MEMORY MANAGEMENT FOR EMERGING MEMORY TECHNOLOGIES

A Dissertation

by

VIACHESLAV FEDOROV

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

DOCTOR OF PHILOSOPHY

Chair of Committee,	A. L. Narasimha Reddy
Committee Members,	Paul V. Gratz
	James Caverlee
	Jean-Francois Chamberland
Head of Department,	Miroslav M. Begovic

August 2016

Major Subject: Computer Engineering

ABSTRACT

The Memory Wall, or the gap between CPU speed and main memory latency, is ever increasing. The latency of Dynamic Random-Access Memory (DRAM) is now of the order of hundreds of CPU cycles. Additionally, the DRAM main memory is experiencing power, performance and capacity constraints that limit process technology scaling. On the other hand, the workloads running on such systems are themselves changing due to virtualization and cloud computing demanding more performance of the data centers. Not only do these workloads have larger working set sizes, but they are also changing the way memory gets used, resulting in higher sharing and increased bandwidth demands. New Non-Volatile Memory technologies (NVM) are emerging as an answer to the current main memory issues.

This thesis looks at memory management issues as the emerging memory technologies get integrated into the memory hierarchy. We consider the problems at various levels in the memory hierarchy, including sharing of CPU LLC, traffic management to future non-volatile memories behind the LLC, and extending main memory through the employment of NVM.

The first solution we propose is "Adaptive Replacement and Insertion" (ARI), an adaptive approach to last-level CPU cache management, optimizing the cache miss rate and writeback rate simultaneously. Our specific focus is to reduce writebacks as much as possible while maintaining or improving miss rate relative to conventional LRU replacement policy, with minimal hardware overhead. ARI reduces writebacks on benchmarks from SPEC2006 suite on average by 32.9% while also decreasing misses on average by 4.7%. In a PCM based memory system, this decreases energy consumption by 23% compared to LRU and provides a 49% lifetime improvement

beyond what is possible with randomized wear-leveling.

Our second proposal is "Variable-Timeslice Thread Scheduling" (VATS), an OS kernel-level approach to CPU cache sharing. With modern, large, last-level caches (LLC), the time to fill the LLC is greater than the OS scheduling window. As a result, when a thread aggressively thrashes the LLC by replacing much of the data in it, another thread may not be able to recover its working set before being rescheduled. We isolate the threads in time by increasing their allotted time quanta, and allowing larger periods of time between interfering threads. Our approach, compared to conventional scheduling, mitigates up to 100% of the performance loss caused by CPU LLC interference. The system throughput is boosted by up to 15%.

As an unconventional approach to utilizing emerging memory technologies, we present a Ternary Content-Addressable Memory (TCAM) design with Flash transistors. TCAM is successfully used in network routing but can also be utilized in the OS Virtual Memory applications. Based on our layout and circuit simulation experiments, we conclude that our FTCAM block achieves an area improvement of 7.9× and a power improvement of 1.64× compared to a CMOS approach.

In order to lower the cost of Main Memory in systems with huge memory demand, it is becoming practical to extend the DRAM in the system with the less-expensive NVMe Flash, for a much lower system cost. However, given the relatively high Flash devices access latency, naively using them as main memory leads to serious performance degradation. We propose OSVPP, a software-only, OS swap-based page prefetching scheme for managing such hybrid DRAM + NVM systems. We show that it is possible to gain about 50% of the lost performance due to swapping into the NVM and thus enable the utilization of such hybrid systems for memory-hungry applications, lowering the memory cost while keeping the performance comparable to the DRAM-only system.

To my dearest mother, father, and sister.

ACKNOWLEDGEMENTS

I sincerely and wholeheartedly thank Dr. Narasimha Reddy for his tremendous support, guidance and patience during my years at Texas A&M. I admire his knowledge and, more importantly, the ability to think outside the box, easily combining concepts from multiple disciplines to produce elegant solutions to tricky problems.

I thank Dr. Paul Gratz for his invaluable help and mentorship throughout the work.

I appreciate my committee members, Dr. Jean-Francois Chamberland and Dr. James Caverlee, for their valuable questions, comments, and hints that helped improve this work.

I also thank Dr. Sunil Khatri for his friendship, relentless energy and enthusiasm, as well as great academic and life advice.

Finally, I wholeheartedly thank my parents and my sister for their support throughout this journey, for always being there, encouraging me through the toughest times and celebrating with me during the times of triumph.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

The Memory Wall, or the gap between CPU speed and main memory latency, is ever increasing. The latency of DRAM is now on the order of hundreds of CPU cycles. Furthermore, traditional DRAM-based main memory is now hitting hard power, performance and capacity constraints that will limit process technology scaling [28]. Typically, this gap has been mitigated by the use of cache hierarchies in the CPU which would hide the high latency for the localized, frequently-used application data sets. Increasing data sets and higher number of applications running on a processor due to virtualization are making caches less effective [17]. Alternate memory technologies, are emerging that promise to alleviate main memory capacity and scalability concerns [19, 59]. These emerging NVM technologies drive changes to the memory/storage organization that force us to revisit the common assumptions that have been relied upon in the modern systems.

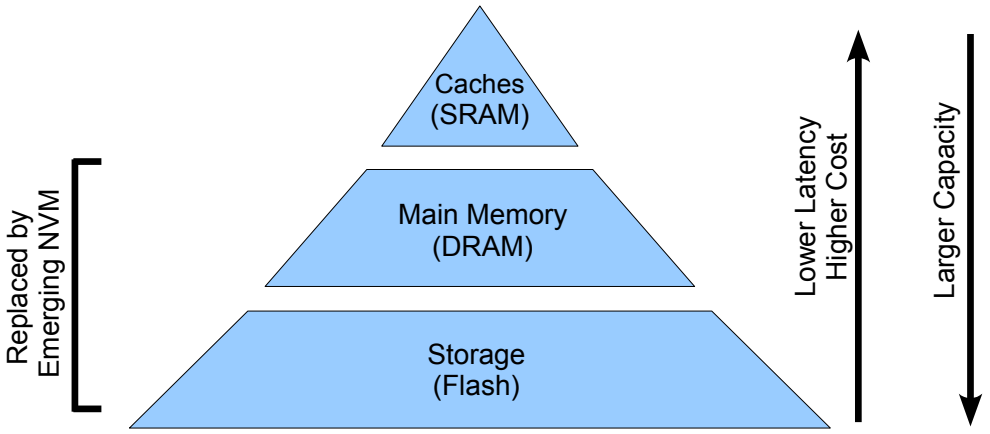


Figure 1.1: Typical memory hierarchy in computer systems, and the role of the emerging NVM technology.

Historically, there have been several types of memory available, that fit specific niches in the memory hierarchy (refer to Figure 1.1). Flash, a non-volatile, tightly-integrated and volume-produced memory, has been used in data storage. Volatile memory, typically represented by SRAM and DRAM technologies, has been used for enabling faster, random access to the frequently used data. DRAM memories, due to being compact, efficient in production, and lower-power than SRAM, have been ideal for utilization in Main Memory application. SRAM cells, because of their high speed and relatively high costs and power consumption, have been typically utilized in CPU caches which are orders of magnitude smaller than the Main Memory and where low access latency is critical.

The distinction between storage and memory is getting erased with the drive to operate on large data sets in modern applications, which in turn exerts greater pressure on the memory system, both in terms of performance and capacity. At the same time, power and energy constraints in current process technologies are imposing new limits on off-chip bandwidth. Traditional DRAM-based main memory is now hitting hard power, performance and capacity constraints that limit process technology scaling [28].

NVM (typically represented by Flash memory) used to be much slower than DRAM and thus only suitable for longer-term storage (such as a replacement for the spinning disks). On the other hand, the modern emerging NVM technologies such as PCM, MRAM, Memristors, etc. [59] outperform Flash in both access latency and the rewrite performance (they do not need to erase the whole block of data as is done in Flash, thus they present byte-granularity similar to DRAM). These emerging technologies are approaching the performance of DRAM, with larger capacity, higher scalability, and non-volatility, making them the ideal candidates for the novel class of Storage-Class Memory [19].

As compared to the conventional memory hierarchy, the emerging NVM has several different key characteristics which challenge many of the traditional assumptions about the design of memory architectures on both the hardware and the software level. The characteristics include:

- Low latency random access capability compared to spinning disks.
- Relatively low endurance compared to DRAM.
- High write energy and latency compared to DRAM.

In this work, we address the issues related to the adoption of NVM on both the architecture and the software levels. We look into the roles of NVM for complete DRAM replacement as well for DRAM extension.

In Chapter 2 we address the challenges related to utilization of NVM for replacing DRAM as Main Memory. In particular, the limited endurance and high write latencies of NVM need to be exposed to the CPU caches for optimal performance and maximum NVM lifetime. It is vital to control the amount of writebacks from the Last-Level Cache (LLC) to Main Memory. We propose an adaptive LLC management policy that modifies replacement *as well as* insertion policies in the cache, in response to the changing program behavior, so as to optimize the writebacks to NVM main memory and at the same time improve the cache miss-rate.

