure 5.2 shows the efficiency of saving the entire L2 (2MB, 4MB) before migration and restoring it post-migration and measuring the migration-induced L2 miss reduction in a

PMW of 2 million instruction executions from each benchmark. Even with mcf's and bzip2's large migration-induced L2 misses, we have only touched less than 30% of the restored cache lines in L2. For most benchmarks, 16% or less of saved L2 cache lines were touched for 2MB L2 within the PMW of 2 million instructions. The working set of all benchmarks, except mcf, seemed to have fit entirely within a 2MB L2 and thus showed diminishing L2 miss reductions with a larger L2 of 4MB. The majority of the saved L2 lines would still be untouched; saving and restoring them results in extra storage and bandwidth pressure in memory, which could be better used for other purposes (e.g., reduced page swapping, demand misses).

Given that only a small part of the entire saved L2 cache lines would actually be touched in a PMW, the cost and overhead may be too high to save and restore the entire L2 to obtain performance benefit from the reductions of migration induced L2 misses. So, this approach is very inefficient from the point of storage and memory bandwidth requirements, as well as the time required to save and restore the entire L2.

In this chapter, we explore techniques to improve efficiency of the mechanism of saving and restoring L2 cache lines on a migration by selecting fewer L2 cache lines but still reducing migration-induced L2 misses as much as possible. The idea is to select the most likely cache lines to be accessed again by the migrated thread. Intuitively, because of temporal locality, this means selecting recently-used cache lines.

The simplest approach in a set-associative L2 is to select the most-recently-used (MRU) line in each cache set. For example, consider a 2MB L2 that is four-way set

associative. Each set has four cache lines, and we select the MRU line from each set to save and restore. We call this the *Local MRU* approach.

However, not all sets are accessed equally. Some sets might be relatively stale, meaning that their MRU lines were not all that recently accessed. An alternative, then, is to keep track of the *Global MRU* lines — the ones most recently accessed in the entire cache, not restricted to one per set.

In Section 5.3 we evaluate these techniques in terms of their efficiency and how effective they are at reducing migration-induced misses and improving CPU performance. We find that Local MRU works nearly as well as Global MRU, and it is much easier to implement.

5.1 Simulation Methodology

To measure migration-induced misses, we simulate a Pentium 4 system with the Simics [23] full system simulator. We run an unmodified Fedora Core 6 distribution of the Linux Operating System on the processor. The processor has a scalar in-order issue pipeline with a 3GHz frequency. All instructions take one clock cycle, except for loads and stores, whose latencies depend on the memory hierarchy.

The memory hierarchy has two cache levels. The L1 instruction cache is 16KB, 2-way set associative, and has a 3-cycle access latency. The L1 data cache is 16KB, 4-way set associative, and has a 3-cycle access latency. The L2 cache is 16-way associative, has a 10-cycle access latency, and its size is either 2MB or 4MB. All caches have a block size of 64 bytes, implement write-back policy, and use the LRU replacement policy.

Note that there is no L3 cache in our simulations. Even though modern multicore systems usually have large shared L3 (LLC), a process/thread/VM running will be competing with other processes/threads/VMs running on cores that are sharing the LLC, and it is likely that its allocation of LLC capacity will be limited and the high over-subscription of threads to a core will still cause L2 misses to miss the LLC as well and incur large average miss penalty. Therefore, we model a typical 2-4MB L2 with average miss latency value close to observed L2 miss latency value in a three level cache hierarchy configuration found in modern hardware.

We run a selection of the SpecCPU2006 C/C++ benchmarks. We chose particularly those benchmarks that have characteristics of VDI workloads. The VDI workloads typically consist of computation such as compression, video encoding, streaming, random accesses, etc. For each benchmark we first warm up the system, including the cache, with over half a billion instructions. Then we simulate over 20 different context switches (CS). We treat each context switch as a migration by flushing the TLBs and cache hierarchies, modeling migration to a completely new CPU with no previous history for this process. Between each context switch, we continue simulation for about 2 million instructions, and we call this the post migration window (PMW). We track the processes during the PMW. If we determine that other system processes not related to our benchmark ran during the post migration window, we do not include those statistics in our analysis. (In other words, the PMW is 2 million instructions from the application process.)

We then compute the average of L2 misses across the simulated migrations and obtain the average L2 misses incurred in a single PMW. To identify migration-induced L2 cache misses, we repeat the above simulation but we do not flush L2 at the CS points. This models saving the entire L2 from the core we are migrating from and restoring it at the core where we are migrating to. The difference of L2 misses incurred in a PMW from the two simulations gives us the migration-induced L2 misses.

5.2 Efficiency Improvement

To improve efficiency, we focus on selecting those L2 entries that are likely to be reused upon migration in a PMW. Instead of saving and restoring the entire LLC, we want to choose a subset of lines to save, reducing storage and bandwidth costs, while still eliminating most of the migration-induced L2 misses. In addition to the efficiency metric defined earlier, we have defined another metric called *migration induced L2 miss reduction ratio (MI-MRR)* to determine what fraction of migration-induced L2 misses are eliminated, compared to saving and restoring the entire L2 as shown below -

$$MI-MRR = \frac{\text{Migration-induced L2 miss reductions in PMW from saving selected cache lines set}}{\text{Migration-induced L2 miss reductions in PMW from saving all cache lines set (i.e. entire L2)}}$$

We devised three schemes for saving a reduced set of L2 cache lines, described in the following subsections. Focusing on simplicity, we choose to use already-existing temporal use information that is typically tracked for the cache's replacement policy.

5.2.1 Local MRU Selection

In local MRU based selection we select the valid MRU entry (if it exists) from each set in the L2 to save before migration. This is essentially saving one “way” of an n -way associative cache, but selecting only the MRU entries from each set. We can increase the number of entries by selecting n MRU entries from each set in the LLC, with n being 2, 3, 4, etc. This scheme can be easily implemented in hardware, if not already available in a cache.

5.2.2 Global MRU Selection

In the global MRU based selection we select the n most recently-used entries from the entire L2. This scheme would be more expensive and complex in terms of hardware resources to implement, compared to the local MRU scheme. We have tried several settings of increasing values for n .

5.2.3 Regional MRU Selection

In the Local MRU selection, we select the MRU entry from each set, so we would end up with 2048 entries for a 2MB L2 and 4096 entries for a 4MB L2 and so on. If we want to select fewer entries (for efficiency reasons), we can either skip some sets (every other, every 3rd, or every 4th,etc) or find an MRU from a group of successive sets as opposed to selecting the MRUs from a single set. We call the selecting of a MRU line from cache lines that span over more than a set “regional” MRU based selection. We applied this method for selecting 256, 512, and 1024 entries for both 2MB and 4MB L2, and for 2048 entries for 4MB L2.

Figure 5.3: bzip2: 2MB L2

Figure 5.4: bzip2: 4MB L2

Figure 5.4: h264: 2MB L2

Figure 5.6: h264: 4MB L2

Figure 5.7: hmmer: 2MB L2

Figure 5.8: hmmer: 4MB L2

Figure 5.9: mcf: 2MB L2

Figure 5.10: mcf: 4MB L2

Figure 5.11: omnetpp: 2MB L2

Figure 5.12: omnetpp: 4MB L2

Figure 5.13: sjeng: 2MB L2

Figure 5.14: sjeng: 4MB L2

5. 3 Simulation Results

In Figures 5.3–5.14, we show the comparison of the three schemes outlined earlier for each benchmark, with L2s of size 2MB and 4MB. Each figure contains the efficiency (left two bars) and MI-MRR (right two bars) as we increase the number of L2 cache lines to be saved and restored on a migration.

For a given selection set size, either Local MRU or Regional MRU is used, but not both. This is because Local MRU selects at least one MRU line from each set, so there is a limit to how small a set can be selected. For smaller than 1K or 2K lines (for 2MB and 4MB, respectively), Regional MRU is used to select an MRU from a collection of sets.

Even though efficiency decreases, the overall performance gets closer and closer to the full-L2 save and restore as the selection set increases, as we would expect. In several cases, we get at least 50% of the benefit by only saving 12.5% of the L2 cache lines (4K lines for 2MB L2, 8K for 4MB)

We also see that the Local/Regional MRU scheme is generally as good or better than Global MRU. Therefore, there isn't compelling evidence that the increased complexity of tracking global reuse patterns would be justified.

In case of h264, the Local MRU scheme outperformed the Global MRU scheme significantly due to the streaming characteristic of h264. In such streaming scenario it may be best just to use prefetching rather than utilizing the accelerator to save and restore L2 data. Accordingly, stream prefetcher activity should be considered in enabling the hardware to

select cache lines from L2 for save and restore on migration for a thread/process or a virtual CPU in case of VM.

We also note that the measured efficiency is greater than 100% in a couple of instances: bzip (512 lines) and omnetpp (256 and 512 lines). The same is true for MI-MRR in the case of sjeng. This is likely due to slightly different migration points between the full-L2 case and the limited selection set case.

For most benchmarks, efficiency either decreases slightly or remains the same with the larger L2 size, indicating that their working set sizes fit within the 2MB L2. In the case of mcf, due to its large cache footprints and random accesses from pointer chasing, we see that efficiency actually increases with larger L2 for the same number of cache lines saved. Efficiency is high when a small number of L2 lines are selected, especially for Global MRU. Except for h264, efficiency is almost always over 70% for L2 selection set smaller than 1024 entries for 2MB L2 and 2048 for 4MB L2. Even though the efficiency is high for smaller selection set, the resulting performance benefit is not expected to be high given the small number of eliminated L2 cache misses.

5.3.1 Impact of L2 Cache Line State

We have also looked into how the cache line states, i.e., shared or modified (dirty), impact the migration-induced L2 miss reduction. We show in Table 5.2 for each selection set size (i.e. 256, 512, 1024, etc.), the number of unmodified cache lines (shared/exclusive), the number of modified (dirty) cache lines and what percent of such cache lines were touched in PMW of execution.

Table 5.2: Decomposition of selection set and impact of cache-line state on usefulness