In Chapter 3 we notice that the growing working sets of modern applications and the increased degree of memory sharing between CPU cores, exert increasing pressure on the cache system. Combined with the conventional decade-old assumptions in the Operating System about the typical cache sizes and application working sets, this leads to suboptimal performance with increased amount of unnecessary cache misses and writebacks. These OS effects on the caches have not been considered in the

cache management literature previously. We analyze the degree of the OS influence on caching performance and propose a software-only, adaptive approach to mitigating such influence with minimal overheads.

For large machines in the data centers the capacity and latency of pure NVM as main memory might prove insufficient. An alternative approach to utilizing the NVM presented in the literature is to expose both the devices on the memory bus. In such systems, not only does the memory controller hardware need to support both the device types, but it also has to be modified in order to adaptively move the data between the two types of memory based on the usage patterns, i.e., the frequently-updated data would be placed into the faster DRAM, while the more static data can reside in NVM. In Chapter 5 we explore an alternative approach. We propose the use of PCIe-attached NVM [91] (based on the NVMe standard) for transparent DRAM extension as a software-only approach, using the OS swapping subsystem as a base. We show that the OS swapping support of NVMe devices is inefficient since it is based on the old assumptions about the high-latency, sequential-access spinning disks. In particular, the OS fetches extra pages of data beyond what the application demands, while the application is waiting. While the overhead of doing such work is effectively hidden with higher-latency disks, using NVMe memory exposes such bottlenecks in the OS design. In our work, we address the issues with OS swapping, and build a framework for managing the DRAM+NVMe hybrid system by adopting a predictive page fetching algorithm, providing an impression of large memory capacity with the effective latency of DRAM.

As an unconventional approach to utilizing emerging memory technologies, in Chapter 4 we present a Ternary Content-Addressable Memory (TCAM) design with Flash transistors. Such design could be utilized in Virtual Memory accelerator applications, allowing faster operation when employing NVM for data storage. Instead

of increasing the TLB sizes, which would induce additional delay on the critical path of CPU reads, for instance, TCAM could be leveraged for storage of the Page Table and hence fast Virtual-to-Physical address translations, obviating the need of the costly Page Table walks on TLB misses.

2. ARI: ADAPTIVE LLC-MEMORY TRAFFIC MANAGEMENT *

2.1 Introduction

The working sets of modern applications keep increasing, which in turn is placing greater pressure on the memory system, both in terms of performance and capacity. At the same time, power and energy constraints in current process technologies are imposing new limits on off-chip bandwidth. Furthermore, traditional DRAM-based main memory is now hitting hard power, performance and capacity constraints that will limit process technology scaling [28]. Alternate memory technologies, such as Phase-Change Memory (PCM), have been proposed to alleviate main memory capacity and scalability concerns, however these technologies impose further costs on off-chip writes in terms of power consumption and wear-out. In this chapter we present an adaptive approach, leveraging program phase behavior, to address all of these challenges. In particular, we propose to adaptively modify Last-level Cache (LLC) management policy to simultaneously reduce writebacks while improving miss-rate, addressing both power and performance concerns. Although there has been considerable prior work exploring LLC management policies to reduce miss-rate and improve performance and some prior work examining policies to reduce writebacks for optimized bandwidth consumption, we present the first work to our knowledge to simultaneously address both.

To mitigate the greater memory system pressure placed by applications on their memory systems to maintain data and instruction stream needs, current chips employ memory system hierarchies with several levels of cache prior to main memory

*Viacheslav V. Fedorov, Sheng Qiu, A. L. Narasimha Reddy, and Paul V. Gratz. 2013. ARI: Adaptive LLC-memory traffic management. ACM Trans. Archit. Code Optim. 10, 4, Article 46 (December 2013). DOI=http://dx.doi.org/10.1145/2543697

[63]. Last-level Caches (LLCs) are optimized towards capacity rather than speed, and are often highly associative. LLCs typically employ Least Recently Used (LRU) or approximations of LRU policies to choose which victim block is to be replaced when a cache miss occurs. If the victim block is dirty, it must be written to memory before the new incoming block may be written to cache. These writebacks constitute an increase in off-chip bandwidth consumption, particularly for cache-lines that ping-pong back and forth between the LLC and main memory. Furthermore, for alternate memory technologies such as PCM, this writeback cost is particularly high and undesirable.

Considering the high costs of writebacks, it is better to replace a clean cache block in the LRU stack rather than the dirty LRU one, however, the replaced clean cache block should not generate more misses otherwise overall system performance might suffer. We observed that the number of hits often distributes unevenly among cache ways of the LRU stack in the LLC (see Figure 2.1). Most of hits accumulate on the first few MRU ways, with a much lower, flat distribution among other parts of the LRU stack. Based on this observation, we can avoid generating writeback traffic by replacing those clean blocks in the LRU stack which have less (or comparable) hits relative to the dirty LRU ones without introducing additional misses.

In exploring LLC hit distribution across applications, we find they vary greatly from application to application and even from program phase to program phase within a given application. A one-size-fits-all approach to LLC management either imposes a significant cost on miss-rate for some applications or does not reduce writebacks sufficiently. In this work we propose an LLC management policy which adapts to program phases such that both writebacks and miss-rate are improved. The primary contributions of this work are as follows:

- An adaptive cache management scheme, ARI, that simultaneously reduces *both* the miss-rate and writeback rate compared to LRU, to accommodate the behavior of different applications and different phases of an application.

- Our design reduces main memory energy consumption, while increasing IPC by 4.9% on average over traditional LLC cache design.

- When used in conjunction with PCM-based main memory, ARI improves system lifetime on average by 49% beyond a randomized wear-leveling baseline [73, 86].

The remainder of the chapter is organized as follows: In Sections 2.2 and 2.3 we provide design and implementation details on ARI which is then evaluated in Section 2.4. We present further analysis on our design in Section 2.5. We then discuss prior work in cache replacement and PCM-based main memory in Section 2.6. Section 2.7 concludes the chapter.

2.2 Design

Our **goal** is to reduce the writebacks as much as possible, while simultaneously keeping the miss rate at least equal or better than conventional LRU policy, to avoid the performance loss. Since our scheme has comparable or better miss rate than LRU as well as writeback reduction, it produces a significant main memory bandwidth reduction and improves the system performance. Therefore, the proposed scheme, Adaptive Replacement and Insertion (ARI) is potentially useful for a broad range of future memory systems, including DRAM-only systems, and hybrid memory systems employing both PCM and DRAM.

In this section, we discuss static replacement policies which favor writeback reduction, followed by our adaptive replacement and adaptive insertion techniques which

together form the proposed ARI LLC cache management policy.

As the cache hit distribution across ways in the LRU stack is nearly bimodal, we mentally divide the LRU stack into two partitions. We call these partitions the "High-hit" and "Low-hit" partitions, reflecting the number of hits in cache ways belonging to a given partition. To distinguish between different applications' varying Stack Distances, we refer to the partitioning as nHm where n is the number of cache ways belonging to the High-hit partition and m is the cache associativity (i.e., Figure 2.1 gives an example of a 4H16 partitioning: *mcf* benchmark has the majority of its cache hits in the four MRU ways, whereas twelve other ways receive a relatively low number of hits).

The items in the High-hit partition should obviously stay in the cache as long as possible since they are being frequently accessed. On the other hand, the items in Low-hit partition are accessed much less frequently and, importantly, *at nearly the same rate*, thus evicting a line in any position in the Low-hit partition has approximately the same effect on miss rate as any other. Therefore, evicting any clean block within the Low-hit partition is more beneficial than evicting a dirty LRU block (reducing writebacks). Thus upon a cache miss, we use a policy within the Low-hit partition which first tries to find a least recently used clean block. As an example, Figure 2.2 depicts potential eviction candidates within one cache set, for various hit distributions within the set (cf. Figure 2.1). In the case there are no clean blocks, the least recently used dirty block is replaced. If there are no Low-hit blocks (i.e., 16H16), the block in LRU position is evicted. By doing this, we greatly reduce the writeback traffic to memory without degrading overall performance of the system.

In our initial experiments we found that any given static policy, while decreasing writeback rate, will typically sacrifice miss rate relative to LRU on average across

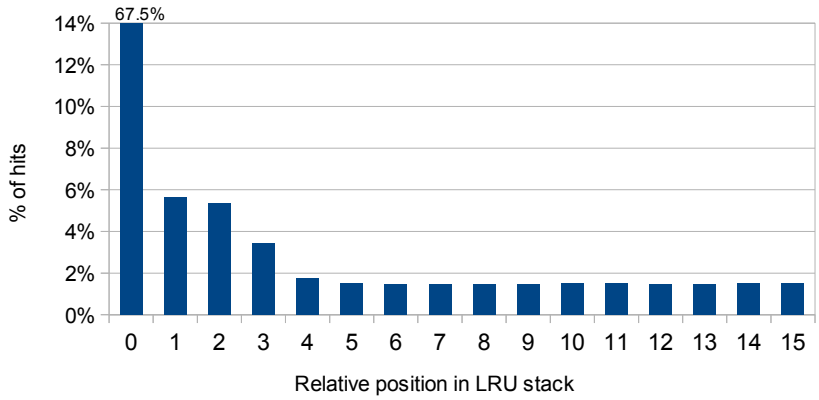


Figure 2.1: Distribution of total number of hits across a 16-way, 2MB LLC for *mcf* application (0 - MRU, 15 - LRU).

a suite of benchmarks. This is because it is difficult, if not impossible, to statically tune a single one-size-fits-all policy since different applications exhibit varying Stack Distance distribution. Furthermore, applications have execution phases where the cache accesses follow different patterns, hence the Stack Distance distribution and optimal cache partitioning varies by phase. As a result, in some applications the miss rate increases by up to 3× relative to LRU, while the writeback count is only marginally decreased when a static policy is used.