<i>L2 entries chosen</i>	256	512	1024	2048	4096	8192	16384
<i>Omnetpp</i>							
<i>Shared lines chosen</i>	191	387	777	1541	3097	4555	5545
<i>Dirty lines chosen</i>	65	125	247	507	999	3637	10839
<i>Shared lines hit %</i>	94	99	94	81	74	43	37
<i>Dirty lines hit %</i>	92	91	69	65	48	36	12
<i>Valid lines hit %</i>	93	97	88	77	67	40	20
<i>Sjeng</i>							
<i>Shared lines chosen</i>	179	362	663	1237	2493	4690	7859
<i>Dirty lines chosen</i>	77	150	361	811	1603	3502	8525
<i>Shared lines hit %</i>	97	94	82	58	39	27	20
<i>Dirty lines hit %</i>	94	86	69	48	29	15	7
<i>Valid lines hit %</i>	96	91	77	54	35	22	13
<i>Hammer</i>							
<i>Shared lines chosen</i>	154	293	502	777	1133	1634	2267
<i>Dirty lines chosen</i>	102	219	522	1271	2963	6558	14117
<i>Shared lines hit %</i>	98	98	88	83	73	62	49
<i>Dirty lines hit %</i>	64	33	14	6	5	6	3
<i>Valid lines hit %</i>	84	70	50	35	24	17	9
<i>h264</i>							
<i>Shared lines chosen</i>	120	313	756	1575	3361	6695	13150
<i>Dirty lines chosen</i>	136	199	268	473	735	1497	3234
<i>Shared lines hit %</i>	35	39	37	32	16	8	11
<i>Dirty lines hit %</i>	72	74	71	55	42	22	21
<i>Valid lines hit %</i>	54	53	46	37	21	11	13
<i>Bzip</i>							
<i>Shared lines chosen</i>	195	431	813	1618	2850	3947	5654
<i>Dirty lines chosen</i>	61	81	211	430	1246	4245	10730
<i>Shared lines hit %</i>	86	84	77	74	69	67	55
<i>Dirty lines hit %</i>	85	79	58	43	33	28	30
<i>Valid lines hit %</i>	86	83	73	68	58	47	39
<i>Mcf</i>							
<i>Shared lines chosen</i>	164	330	681	1349	2709	5394	10640
<i>Dirty lines chosen</i>	92	182	343	699	1387	2798	5744
<i>Shared lines hit %</i>	64	64	64	52	43	37	32
<i>Dirty lines hit %</i>	67	62	62	53	48	44	44
<i>Valid lines hit %</i>	65	63	63	53	44	40	36

Across all benchmarks we see fewer dirty cache lines selected, compared to shared/unmodified cache lines. Except for h264, we see that, compared to dirty cache lines, shared/unmodified cache lines have a higher post-migration hit rate. The gap in hit rates of these two cache line types increases with larger selection set from L2. We see similar patterns with local and global MRU based selection scheme. This data highlights that a smaller and more efficient set of L2 lines can be targeted by selecting only the shared/unmodified MRU cache lines if memory storage and bandwidth concerns restrict the L2 lines selection set size.

5.3.2 CPI Improvement

We measured the improvement (reduction) in CPI (clock cycles per Instruction) reduction as shown in the figures 5.15 and 5.16 for 2MB L2, and figures 5.17 and 5.18 for 4MB L2. (Selection sets of less than 1K entries are not shown, because they had very little impact on performance.) For both L2 sizes, overall CPI decreases with selection set size for both local and global MRU. Most benchmarks except h264 showed noticeable improvement starting at selection set size of 2K cache lines (256KB). Even though the maximum reduction in CPI is observed when the entire L2 is saved and restored, the CPI reduction per cache line saved is much higher around 4K–8K cache lines, and the maximum CPI reduction per cache line occurs around 2K cache lines. The CPI reduction ranges from 2% to 7% for selection set size 8K except for h264, for which the reduction is only around 1% (because of its streaming behavior).

The CPI reduction rate also depends obviously on the working set size of the benchmark. For example, mcf (with large working set) experiences smaller CPI reduction per cache line for 2MB L2 than 4MB L2. The cache model we have used did not model the dependence of L2 miss penalty on available memory bandwidth. It rather assumed a constant penalty of 200 cycles. The L2 miss penalty is likely to be higher when available memory bandwidth is low and thus the potential CPI reduction is likely to be higher when memory bandwidth pressure exists.

Figure 5.15: CPI reduction using Global MRU selection: 2MB L2.

Figure 5.16: CPI reduction using Local MRU selection: 2MB L2.

Figure 5.17: CPI reduction using Global MRU selection: 4MB L2.

Figure 5.18: CPI reduction using Local MRU selection: 4MB L2.

5.3.3 Sensitivity to PMW Size

All of the above data analyses were based on a PMW size of 2 million instructions. This was based on experiments of simulating two benchmarks on a single Simics processor, in which we found that the average context switch window was around 2 million instructions. However, the benefit is likely to increase for a smaller PMW, because the reduced misses are amortized over fewer instructions.

. To quantify this effect, we computed the geometric mean of CPI for each selection set size and compared three different PMW sizes: 2M instructions (our baseline), 1.5M instructions (a 25% decrease), and 1M instructions (a 50% decrease). The results are shown in Figure 5.19, for Local-MRU selection policy on a 4MB L2 cache.

Figure 5.19: CPI reduction for various PMW sizes (Local-MRU, 4MB L2).

The performance metric (reduction in CPI) is improved if we consider a smaller post-migration window of instructions. For 1.5M instructions, the difference is modest, but the difference between 1M and 2M instructions is 33–41% in the data points shown in the figure. (We found a general difference of around 20–40% across all variations of selection set size and selection method.)

Therefore, most of the performance benefits of reducing migration-induced misses come soon after the migrated thread resumes execution. This is to be expected, because a longer running time allows the caches to warm up and regain the original performance level. As migrations become more frequent, as shown by Figures 1.1-1.3 (Chapter 1), the performance impact of migrating cache history becomes more pronounced.

In virtualization platforms with VCPUs over-subscribed to CPUs, the PMW is likely to be smaller. In such environment, each I/O operation within a guest VM results in an exit into the hypervisor kernel. The hypervisor handles the I/O operations on behalf of the guest

VM and is likely to take such opportunity to invoke the scheduler, increasing the likelihood of migration of virtual CPUs, and also increasing the potential for performance boost from reduction of migration-induced L2 misses.

5.4 Implementation Issues

The evaluation in Section 5.3 does not consider the actual time to transfer the data from the original CPU to the target. It simply assumes that the selected cache lines are available at the L2 when the migrated thread begins executing. In an actual implementation, there are timing, storage, and bandwidth issues, and data and threads do not migrate instantaneously.