2.2.1 Adaptive Replacement

We propose to adapt to application behavior, as well as to the execution phases inside applications by extracting run-time information about the program execution from the cache itself. In particular, we propose sampling a small number of cache sets to estimate the current application behavior under different partitionings of the cache and choose the best cache partitioning according to that data, balancing miss rate against writeback benefit.

While it is possible to extract the stack distance distribution directly from cache

and use this information to adapt the cache policy, the stack distance is a function of the replacement policy and is therefore effected by the policy currently in use. Instead, to account for the impact of the policy on cache metrics, we plan to adopt a direct measurement approach, implementing various sample partition schemes on selected sets, using shadow tags, rather than trying to measure the stack distance of the cache as a whole to guide the policy.

Our approach employs p static sample partition policies, where each policy partitions the ways in a set at a particular stack distance. Among our sampled policies, we always include two extremes, one being LRU and the other 0H16. The other policies partition their sets at different locations. To improve the accuracy of the modeling, we implement the sample policies in shadow tags, i.e. for a sampled cache set, we have additional p tag sets which implement the p static policies.

For each sampled policy i, where $i = 1...p$, we maintain two performance counters. The first counter measures the number of misses, the second counter measures the number of writebacks. These performance counters are used to compare the different sample policies at the end of an epoch. Based on the comparison of the performance counters, a policy for the entire cache is chosen for the next epoch. Rather then resetting the counters to zero at the end of each epoch, we adopt a decaying average similar to that used in RTT calculation in TCP/IP protocol.

The cache controller maintains an epoch countdown timer which counts down the epoch length before triggering the decision logic. Once the epoch has ended, the miss counts, mr(i), and writeback counts, wb(i), are summed together for each of the sampled sets, and the policy(i) with the lowest sum is chosen.

The intuition for considering the sum of mr(i) and wb(i) is that it provides a rough guideline for the savings in memory bandwidth relative to LRU, if this policy is adopted. If a policy provides a pronounced decrease in writeback rate, it may be

tolerable to allow some increase in miss rate, still keeping the memory traffic lower than LRU, and vice versa. If all the sampled policies perform the same as LRU (or there are no accesses to the sampled sets), the main cache policy is not changed. After the decision hardware has picked the best policy for the current epoch, the cache controller applies it toward the whole cache.

We also tried varying the weights of mr(i) and wb(i) in the sum, to see if this could lead to a better trade-off. Our results indicate that writebacks can be further reduced, by up to 2%, when misses are not considered. Similarly, misses can be reduced by up to 2% if writebacks are ignored. The equal weights used in our work yield the best writeback improvement without negatively affecting performance.

We note that the power of sampling lies in its simplicity. More intelligent schemes might yield a better low- vs. high-hit partitioning, but would require more information than what can be extracted just from the LLC. IPC counts or an address of current load/store instruction are the examples. LLC is typically very distant from the main core, so providing such additional information to the cache controller would not only complicate the controller, but also disturb the core (i.e., stronger drivers are needed to support long wires, etc.).

2.2.2 Dynamic Insertion

In conventional replacement policies, when a block is brought into the cache, it is typically inserted as an MRU element in the stack; however, prior work has shown that this is not optimal for miss rate in the presence of reference streams with varying amounts of locality [72]. Earlier work used set dueling to choose the insertion point between the MRU or LRU position, depending on the expected locality of the inserted cache line. We also propose to adopt a dynamic insertion policy, however, we leverage our characterization of the LRU stack into low-hit and high-hit partitions to better

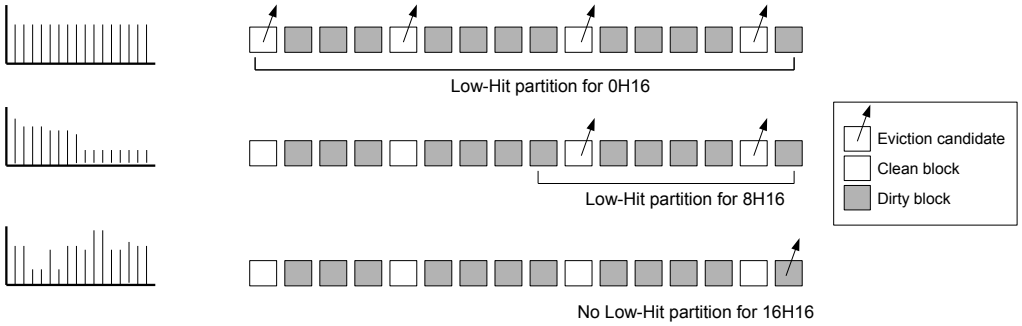


Figure 2.2: Eviction candidates for various static policies.

place low locality cache blocks (effectively yielding $p + 1$ insertion locations instead of just 2 as we will show). Set dueling utilizes a single bimodal misses counter, and picks one of the two policies to use based on that counter. If the number of misses for both policies is similar, the counter might get stuck, or it might oscillate around the threshold, switching the policies even if it actually were beneficial to keep the certain policy. ARI uses miss- and writeback counts in making decisions for a number of policies for every epoch, which allows a faster and more precise adaptation. The phases in set-dueling are those when the miss rate of one policy gets sufficiently better/worse than the other as to bias the counter and trigger the policy change (i.e. if one policy initially experiences a lot of misses and then performs better compared to the other policy, it may not be chosen until the bimodal counter gets to the threshold). Phases in our work are when the application's high-hit and low-hit way distribution changes, so the appropriate policy is chosen as soon as possible.

We propose inserting clean data with expected low locality at the top of the low-hit partition, as illustrated in Figure 2.2, under the expectation that this will yield a lower miss rate than LRU insertion in the event our speculation is wrong. Clean data with expected high locality is always inserted in the MRU location. The intuition is, if data has low locality and we correctly insert it in the low-hit partition, we are

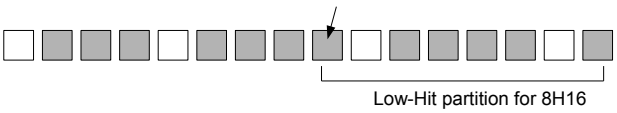


Figure 2.3: Insertion point for 8H16 static policy, for low-locality blocks.

likely to see a reduction in writebacks and misses as we don't disturb the elements belonging to the high-hit partition that are receiving significantly more hits.

Again, we take a measurement approach to decide which insertion location reduces writebacks and misses best for the current phase of the running application. For each partitioning policy, we consider insertion at both MRU and the top of the assumed low-hit partition, and choose the insertion policy for the next epoch based on the observed behavior in the current epoch. As a result, a direct implementation of both the replacement and insertion policies, we need to double the number of shadow tags and associated counters. We will show later that this overhead can be reduced to a smaller number of tags while keeping the performance nearly the same as a full-scale implementation. We note that ARI may be used to control non-stack-based cache management policies, provided that the sample sets employ LRU to estimate the high- vs. low-hit distribution in the LLC.

2.3 Implementation Details

The proposed scheme is relatively easy to implement; it requires little additional hardware, and the decision-making delay is insignificant relative to our epoch length.

Figure 2.4 shows the baseline 8-node CMP with three levels of cache hierarchy, and Figure 2.5 depicts the ARI design implemented in the CMP L3 cache banks. L1 and L2 caches are assumed private to each core, with each node having a single bank of shared L3 cache. Our scheme assumes that the LLC is non-inclusive. Although

Table 2.1: Baseline cache configurations.

System	Single core	System	Multicore
L1 cache	32KB L1I + 64KB L1D, 2-way, LRU, 64B block	**L1 cache (Private)**	64KB L1I + 64KB L1D, 2-way, LRU, 64B block
L2 cache	256KB, 8-way, LRU, 64B block	**L2 cache (Private)**	256KB, 8-way, LRU, 64B block
L3 cache	2MB, 16-way, LRU, 64B block	**L3 cache (Shared)**	16MB, 16-way, LRU, 64B block
Main memory	4GB, DDR3-1333 DRAM, 32-entry write buffer	**Main memory**	4GB, DDR3-1333 DRAM, 32-entry write buffer

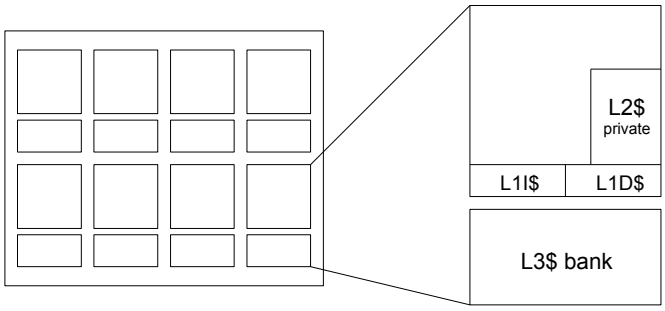


Figure 2.4: Baseline CMP.

inclusivity simplifies cache coherency, prior work has explored cache coherency in non-inclusive caches [107, 30, 22].

We randomly pick 8 of each L3 cache bank's sets to shadow. For each selected set we create $p \times 2$ (p partition policies times two insertion policies) shadow tag sets to be sampled. Each of the p shadow tags for a given set implements one of the sample partitioning policies. These sample sets are doubled to include one sample set for each insertion policy: at the top of the low-hit partition (LH), and standard MRU. For example, given $p = 3$, 8 sets of sample tags will be maintained for 0H16-MRU, 0H16-LH, 8H16-MRU, 8H16-LH, 16H16-MRU (traditional LRU), and 16H16-LH per

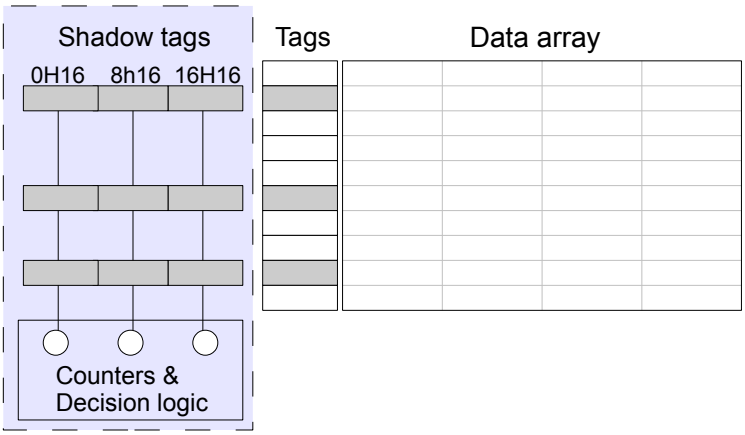


Figure 2.5: L3 bank with shadow tags array.

L3 cache bank.

Assuming a 16MB (2MB per core), 16-way L3 with 64-byte lines, there are 16384 total sets in the cache. The storage overhead for ARI is thus 1152, 640, and 384 tag sets, or 7%, 4%, and 2% of L3 tag array (14kB, 7.5kB, and 4.6kB of storage, using the hashed tags approach, as discussed later) with $p = 9$, 5, and 3, respectively. As we will show later, these overheads can be lowered without impacting the performance.

In a single-core configuration with a 2MB LLC, ARI induces 7kB of storage overhead; in comparison, sampling Dead-block predictor (DBLK) [38] has 13kB storage overhead for a 2MB LLC, and requires additional communication circuitry from CPU core to the LLC.

We note that no modification to the main cache memory structures is needed. Only the cache controller needs to be modified. The shadow sample sets are collocated with the L3 cache banks that contain the sets they shadow such that the addresses matching the shadow sets are sent to both the set itself and the shadow (Figure 2.5). Shadow tags are independent of the main cache structures and do not

affect the cache operation.

On an epoch boundary the cache controller examines the sample sets to determine the best policy. The epoch timer counts up to a maximum of 25 thousand LLC references and thus requires 15 bits. The performance counters for each sample policy can be made 15-bits wide, to ensure no counter overflows. Basing the epoch on reference count rather than cycle count allows adaptation of the epoch's resolution to the relative activity of the memory system at that time. The epoch length was determined experimentally. Once the epoch size has been set, no further tuning is necessary, as ARI dynamically adapts to the runtime phases regardless of the nature of the phases (e.g., one application using the cache in a single-core CPU, or several applications competing for multi-bank cache).

To increase the accuracy, we maintain some history in the performance counters. At the end of each epoch, we compute the performance measure as both a function of the previous history and the current sample obtained during the current epoch. This smooths the sampled data and allows better adaptation. We compute each sample according to the following formula: $new\_value = 0.9375 \times previous\_value + 0.0625 \times current\_value$. The fractions are chosen for ease of implementation.

To compute the miss and writeback sums, we propose using a low-power, pipelined adder. We expect the time to compute the sums and comparisons and change policy should be on the order of 100-200 cycles, insignificant relative to epoch size. To ensure we never use policies that increase misses excessively, we remove from consideration policies with positive miss deltas above the preset threshold. If the threshold is chosen carefully, (say as 6.25% or 1/16), then this threshold check can be performed with simple shift operations.

2.4 Evaluation

In this section we examine the performance of ARI under various workloads. We also compare ARI to the LLC management schemes from previous work.

2.4.1 Methodology

For single-core as well as single-processor multitasking simulations we use the gem5 simulator [6] paired with the DRAMSim2 main memory simulator [82], running the compiled code from SPEC CPU2006 benchmark suite utilizing the single SimPoint methodology [69]. We picked applications from SPEC 2006 package that provide good representation of the whole suite in terms of stack distance behavior and cache demands.

For multi-core and multi-threaded simulations, the gem5 simulator running the PARSEC suite is utilized. We used nVidia Tegra-like system with a dual-issue out-of-order processor, as the baseline for single-core benchmarks, and a multiprocessor system for multicore benchmarks. The cache configurations used are shown in Table 2.1. The implementation only impacts the L3 cache, the LLC of the system. We simulated three levels of caches, but all the techniques presented in this work can be applied to any last-level cache.

Table 2.2: DRAM and PCM characteristics for 1GB chip.

Power	**DRAM**	**PCM**
Row read	210 mW	78 mW
Row write	195 mW	773 mW
Activate	75 mW	25 mW
Standby	90 mW	45 mW
Refresh	4 mW	0 mW

Latency	**DRAM**	**PCM**
Initial row read	15 ns	28 ns
Row write	22 ns	150 ns
Same row read/write	15 ns	15 ns

The latency and power estimates for DRAM and PCM shown in Table 2.2 are adopted from Dhiman et al. [14].

2.4.2 Performance

In single-core configuration, ARI samples $k = 9$ distinct policies (including 0H16 and LRU) evenly distributed over the range of possible stack distances, with 32 sets for each sample policy. Figures 2.6 and 2.7 present the miss- and writeback gains for ARI as well as 4H16 static policy, DIP [72], Dead Block Prediction (DBLK) [38], and DRRIP [31]. The results are normalized to LRU.

We see that ARI performs well, achieving 33% LLC writeback improvement on average. ARI never causes misses to increase by more than 5% (milc, sjeng), and on average improves them by 4.7%. In comparison, DIP, DBLK, and RRIP decrease writebacks on average by 11%, 18%, and 13%, respectively. DIP improves misses by 6%, DBLK by 11%, and RRIP by 8%, on average. Note how for astar application, ARI identifies the fact that there is little gain to be exploited and reverts to LRU, while DBLK and RRIP increase misses by 10%. The 4H16 static policy achieves 25% writeback improvement but increases misses by 5% on average. Note that for bzip2 and astar 4H16 miss rate is 80% and 30% greater than with LRU. Our simulations show that as we go from 0H16 toward 16H16, the negative effect on misses declines, but so does the writeback improvement. ARI dynamically tunes the replacement policy to achieve significant improvement in writebacks and misses.

2.4.3 Memory Bandwidth Reduction

Decreasing the traffic from CPU LLC to main memory is important in modern systems. ARI yields the main memory bandwidth reduction of up to 39% (for h264ref), and 8.3% on average. In comparison, our simulations show that DIP, aimed at reducing LLC miss traffic, reduces the main memory bandwidth by 4.6%. RRIP

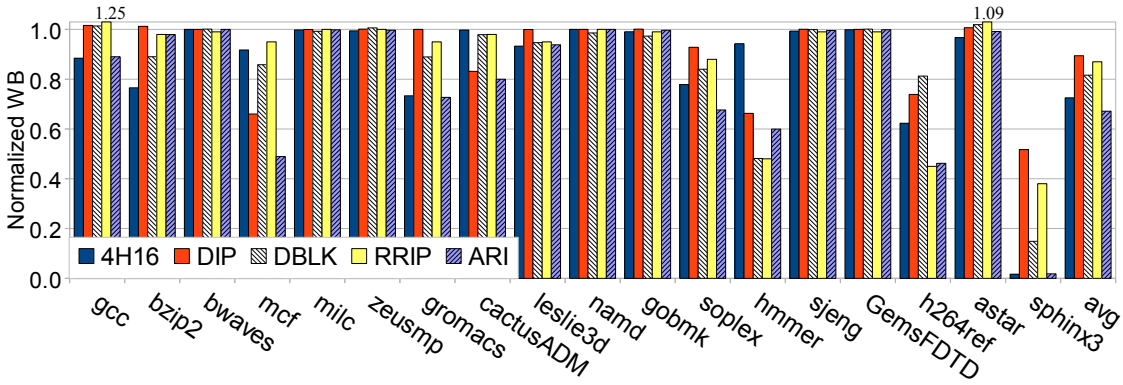


Figure 2.6: Writebacks improvement, normalized to LRU.