Another concern with saving and restoring data is the issue of cache coherence. Though our study only considers single-threaded programs right now, there is no reason why a program couldn't include multiple threads that share memory. Even in the context of single-thread programs, multiple migrations can leave copies of data in multiple caches at multiple levels of the memory hierarchy. Taking a data copy out of the normal coherence protocol, either to save it for later or to move it to another cache, creates the possibility of stale, non-coherent data.

For this reason, we propose to save/migrate *tags* instead of the cached data. The tags will be used by a prefetcher at the target L2 to move data locally, fully participating in the cache coherence protocol. Details of this implementation approach are described in Section 5.5.

5.5 Implementation Study

Figure 5.20: Modeled system: collection of multicores with on-chip banked L2 and banked off-chip memory. Networks can be direct (NUMA) or indirect (UMA). This particular study uses two chips, with one CPU and L2 per chip, and two memory banks.

While the previous studies are realistic in how a reasonable set of L2 cache lines can be chosen for migration, they are idealized in the sense that there is no cost for moving data between L2s, and there is no consideration of when and how the data is moved. In this section, we describe a prototype implementation that begins to address those issues.

The first issue is one of timing: When does the system know that it should move data from point A to point B? The point at which a thread is switched out is the ideal time to capture the data, because that's when the data is resident in the cache. However, the OS may not choose to migrate the thread until the next time it is ready for scheduling; this can be many cycles later, and much of the cached data may be already gone. We could choose to save and restore the data on every context switch, but this much overhead could eliminate much of the benefit, especially in the presence of a large L2.

There are two interesting scenarios: (1) live-migration - the thread is migrated immediately to another CPU, and (2) delayed-migration- the thread is simply swapped out, and a migration decision will be made when it is re-scheduled. In the first case, we can start moving data to the destination immediately. In the second, the cached data (or tags) must be stored in memory for later retrieval. Both these migration types have been explained in detail in Chapter 3.

The second issue is the actual movement of data from one L2 to another. Because of concerns about cache coherence, described in Section 5.4, we choose to send only the cache tags, not the data, to the destination L2. A prefetcher will then be used to perform coherent transfers of data. We called this migration prefetching.

Figure 5.20 shows the system architecture assumed for this study. Each chip (or socket) contains multiple processors, connected to a shared, banked L2. An on-chip packet-switched interconnection network provides the connection between the processors (with private coherent caches) and the shared L2. A separate packet-switched network connects chips to each other and to a banked main memory. We assume a directory-based cache coherence protocol, and the system-level directories for inter-chip coherence reside with the memory controllers.

We trigger a real migration of a process from one CPU to another by changing the affinity of the process by the OS-supplied affinity setting mechanism. We then used a Linux process tracker within SIMICS to track the transition of a process from one CPU to another

and detect the triggered migration. The entire process is explained below for both live and delayed migrations.

Figure 5.21: Triggering live and delayed migrations

Live Migration, shown in Figure 5.22 from step 1 to step 3, is explained with the following detailed step actions-

1. Launch and Set the affinity the of the process to cpu P1
  - a. `taskset -c 1 -p 1901`
  - b. The process id is assumed here to be 1901
2. Change the affinity of the process to cpu P3 to cause a migration

- a. Taskset -c 3 -p 1901
3. Process tracker detect the process migration on cpu P3

Similarly, delayed migration shown above from step4 to step 5 is also explained with the following detailed step actions-

4. Change the affinity of the process to cpu P2 to cause a migration
  - a. Taskset -c 2 -p 1901
5. Process tracker detect the process migration on cpu P2

We now discuss the implementation details and related results in section 5.5.1 (live-migration) and 5.5.2 (delayed-migration)

5.5.1 Live-Migration

At each L2, we add a migration/prefetch engine shown in the Figure 5.22 . When a migration is triggered, the source L2 chooses the appropriate set of cache line tags (depending on one of the policies described above) and queues them into the outgoing queue for sending to the destination L2. When a certain number of tags have been queued (16, in our experiment), a network message containing the tags is sent to the destination L2. Thus, migrating a set of 2048 cache lines requires $2048/16=256$ messages to be sent across the inter-chip network.

When a migration message arrives at the destination L2, the tags are copied into the prefetcher (incoming) queue. The prefetcher then issues prefetch requests to the memory system as quickly as possible — one request per cycle in our experiment. We only issue

prefetch load requests, although it would be possible to include dirty-bit information in the cache tags to serve as hints for issuing prefetch-exclusive requests. The data is only prefetched to the L2. Demand misses are used to move the data into the CPU's local L1 cache. We augment the cache coherence messaging protocol to handle the prefetch requests as shown in Figure 5.23. The states marked in green, and all the associated transitions, were added to allow migration prefetches while preserving cache-coherency across the system.

Figure 5.22: Addition of Incoming and outgoing queue to L2

The operating system or VMM scheduler issues special system command² when it is ready to migrate a process/VMM specifying the process/VM identifier, source and destination CPUs to initiate the tags transfer from source to destination followed by prefetching of the addresses specified by the tags in the second queue at destination CPU. If

² The special command is implemented in the simulator, not in the target machine's OS.

the migration were to occur between two CPUs sharing last level cache, then no migration message would be sent. In such cases, given the live-migration, the last-level cache is likely to retain some of the working set data and be able to supply them for consumption by the destination CPU within same socket.

Figure 5.23: Cache-coherence Protocol modification to accommodate migration prefetch

We have evaluated a two-socket system with NUMA memory as shown in Figure 5.20. Each socket consists of a single Pentium 4 CPU, L1 (4-way, 64KB) and L2 (8-way, 2MB) caches. Each socket is expandable to multiple CPUs with L2 being shared among

the CPUs. The CPU was simulated with the SIMICS full system simulator. The cache and memory was simulated using Ruby from the Wisconsin GEMS toolset [24].

We ran as before an unmodified Fedora Core 6 distribution of the Linux Operating System on the processor. We have run the same subset SpecCPU2006 C/C++ benchmarks used in the earlier sections. For each benchmark, we first launch the benchmark on one CPU (P0), and warm the cache with over half a billion instructions. We then initiate a migration to the remote CPU (P1) by changing the CPU affinity of the process to P1. When we detect the execution of the process on P1, we issue a special migration command⁴ on P0, which initiates the migration of tags from P0's L2 to P1's L2.

We then continue simulation for a PMW of 2 million instructions of the process on P1. We compared the results from above experiment against the results from the baseline system without our migration hardware support. We repeat the above experiment for each benchmark with different warm up periods and take statistical average.

Figure 5.24: Reduction of LLC misses in two-core migration experiment.

⁴ The special migrate command is implemented in the simulator, not in the target machine's OS.

Figure 5.25: Reduction of CPI in two-core migration experiment.

Figure 5.24 shows the reduction of total cache misses in PMW resulting from prefetching over the baseline system without our migration hardware support. Migration messages consist of either all valid tags from the L2 cache or a subset made of Local MRU tags from the L2 cache. Both tag selection schemes were able to eliminate a significant number of cache misses. Selecting all tags reduced a lot more cache misses for bzip2, compared to Local MRU. On the other hand, Local MRU eliminated more misses for hmmer, h264 and omnetpp benchmarks. These three benchmarks have smaller working set in PMW compared to bzip2 and likely suffered the eviction of some useful data footprints due to prefetching of the larger cache data footprint from P0 following a migration with entire last level cache valid tag set. If the working set is relatively small then some of the valid tags in the L2 cache of P0 may come from other system jobs and thus would cause negative impact.

We could augment the tag bits to include a process/VM identifier info and thus include only tags that belong to migrating process in comprising migration messages.

The performance benefit in terms of CPI reduction over the PMW is shown in Figure 5.25. The performance benefit is derived mainly as a result of the reduction in cache misses, despite all the overheads associated with migration messages and prefetches. The Local MRU tag prefetching improved CPI, ranging from 1.5% to 12%. Except for bzip2, the Local MRU tag set prefetching performed as well or better compared to entire L2 cache set (valid entries only) prefetching from P0. Even for bzip2, Local MRU improved CPI by as much as 12%. We did not see much improvement for mcf given its very large cache footprint and rather unpredictable cache access patterns. The benchmark omnetpp has much smaller footprint and hence we did not see much improvement from reduction of cache misses.

Figure 5.26: Increase in message traffic (over all network links).

The performance benefits observed above were realized despite the messaging and prefetching overhead involved in sending the tags from source CPU to destination CPU and

subsequently prefetching the same tags at destination CPU upon arrival of migration messages. The overhead incurred in terms of extra messages transferred over all links in the interconnection network is shown in Figure 5.26. These messages are “extra,” compared to the base case without migration-based prefetches — i.e., messages that would not be required to satisfy the demand misses incurred during the PMW.

The higher the extra messages, the more ineffective the prefetches from migration messages is. Sending all valid L2 tags from P0 to P1 generated a very large number of unused extra messages and likely caused extra conflict misses in PMW, thus making it ineffective. On the other hand Local MRU tags were effective in keeping low the extra messages, indicating migration messages were consumed by the benchmark in the PMW. We believe we can further reduce the extra messages once we augment the tag bits with the process/VM identifier and send only tags with matching id during migration. Also, additional reduction of message overhead is possible if we filter cache-line selection set based on coherence state of the cache-line as the data in Table 5.2 suggests.

5.5.2 Delayed-Migration

At each directory we add a tag receive/send engine. The directly also maintains a small table called tag-table to hold limited number of 2-tuple entries. Each 2-tuple entry contains thread/VCPU identifier and a memory address pointer. A specific local memory region is reserved to hold tags and relevant info of a thread/VCPU. The memory address pointer points to the local memory where all tags info for a given thread/VCPU is stored or

retrieved if exists. Figure 5.27 shows a directory holding tags info from 4 different threads/VCPUs.

Figure 5.27: Reserved directory memory for local L2 cache line storage

When a thread/VCPU is de-scheduled (due to either I/O, expiration of its time quantum or load balancing) and is not likely to be re-scheduled immediately either on the same CPU or a different CPU (local or remote), the chances of retaining any working set footprints in the cache hierarchy including the last-level shared cache is very slim due to the high over-subscriptions of threads/VCPUs to hardware threads. The OS/VMM scheduler could notify the source L2 to send the messages containing tags of the de-scheduled thread/VCPU to the local directory. Just like live-migration the engine at L2 chooses the appropriate set of cache line tags (depending on one of the policies described earlier) and queues them for sending to the local directory. When a certain number of tags have been queued, a network message containing the tags is sent to the local directory along with

message header containing the thread/VCPU identifier. We experimented with different queue size and found queue size of 16 to be most effective for our experimental setup.

We store tags from a particular thread/VCPU into a contiguous memory space. The tag-table helps us to do that by remembering the local buffer space used for each thread/VCPU. When a message arrives, it's looked up in the tag-table. If an entry is found, the new tags are appended to the memory associated with that thread. If not, a new buffer is allocated and added to the tag-table.

The OS/VMM preserves the directory location where tags are stored for a given thread/VCPU either into the process/VMCS structure or maintain a table containing directory location for a given thread/VCPU.