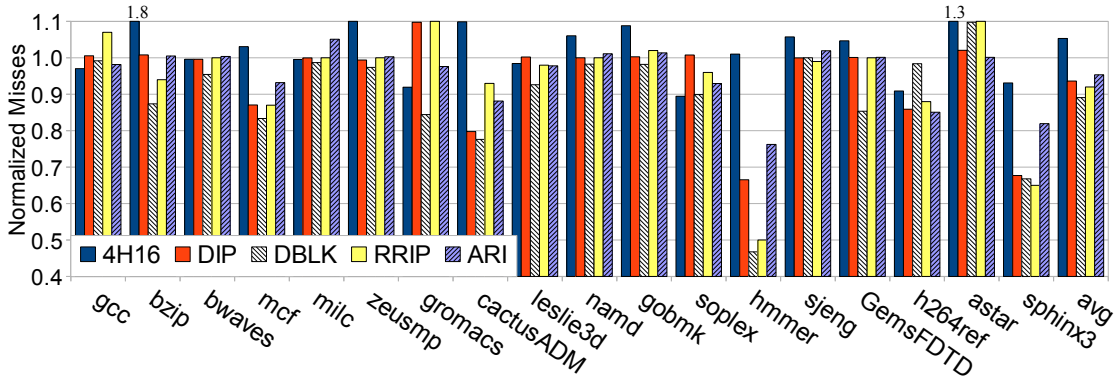


Figure 2.7: Misses improvement, normalized to LRU.

decreases the bandwidth by 7.4%. Sampling dead block prediction scheme achieves 10% bandwidth reduction, although with twice the storage overhead.

2.4.4 Application Speedup

The improvement in misses and writebacks, and thus the decreased main memory traffic, leads to IPC improvement and program speedup. Note that IPC is mostly dependent on cache misses reduction. We conclude that the writebacks are absorbed in the write buffer. They only affect the performance when write traffic congests the memory bus, or when the write buffer gets full and the CPU core is forced to wait for

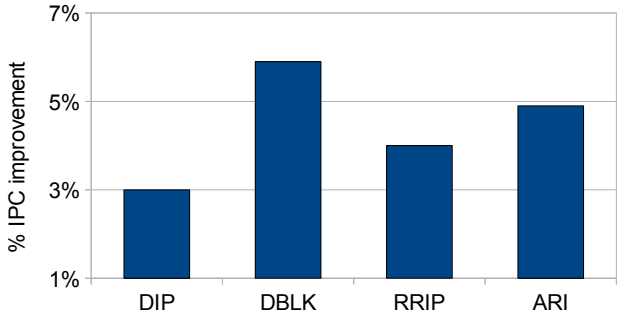


Figure 2.8: IPC improvement with ARI, DIP, and DBLK, normalized to LRU.

a write to commit. ARI decreases the number of writebacks substantially, so there is less congestion on the bus, and more opportunities to free the entries in the write buffer for the incoming blocks.

Figure 2.8 presents IPC improvement over the SPEC applications simulated, normalized to baseline LRU. ARI achieves a 4.9% speedup on average, outperforming DIP and RRIP, and nearly equivalent to DBLK.

2.4.5 Energy and Lifetime

In order to understand the impact of reduced writebacks and misses on the memory system, we simulated three different memory architectures. The first consists entirely of DRAM, the second consists entirely of PCM, and the third employs a 256MB DRAM cache [77] in front of PCM.

Note that we do not include the energy overhead introduced by ARI sampling structures in LLC, since it is negligible compared to the main memory power consumption. We estimate that ARI structures use less than 5% of the total L3 power. Taking into account that a typical LLC consumes 2.75W peak power [38], ARI adds less than 150mW of power overhead - this is half of 1GB DRAM bank read power. Furthermore, to be completely fair in reporting total system power, we would have to

analyze the reduction in memory bus power, power savings due to less cache misses and faster application runtime, etc., in addition to sampling overhead. This is a matter of a separate paper.

With DRAM-only memory, the total energy is mostly dependent upon application running time, as write energy consumption is not significant compared to standby and refresh energy. The average energy savings in a system employing ARI are thus 4.9%.

Because ARI is aimed at reducing writebacks, more prominent energy savings can be achieved when memory technologies with expensive write accesses, such as PCM, are employed. Applications such as mcf, h264ref and hmmer experience more than 10% savings. The average energy savings across the simulated applications are 8.9% compared to the baseline LRU.

As the baseline for PCM lifetime comparison, we use randomized wear leveling, where the cells receive equal number of writes, and thus lifetime is essentially the number of writes a cell can sustain until it is burned out. The simulations suggest that PCM-only system with ARI-managed LLC sees an average lifetime improvement of 49%, with some workloads attaining up to 2.5× lifetime extension. In contrast, DIP yields 12%, DBLK 23%, and RRIP a 15% PCM lifetime improvement on average. However, in a system implementing DBLK or RRIP, PCM lifetime *decreases* by a negligible amount for gcc, zeusmp and astar applications.

With DRAM cache in front of PCM, the cache absorbs all the memory read and write requests (with the exception of bwaves, GemsFDTD, and zeusmp). The energy reduction is limited to 0.96% on average, because the 256MB DRAM cache uses much less energy than 4GB DRAM memory. In low power systems, e.g. phone and tablet systems, however, DRAM caches may be too expensive.

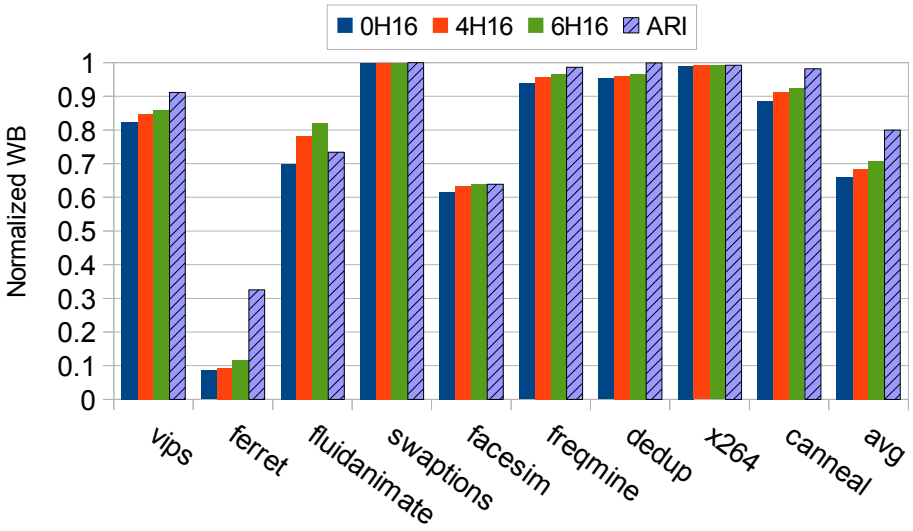


Figure 2.9: Writebacks for the PARSEC applications, normalized to baseline LRU.

2.4.6 Multiprocessors

We ran nine PARSEC applications to explore ARI's performance in a multiprocessor configuration with a 16MB LLC. The results are presented in Figures 2.9 and 2.10 below, with the rates normalized to baseline LRU. The last three bars are geometric mean over the 9 applications. Once again, we observed that ARI does not increase miss rate by more than 4% for any single application, while reducing the writeback rate by 20% on average and keeping the average miss rate same as LRU policy. In contrast, the 4H16 static policy reduces writebacks by 29% on average, but allows miss rate to increase by 26% in the worst case, and by 9% on average.

We also varied the size of the L3 cache, and found the writeback and miss rates improvement to be 27% and 0.9% on average for an 8MB cache utilizing ARI, and 9% and 3.3% on average for 32MB cache. The results for multiprocessor simulation are consistent with the single-core results (Figures 2.6, 2.7).

We expect that multi-program workloads, where distinct applications run on

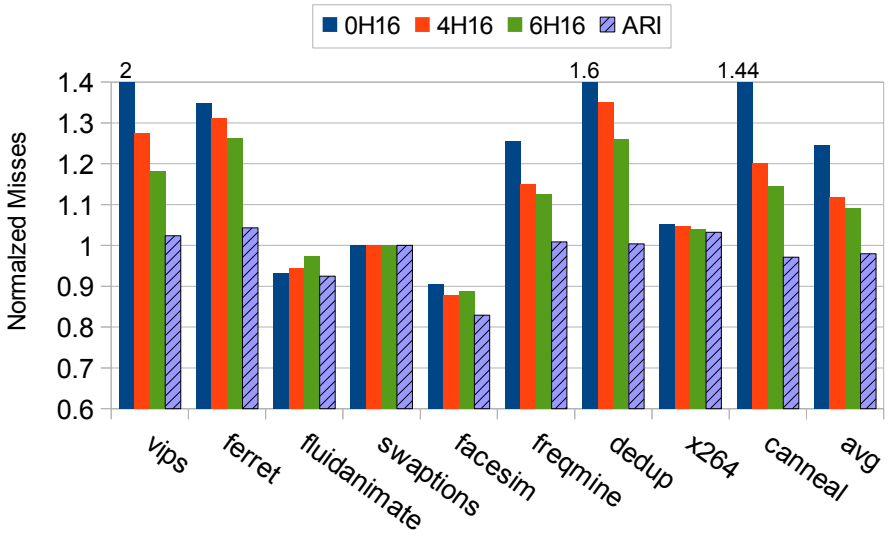


Figure 2.10: Cache misses for the PARSEC applications, normalized to baseline LRU.

different cores, will behave similar to muti-threaded workloads. ARI generally adapts to the phase behavior observed in the shared last-level cache. This should hold true if the behavior is due to a single application, or the aggregate of many applications.