Subsequently, when the OS/VMM re-schedules the thread at a later time, a message is sent from the new destination CPU to the appropriate directory to request the tags for the corresponding thread/VCPU. The directory, upon receiving such a message, first validates the existence of a matching record in the tag-table. If a valid 2-tuple exists, it retrieves the tags from the appropriate local memory and then starts to send out messages containing up to 16 tags to the requesting CPU until all tags are sent. The requesting cache, upon receiving the messages, puts the tags into its prefetcher queue and the prefetcher then issues prefetch requests to the memory system as quickly as possible — one request per cycle in our experiment.

The OS is in best position to know when a process would be moved and thus, it would be best for the OS to designate a process for migration while it is still in execution and

trigger the command for the source L2 to send out the tags to the local directory either prior or immediately after the process is de-scheduled and thus maximizing the likelihood of preserving important working sets the process.

The delayed-migration prefetching can be also applied to preserve L2 footprints over context-switches since a context-switch is essentially a special case of local-delayed migration where the source and destination CPUs are the same.

Figure 5.28: Reduction in L2 Cache-Misses in PMW

Figure 5.29 Reduction of CPI in two-core migration experiment

Figure 5.30: Increase in message traffic (over all network links).

Figure 5.28 shows the reduction of total cache misses in PMW resulting from migration prefetching over the baseline system without our migration prefetch hardware support. Migration messages consist of either all valid tags from the L2 cache or a subset made of Local MRU tags from the L2 cache. Both tag selection schemes were able to eliminate a significant number of cache misses. Selecting all tags reduced a lot more cache misses for bzip2, omnetpp and sjeng compared to Local MRU. On the other hand, Local MRU eliminated almost same number of misses for hmmer and h264 benchmarks. These two benchmarks have smaller working set in PMW compared to bzip2 and likely suffered the eviction of some useful data footprints from migration prefetching. If the working set is relatively small then some of the valid tags in the L2 cache of P0 may come from other system jobs and thus would cause negative impact. As we have indicated before that we

could augment the tag bits to include process identifier info and thus include only tags that belong to the migrating process in comprising migration messages.

The performance benefit in terms of CPI reduction over the PMW is shown in Figure 5.29. The performance benefit is derived mainly as a result of the reduction in cache misses, despite all the overheads associated with migration messages and prefetches. The Local MRU tag prefetching improved CPI, ranging from 1.5% to 27%. Except for bzip2, sjeng and omnetpp, the Local MRU tag set prefetching performed as well or better compared to entire L2 cache set (valid entries only) prefetching from P0. Even for bzip2, Local MRU improved CPI by as much as 10%. We did not see much improvement for mcf given its very large cache footprint and rather unpredictable cache access patterns as mentioned in the case of live migration.

The observed performance benefits were realized despite the messaging and prefetching overhead involved in sending the tags from source CPU to destination CPU and subsequently prefetching the same tags at destination CPU upon arrival of migration messages. The overhead incurred in terms of extra messages transferred over all links in the interconnection network is shown in Figure 5.30. These messages are “extra,” compared to the base case without migration-based prefetches — i.e., messages that would not be required to satisfy the demand misses incurred during the PMW.

The higher the extra messages, the more ineffective the prefetches from migration messages are. Like the case of live migration, sending all valid L2 tags from the source CPU to the destination CPU generated a very large number of unused extra messages and likely

caused extra conflict misses in PMW, thus making it ineffective. On the other hand Local MRU tags were effective in keeping low the extra messages, indicating migration messages were consumed by the benchmark in the PMW. We believe we can further reduce the extra messages once we augment the tag bits with the process identifier and send only tags with matching identifier during migration. Also, additional reduction of message overhead is possible if we filter cache-line selection set based on coherence state of the cache-line as the data in the Table 5.2 suggests.

Figure 5.31:CPI reduction from Local MRU

Figure 5.32: CPI reduction from entire L2 cache set selection

The average PMW size may not be always 2 million instructions and will likely depend on the workload's characteristics and overall system load. Therefore, we have also evaluated the impact of the variation of PMW size on CPI reduction, and the results are shown in Figure 5.31 for local-MRU and Figure 5.32 for entire L2 cache set selection. Except for omnetpp, CPI reduction starts to level off after 1-1.5 million instructions following a migration, as indicated also by the geomean when the entire L2 cache is preserved in Figure 5.32. For local-MRU CPI reduction diminishes though slowly with higher PMW sizes as indicated by the geomean in Figure 5.31. Omnetpp, however, experienced higher CPI reduction with PMW of 1.5 million instructions after falling at 1 million instructions, indicating higher irregular access patterns from 1 million to 2 million instruction stream.

The migration-prefetching would be potentially more beneficial (1- 5%) if the PMW size remains small, within 512K-1M instructions. This is very highly likely to be the case in

VDI environments with high over-subscription rate and frequent I/O exits, causing VMM scheduler to be invoked frequently to either resort to context switches or migration.

Figure 5.33: Impact of L2 miss latency on migration prefetcher performance

We have assumed constant L2 miss latency in all our measurements so far, since there were no other workloads that were running in the system. However, L2 miss latency will not be constant in actual usage; rather, it is expected to be impacted by the overall bandwidth utilizations, and thus could be much higher than the default L2 miss latency of 150 cycles we assumed in our experiments. So, we introduced variability in L2 miss latency and measured the overall performance impact of our migration prefetching over a range of L2 miss latencies as shown in Figure 5.33. We see the trend is mostly upward in terms of extra performance benefit realized from migration prefetching when the L2 miss latency is

increased up to 225 cycles, and then mostly levels off. However, omnet and h264 continue to see additional benefit past 225 cycles. Thus, migration prefetching described in this dissertation would provide higher benefit in the presence of high bandwidth utilization.

5.6 Comparison with Hardware Prefetcher

To address the memory-bottleneck, the processor used in today's server comes with a hardware feature known as prefetching to bring the data early into cache from memory before the instruction is ready to consume the same and hence avoid the latency/stall otherwise would have incurred. The migration-induced cache misses also could be avoided or at-least minimized if the data could be prefetched early by the hardware prefetcher as was the case observed with our proposed migration prefetching earlier. We ask the question – does the migration prefetcher provides any additional performance benefit over the hardware prefetcher?. We have looked into two common hardware prefetching mechanisms namely:

- I. Stride prefetch – An address that is offset by a distance from a recently missed address is likely to be used in near future and thus brought into cache from memory early. The prefetcher works by monitoring load/store instructions that make accesses with regular strides, not necessarily consecutive cache lines [27]. This mechanism is usually deployed to eliminate compulsory and/or capacity misses and is particularly useful strided access pattern.
- II. Stream prefetching - A stream prefetcher looks for streams of sequence of consecutive cache lines accessed by the program. When such a stream is found the

processor starts prefetching the cache lines ahead of the program's accesses. The prefetcher maintains a table to keep track of multiple streams in flight [28].

We added both these hardware prefetchers at L2 cache level.

We compared the performance of our proposed migration prefetcher against both the stride and stream hardware prefetchers. Specifically, we have measured CPI in a PMW of 2 million instructions for the following 5 cases for the benchmarks listed earlier:

- i. Only Migration prefetcher is active
- ii. Only Stride prefetcher is active
- iii. Only Stream prefetcher is active
- iv. Both Migration and Stride prefetchers are active
- v. Both Migration and Stream prefetchers are active

We then compared performances of the cases ii-iv with the case when only migration prefetcher is active (i.e. i) and showed the results in Figure 5.34(a-f). With only one exception, migration prefetcher always performs significantly better than just stride prefetcher or just stream prefetcher. The only exception is with the benchmark *hammer*, which is known to have heavy streaming pattern accesses and thus stream prefetcher alone outperforms migration prefetcher alone, but only by 2.85%. However, when migration-prefetcher is combined with either stride or the stream prefetcher, we see further gains for all benchmarks except *omnetpp*, for which just the migration prefetcher provides the most benefit. Even, the *hammer* benchmark with heavy streaming accesses saw performance boost by 3.97% with combined migration and stream prefetchers.

5.34(a)

5.34(b)

5.34(c)

5.34(d)

5.34(e)

5.34(f)

Figure 5.34(a-f): Performance (CPI) comparison of migration only prefetch with/without hardware prefetchers

Hardware prefetchers are usually good at detecting regular access patterns and not so good at detecting irregular or random accesses. In contrast, migration prefetcher brings in a lot of footprints that may contain irregular accesses that are unlikely to be prefetched by the hardware prefetchers. This is clearly evident from the results of figure 5.34(a-f).

CHAPTER 6: SUMMARY AND FUTURE WORK

In this chapter, we will first summarize the thesis and then discuss the future work.

6.1 Summary

In a highly over-subscribed multi-processor environment, frequent migrations of threads or virtual CPU have high performance impact. This is particularly challenging for today's data center with increasingly high volume deployment of servers with hardware virtualization running consolidated server workloads or virtual desktop infrastructure workloads. Each migration introduces significant overhead particularly due to Translation Look aside Buffer (TLB) misses and loss of working set footprints within cache hierarchy in particular at L2.

One way to mitigate such overhead would be to affinitize a thread or virtual CPU to be executed on a specific CPU only. However, this approach will likely cause load imbalance, and would still suffer from context switch related misses. Furthermore, effectively determining and managing such constrain assignments of threads to CPUs would not be a trivial task and is unrealistic, particularly in a large data center with thousands of servers running varieties of workloads. Alternatively, if we could effectively preserve important footprints (both cache and TLB) and reload them with migration of the process, we could avoid the expensive misses.

We show that 60-100% of TLB cache misses could be avoided by preserving TLB cache entries over a context switch/migration. However, preserving the entire TLB cache

would increase storage pressure considering high over-subscription rate. So, we present two predictors to select those TLB entries that are most likely to be reused. Both predictors could reduce storage footprints by 20-80% for most workloads while keeping the false negative rate very low.

We also demonstrated the effectiveness of preserving cache - in particular L2 where working set of a process/VM resides - in reducing the migration-induced cache misses. We propose three different MRU based schemes (local, regional and global) to select cache tags to be prefetched following a migration. We found that the simple MRU (most recently used) based selection can provide similar benefit as more complex and costly (in terms of hardware implementation) global or regional MRU schemes and can provide performance benefits of 1.5% -27% reduction in CPI (cycles per instruction) following a migration. We also compared our migration prefetcher with standard hardware stream-based and stride-based prefetchers and found that our migration prefetcher performs almost always better. In some cases, combining the migration prefetcher with a standard hardware prefetcher can provide significant further improvements.

6.2 Future Work

6.2.1 Reduce message traffic

While migration prefetching did improve performance, we also experienced high increase in message traffic in some cases. The extra message traffic may have undesirable power impact. We could further prune our cache line selection set by including process

identifier and cache-line state in the selection criteria. This can be easily accomplished and would be interesting to see effects on overall message traffic and on performance.

6.2.2 Enhancing the OS/VMM scheduler

The operating system or hypervisor scheduler could be enhanced to manage co-operative migration prefetching. The scheduler could use tag-table entries as an additional heuristic to timely schedule processes, particularly the ones in tag-table to the most appropriate CPUs.