2.5 Analysis

In this section we first discuss the dynamic behavior of the proposed scheme. Next, we discuss the effect of several parameters on the performance of the proposed scheme. Finally we discuss various means to reduce the hardware overhead of the technique.

2.5.1 Dynamic Behavior

Figure 2.11 shows how ARI adapts to program phases in soplex application; for each epoch, we plot the best stack distance distribution High-hit ways number (the n in our nHm notation) and the insertion policy used (the higher marks denote

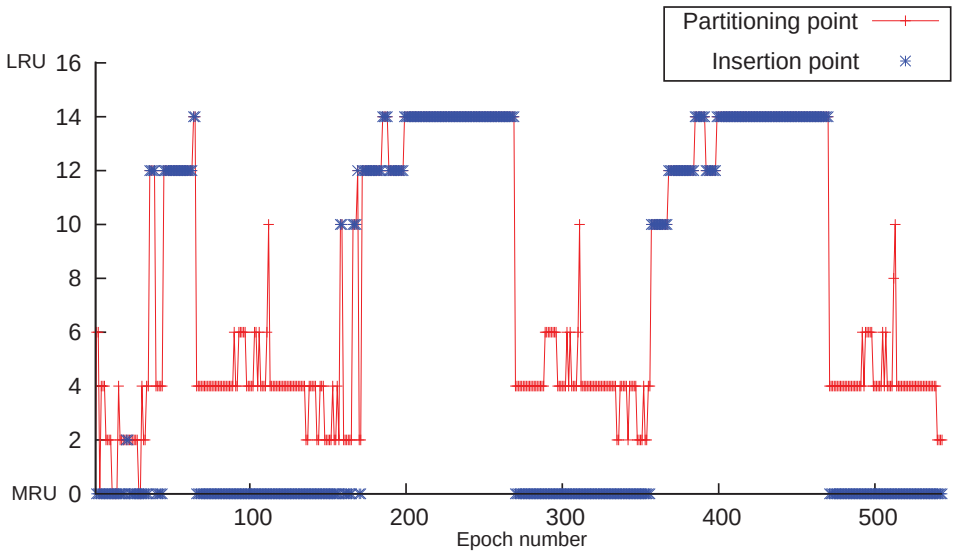


Figure 2.11: Adaptivity graph for soplex application.

Low-hit insertion is used, and lower marks - conventional MRU insertion). As the figure shows, the best policy changes frequently over the execution of a program. ARI adapts the policy based on these observed program phases.

Figure 2.12 shows the writeback rate curves for mcf application (the X axis shows the period number, where we picked the data in intervals of 1 million LLC accesses for this graph; and Y axis shows the number of writebacks for the corresponding period). We observe that the program behavior can change fairly fast from one epoch to the next epoch. We see that ARI achieves lower writeback rate than the best static policy, 0H16, because it adapts insertion *and* replacement policy to the most beneficial stack distance distribution for the current phase. Our simulations show that using dynamic insertion in ARI yields an 8% writeback reduction on average compared to the static MRU insertion.

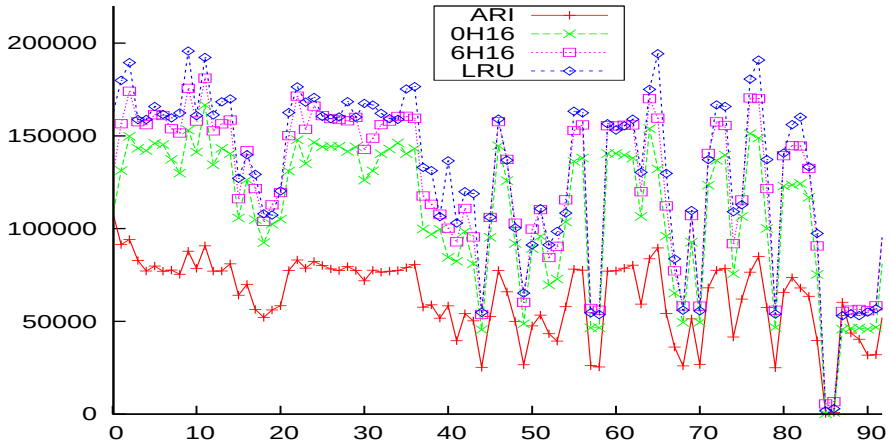


Figure 2.12: Writeback curves for mcf, ARI vs. static policies.

2.5.2 Cache Size and Set-Associativity

We varied the L3 cache size to determine the scalability of ARI. Figure 2.13 shows results for different cache sizes, in single-core configuration. The miss and writeback counts are normalized to the counts of the respective LRU-managed caches. It is clear that ARI scales well. We observed that (a) at 1MB cache size, misses are more frequent, data does not stay in the cache long enough to benefit from another write hitting in the cache before getting evicted; (b) at 8MB, it is more difficult to improve on LRU since due to the larger cache capacity, writebacks are becoming relatively infrequent; (c) ARI works the best with L3 caches 2-4MB in size, for the workloads considered in the analysis, yielding a decrease in writebacks of 33.3% on average, and a decrease in misses of 6.8% on average, as compared to baseline LRU.

We simulated 2MB LLCs with associativity from 4 up to 32 ways as shown in Figure 2.14. The number of policies per sample set is, respectively, 3, 5, 9, and 17. The larger number of ways provide more potential eviction candidates in Low-Hit ways according to the stack distance, thus more opportunities to gain from the

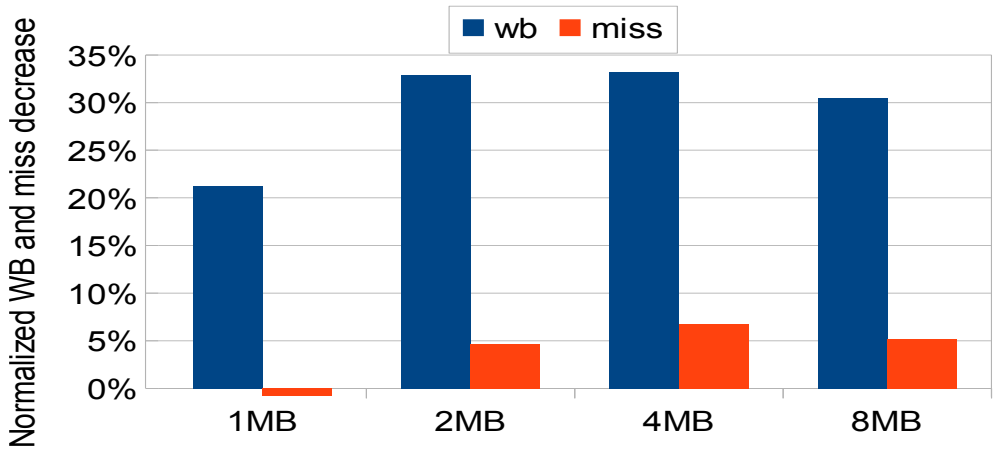


Figure 2.13: Normalized writebacks and misses decrease under varied cache sizes, SPEC2006 (higher is better).

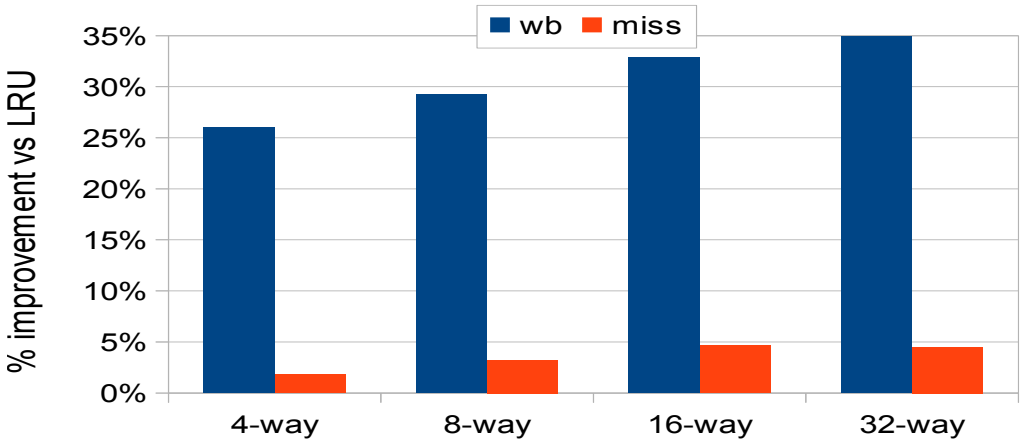


Figure 2.14: Normalized writebacks and misses decrease under varied associativity (higher is better).