6.2.3 Extend migration prefetching to VM motion between servers

Currently, VM motion (i.e. moving a VM from one physical server to another) takes a checkpoint of the entire CPU state and entire memory footprints at the source Server and then transfers the checkpoint to the destination server and resumes execution at the destination. Virtual CPUs of the VM experience a lot of misses, particularly TLB and cache misses, due to warm up. We could improve the warm up by sending cache tags and issuing prefetching of the same.

6.2.4 Extend migration prefetching to fault-tolerant computing

To make a VM fault-tolerant, a second VM with exact configuration (hardware, software, memory mappings of pages) is maintained. A checkpoint of the source VM is taken at regular interval and sent to the second VM. The second VM is refreshed with source

checkpoints accordingly. When the second VM misses heartbeats from the first VM, it assumes the first VM has failed and resumes execution normally (without check-pointing). Since, we have the exact memory mappings between the VMs, the TLB entries and cache tags can also be sent with checkpoints to accelerate execution at the second VM.

REFERENCES

- [1] Login Consultants. Login VSI. Available: <http://www.loginvsi.com/>
- [2] Standard Performance Evaluation Corporation. SPEC CPU2006. Available: <http://www.spec.org/cpu2006>
- [3] C. Li, C. Ding, and K. Shen, Quantifying the cost of context switch, Workshop on Experimental Computer Science, Jun. 2007.
- [4] F. M. David, J. C. Carlyle, and R. H. Campbell, Context switch overheads for Linux on ARM platforms, Workshop on Experimental Computer Science, 2007.
- [5] J. C. Mogul and A. Borg, The effect of context switches on cache performance, Intl. Symp. on Architectural Support for Programming Languages and Operating Systems, pp. 75–84, Apr. 1991.
- [6] F. Liu, F. Guo, Y. Solihin, S. Kim, and A. Eker, Characterizing and modeling the behavior of context switch misses, Intl. Conf. on Parallel Architectures and Compilation Techniques, pp. 91– 101, Oct. 2008.
- [7] J. P. Bradford, H. F. Kossman, and T. J. Mullins, Context switch data prefetching in multithreaded computer, US Patent No. 7,493,621 B2, Feb. 2009.
- [8] J.P. Casazza, M. Greenfield, and K. Shi, Redefining Server Performance Characterization for Virtualization Benchmarking, <http://www.intel.com/technology/-itj/2006/v-10i3/7-benchmarking/6-vconsolidate.htm>
- [9] http://www.intel.com/it/pdf/Evaluating_Two-and_Four_Socket_Platforms.pdf
- [10] VMware.Server consolidation. Available: <http://www.vmware.com/solutions/-consolidation/consolidate.html>
- [11] http://en.wikipedia.org/wiki/Page_table
- [12] http://en.wikipedia.org/wiki/Extended_Page_Table
- [13] P. Jaaskelainen, P. Kellomaki, J. Takala, H. Kultala, and M. Lepisto, Reducing context switch overhead with compiler-assisted threading, Intl. Conf. on Embedded and Ubiquitous Computing, Dec. 2008
- [14] D. Ditzel. (1992) SPARCTM version 9: A robust 64-bit RISC. [Online]. Available: <https://labs.oracle.com/features/tenyears/volcd/papers/-1Ditzel.pdf>

- [15] M.-S. Lee, Y.-J. Kang, J.-W. Lee, and S.-R. Maeng, OPTS: Increasing branch prediction accuracy under context switch, *Microprocessors and Microsystems*, vol. 26, no. 6, pp. 291–300, Aug. 2002.
- [16] S. Parischa and A. Veidenbaum, Novel techniques to improve branch prediction accuracy for embedded processors in the presence of context switches, Center for Embedded Computer Systems, University of California, Irvine, CECS Technical Report 03-24, Aug. 2003.
- [17] A. S. Dhodapkar and J. E. Smith, Saving and restoring implementation contexts with co-designed virtual machines, *Workshop on Complexity-Effective Design*, Jun. 2001.
- [18] H. Cui and S. Sair. Extending data prefetching to cope with context switch misses, **Int. Conf. on Computer Design**, pp. 260-267, 2009.
- [19] A. Sarkar, F. Mueller, H. Ramaprasad, and S. Mohan. 2009. Push-assisted migration of real-time tasks in multi-core processors. *SIGPLAN Not.* 44, 7 (June 2009), 80-89
- [20] T. Constantinou , Y. Sazeides , P. Michaud , D. Fetis , A. Seznec, Performance implications of single thread migration on a chip multi-core, *ACM SIGARCH Computer Architecture News*, v.33 n.4, November 2005
- [21] J. A. Brown, L. Porter, and D. M. Tullsen, Fast thread migration via cache working set prediction, *Intl. Symp. on High Performance Computer Architecture*, Feb. 2011
- [22] S. Reza and G. T. Byrd, Reducing Migration-induced Cache Misses, *Intl. Parallel and Distributed Processing Symp. Workshops & PhD Forum*, pp.1732-1741, 2012
- [23] P. S. Magnusson, et al., Simics: A full system simulation platform, *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.
- [24] M. M. K. Martin, et al., Multifacet’s General Execution-driven Multiprocessor Simulator (GEMS) toolkit, *SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005
- [25] INTEL. Pin User Manual. <http://rogue.colorado.edu/Pin>
- [26] Intel Processor Info. <http://www.cpu-world.com/sspec/>
- [27] S. P. Vanderwiel , David J. Lilja, Data prefetch mechanisms, *ACM Computing Surveys (CSUR)*, v.32 n.2, p.174-199, June 2000

- [28] S. Srinath, O. Mutlu, H. Kim, Y. N. Patt, Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers, Intl. Symp. on High Performance Computer Architecture, p.63-74, 2007