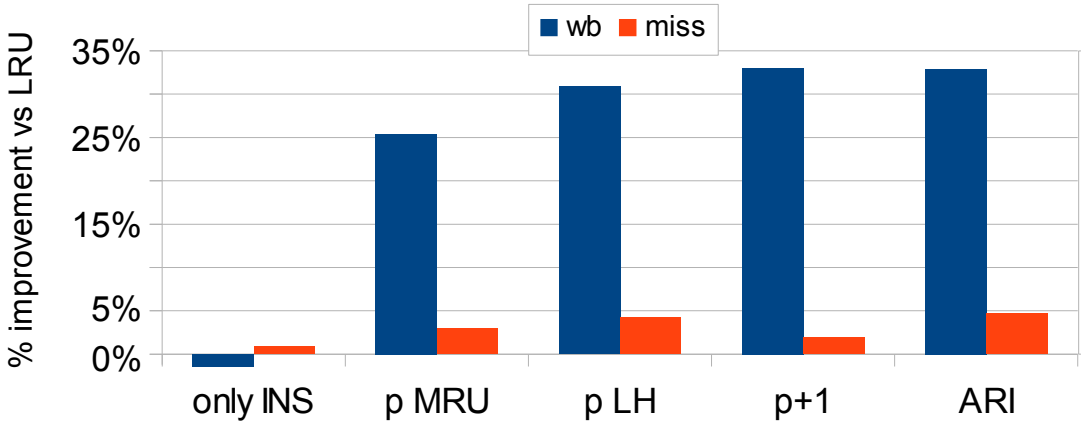


Figure 2.15: Comparison of the schemes with various insertion policies.

sampling scheme. We noticed that the improvements in writeback reduction are around 3% when doubling the LLC associativity.

2.5.3 Impact of Insertion Policies

Applications with constrained hardware budgets may halve the shadow tag hardware by adopting a simpler version of ARI, with relatively low impact on performance.

We performed three experiments with the following policies: 1) "Insertion-only" policy (no adaptive replacement) where sample sets are used to determine the best partition sizes, but then only the incoming cache blocks are inserted dynamically, while evicting only from LRU position; 2) ARI with fixed insertion, i.e. only MRU-, or only LH-insertion; 3) a '$p+1$ scheme' which implements 9 sample policies with alternating insertion (i.e. 0H16-MRU, 2H16-LH, ..., 16H16-MRU) and one additional 'flipped' policy which assumes the same stack distance distribution as the best policy for a given epoch, but the insertion is inverted, i.e. if the best policy is 2H16-LH, then the corresponding flipped policy will be 2H16-MRU. The intent is to allow coverage

of the entire space of $2 \times p$ policies using only $p + 1$ actual samples.

Insertion-only policy does not perform well, only improving misses by 1%, while increasing writebacks by 1.5% beyond LRU. This is because the sampling is done with adaptive ARI policies, which may have different best partition sizes than conventional LRU eviction policy.

We found that using ARI with LH-insertion yields an 8% writeback reduction compared to ARI with static MRU-insertion.

The $p + 1$ scheme reduces writebacks as well as ARI, although it sacrifices 2% miss reduction. These results indicate that the adaptive insertion provides a small but significant boost in ARI in terms of both miss rate and writebacks, though it comes at the cost of doubling the number of sample sets.

2.5.4 Minimizing Hardware Overhead

We examined a number of options to minimize the hardware overhead. We summarize the results here.

1. We examined a simple hashing of the tag bits in the sampled sets, such that the top six bits of the tag is XOR'ed with the bottom six bits, generating a tag of 1/2 the original size (i.e. 6 bits instead of 12 bits). This technique impacted writebacks and misses by $< 1\%$ and is used for all results in Sections 2.4 and 2.5.

2. Another way to reduce overhead is to utilize live sampled sets [72], each of which implements a static policy, instead of shadow tags as we propose. Simulations show this approach to increase writebacks by 5% and misses by 0.8% comparing to the shadow-tag approach.

3. We varied the number of sampled sets used between 72 and 9, keeping the number of sample static policies at 9. We found that the difference is within

1.5% for writebacks and 2% for misses. Using 9 sampled sets instead of 72 yields an 8× decrease in the shadow tag storage.

4. We varied the sampling epoch length between 10k and 40k cache references. Though the difference in performance was not significant, we found the maximum reduction in both miss rate and writebacks occurred for epochs of 25K references.

5. We evaluated an interpolation approach where we only sample the data for $p/2$ policies each half-epoch, and then reconstruct the interpolated data for the full p policies. This reduces the sampling hardware by 50%. We found this approach to be inferior to the fixed LH insertion scheme.

6. We varied the number of sample policies from $p = 9$, to 5, and to 3. We found that, compared to $p = 9$, the scheme with 5 policies is only 2% behind in writebacks and 1.2% behind in misses on average, while the scheme with 3 policies is 6% and 3.2% behind, respectively, which we believe is reasonable considering the 3× reduction in the number of sample policies.

2.6 Related work

A number of studies have been dedicated to improving the performance of cache replacement policies [35, 45, 95, 31]. [21] dynamically select two variants of Segmented LRU algorithm with insertion bypassing. Michaud [62] modified the DIP scheme for use with the CLOCK algorithm [11], and used 4 dueling policies as opposed to 2 in DIP. Khan and Jimenez [37] extended the DIP scheme to multilevel dueling, decreasing the number of leader sets. But this requires several "rounds" of dueling and switching the policies in the leader sets. Ishii et al. proposed to vary the insertion positions of incoming lines in the LRU stack based on their observed reuse

possibility [29]. This scheme utilizes set dueling and has an additional overhead of 8kB beyond DIP, and requires communication circuitry from CPU core to the LLC. RRIP [31] "predicts" the reuse interval. It uses set dueling to choose between two NRU-based policies and gains 4% speedup on average. In contrast, our scheme uses sampling between $2p$ policies and has 4.9% average speedup. We compared ARI against one of the best-performing LLC schemes, the Deadblock-predictor [38], and showed that our scheme is superior in reducing the writebacks, which is very important in PCM-based systems. These recent works mostly disregard the effect of writebacks on the DRAM, and put focus on reducing cache misses. In contrast, our scheme aims at reducing writebacks while simultaneously decreasing the cache miss rate.

With the embedded designs placing greater pressure on the memory system hierarchy, this focus will have to change as off-chip bandwidth becomes a highly constrained commodity. Additionally, writes in the main memory can interfere with reads [44]. Furthermore, as power consumption becomes an important issue in processor design, the extra power required by main memory writes makes reducing writebacks advantageous regardless of the main memory implementation technology.

Goodman discussed the impact of writebacks on memory system bandwidth [24]. Clean First LRU (CFLRU) replacement policy for page cache of SSDs [67], has similar motivation with ours in reducing expensive writes. However, our focus is on appropriate last-level, on-die cache, block replacement policies for main memory. Further, if CFLRU were mapped to CPU LLC, it would roughly correspond to our *nH16* static policy; we showed that ARI outperforms all static policies in reducing misses and writebacks. Wang et al. proposed a Last-Write prediction LLC policy [101] where the dirty blocks predicted to not receive any more writes are speculatively written back to memory before they reach the LRU position. This scheme can be

incorporated into ARI low-hit partition to give two benefits: 1) more intelligent way of breaking the ties in case the partition is full with dirty blocks; 2) more clean blocks in the low-hit partition allows for more room to store other low-hit dirty blocks.

The idea of set sampling has been fruitfully used in the literature [34, 38, 32]. Hybrid cache insertion uses set dueling [72] to dynamically choose between multiple insertion policies. The previous set sampling policies examine the LRU stack insertion points for new lines, while we attempt to determine the best region in the set for replacement *as well as* insertion.

DRAM-aware LLC management policies have been proposed [46, 44, 43, 94]. Eager writeback [46] writes dirty cache blocks to memory during idle cycles, to provide more clean blocks as eviction candidates, effectively shifting the writes in time. Virtual write-queue [94] uses a fraction of the LLC as a write queue for the memory, and the writebacks from the cache are governed by DRAM-controller. Lee et al. [44, 43] exploit row-buffer locality to guide the eviction decisions and minimize the write-caused interference. The memory-aware schemes can be easily integrated with ARI in the following way. The high-hit partition can be left managed by LRU, while memory-aware policies can be utilized in the low-hit partition to further decrease the bandwidth consumption and write interference in DRAM systems. While these techniques can be adopted in DRAM-based main memory, they are detrimental to the lifetime of PCM-based systems as they increase the number of writebacks by an average of 5-10% beyond the conventional LRU scheme [94]. Write interference in PCM memories has been shown to be a minor issue as the writes can be paused to give way to reads [71].

PCM is receiving significant attention for use within memory hierarchy. Hybrid DRAM+PCM memory architectures have been investigated [103, 14]. Wear leveling algorithms have been proposed to distribute writes uniformly [73, 86]. Write reduc-

tion techniques [71] and techniques for improving PCM lifetime [42, 41, 18] have been proposed for hardware implementation. Ramos et al. [80] and Yoon et al. [106] have presented smart page placement techniques for hybrid memory, based on "popularity" and row buffer hit ratios, respectively. [55], Meza et al. [61], and Qureshi and Loh [75] explored hardware support for large-size, fine-granularity DRAM caches, where conventional architecture is impractical due to huge SRAM tag storage overhead. By storing tags in the DRAM itself, they decrease the cache access latency, thus boosting the performance. However, large DRAM caches may be inefficient in mobile applications due to high static power consumption. ARI implemented in L3 cache helps reduce the number of accesses to such DRAM cache, which may help reduce the energy consumption and boost the performance of hybrid architectures. These works on DRAM+PCM hybrid architectures are largely orthogonal and possibly complementary to the work presented here, as we focus upon reducing writebacks from the lower levels in the memory system. Furthermore, by adapting insertion policy as well as replacement policy, ARI further reduces writebacks while reducing misses.

2.7 Summary

In this chapter we presented ARI, a technique for dynamic cache management capable of reducing the memory system bandwidth, by optimizing both misses and writebacks at the same time. We have shown ARI to perform well under various workloads, such as single-threaded and multithreaded applications in a CMP. ARI provides 33% LLC writeback reduction and 4.7% miss rate reduction on average, yielding 8.9% memory bandwidth improvement, and 4.9% IPC speedup. We find that a PCM-based system with an LLC utilizing our scheme uses 8.9% less energy, and enjoys a 49% lifetime improvement on average, as compared to conventional

LRU.

In the future we plan to explore further adaptation schemes and optimize the stack distance estimation. Also, it seems possible to employ different metrics in selecting a cache replacement policy through sampling.

3. VATS: VARIABLE AGGREGATION TIMESLICE SCHEDULING *

Memory performance is important in modern systems. Contention at various levels in memory hierarchy can lead to significant application performance degradation due to interference. Further, modern, large, last-level caches (LLC) have fill times greater than the OS scheduling window. When several threads are running concurrently and time-sharing the CPU cores, they may never be able to load their working sets into the cache before being rescheduled, thus permanently stuck in the "cold-start" regime. We show that by increasing the system scheduling timeslice length it is possible to amortize the cache cold-start penalty due to the multitasking and improve application performance by 10–15%.

3.1 Introduction

Last-level cache (LLC) sizes have grown substantially with the continued march of Moore's Law, with LLC sizes up to 45 MB for recent Intel [27] and 32 MB for IBM [90] processors. These large LLC sizes, however, are beginning to challenge commonly held assumptions in operating system (OS) timeslice scheduling behavior. In particular, fill times for large LLCs have grown to the point that they are a significant portion of the application's scheduled timeslice, resulting in lost performance and efficiency, particularly in highly loaded, multi-process, multi-core environments. This problem promises to be exacerbated as new technologies, such as 3D stacking, make extremely large LLCs feasible [54]. In this chapter we examine how OS scheduling can be adapted for the large LLC sizes found in today's processors, to

*Viacheslav V. Fedorov, A. L. Narasimha Reddy, and Paul V. Gratz. 2015. Shared Last-Level Caches and The Case for Longer Timeslices. In Proceedings of the 2015 International Symposium on Memory Systems (MEMSYS '15). ACM, New York, NY, USA. DOI=http://dx.doi.org/10.1145/2818950.2818968

reduce contention between applications and improve performance in multi-process, and multi-core environments.

In typical computer systems, the OS coordinates application access to system resources, from I/O devices to time on the processor cores themselves. Current OSes have developed fair scheduling algorithms, such as the "completely fair scheduler" (CFS) in the Linux OS, which ensures all running applications' timeslices together constitute an equal share of running time on the processor(s). To provide the illusion that each application has sole ownership of the processor, timeslices are kept fairly short, with 1–5ms being a typical timeslice interval in current systems. Although, in current multi-processor (multi-core) systems, more than one application or thread can actually run simultaneously on different cores, in many situations, particularly server and cloud implementations, many more processes must execute than there are physical processors available so timeslice scheduling persists.

The concept of timeslices emerged long before the era of modern huge LLCs. Historically, it was safe to assume the cache-fill time for a thread would be insignificant because it constituted only a small fraction of the whole timeslice. While this was a reasonable assumption with LLC sizes $\leq$256KB, with current LLC sizes in the Megabytes, this assumption ceases to be valid. Consider a case where each main memory access takes 200 CPU cycles. Assuming random accesses to memory (not streaming), filling a 4MB LLC with 64-byte lines would require fetching 65,536 lines. This would take about 13M cycles, or 6.5ms for a 2GHz CPU. Comparing this time to a typical OS scheduling timeslice length of 1–5ms, it is clear that modern short timeslices might lead to the performance degradation through LLC interference - the threads just do not have enough time to load their working set into the cache. Moreover, the modern workloads exhibit growing working sets which demand more cache space, further amplifying the effect. The larger numbers of threads found in

multicore systems exacerbate the problem since a thread may not be scheduled until all other threads have been scheduled in round-robin fashion, thus increasing the chances that the entire working set of a thread is knocked out of the LLC, even at larger size LLCs. Virtualization contributes to this trend of higher number of simultaneous threads sharing the LLC.

In summary:

- Large modern LLCs require significant time to fill the cache;
- Large numbers of threads in a multicore system lead to cache contention among parallel threads as well as across threads time-sharing the cores;
- Large working sets in modern workloads require more cache space thus take longer to fill the LLC;
- OS scheduling timeslices have traditionally been kept short to provide system interactivity.

These factors unleash severe performance degradation in system workloads. As a result, in real world situations where more threads are being scheduled than there are cores available, each distinct thread (a) experiences a significant cache cold-start effect *every time it is scheduled* on a CPU; (b) is further slowed down by more aggressive, cache-thrashing threads running in parallel; (c) suffers low IPC due to LLC cache misses during the *majority of its timeslice*.

Others have studied the mitigation of this type of contention and interference in shared LLCs. There has been some research on characterization of the misses caused by OS context switches in relation to the CPU cache sizes [52, 53]. The mitigation techniques predominantly rely upon hardware modifications, such as modified cache replacement policies [35, 39, 45, 95] and cache partitioning schemes [76, 78, 84, 96, 98].

Software techniques mainly deal with thread placement [108]. We note that with conventional (16 to 32-way set-associative) cache designs, it is challenging to ensure fair partitioning in many-core systems. Even in an 8-core system with two threads per core, a total of 16 threads share the cache. This leaves about 2–4 ways per thread on average, and requires costly hardware and repartitioning of the cache every several milliseconds due to the OS context-switching in new threads which may have different cache demands.

Further, much of the prior work does not even *consider* the effect of the OS time-sharing the cores, and instead assumes that each thread has a permanently dedicated core with ample time to reach an LLC steady state (often 1–10 billion instructions simulated with no time-sharing). In the real time-sharing systems, threads run for 1–5ms ($\sim$10 million instructions executed per timeslice), or up to 100$\times$ less, before another thread is scheduled on that core. While this behavior is of little consequence for instruction and data access streams of individual threads, the LLC utilization and hit-miss patterns are greatly affected by the fact that several threads contend in the cache in time per core *as well as* across cores.

Unlike prior work which focuses on hardware techniques to deal with LLC contention, in this chapter we explore a different approach. We aim to minimize LLC contention via a purely systems software approach. We look at OS scheduling as a possible approach to improving the system performance.

3.2 Motivation

Cache sharing leads to interference in the LLC. To show the worst-case performance degradation due to this sort of interference, Figure 3.1 presents the throughput degradation of a set of benchmarks taken from the SPEC, PARSEC, and SPLASH benchmark suites, running under the conventional CFS scheduler concurrently with

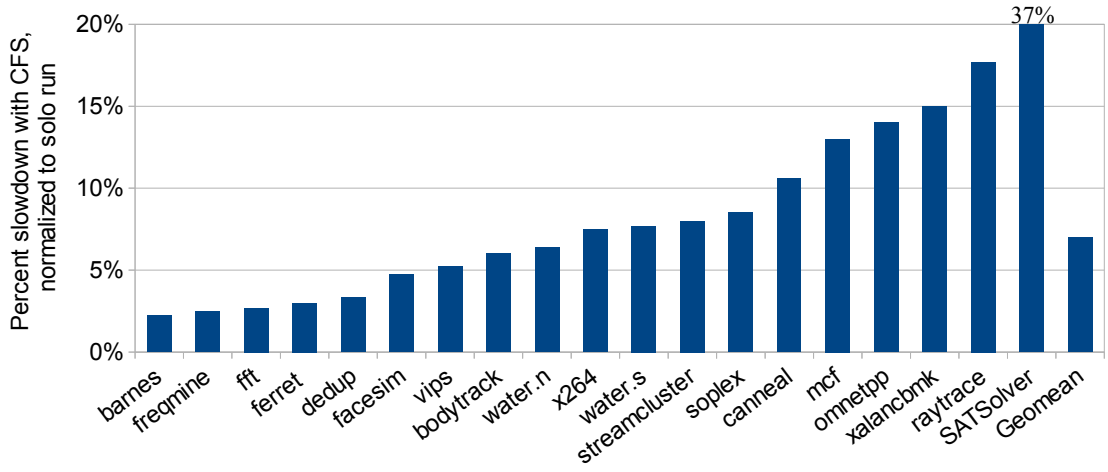


Figure 3.1: Worst-case slowdown of benchmarks running concurrently with cache aggressive microkernels, with conventional scheduler, normalized to solo run.

a cache interference inducing microbenchmark each, as compared to running solo. The microbenchmark has a significantly larger working set than the LLC size and it has a high enough access rate that a large fraction of the LLC will be flushed during its timeslice. Under these conditions, we see that the applications experience up to 37% slowdown, and 8% on average, due to the LLC interference.

With modern multi-core systems, not only is the cache shared across the threads running in parallel, but it is also time-shared by the threads sharing each distinct core. As a result, the interference is amplified. In an 8-core system with only 4 threads per core, each distinct thread will experience the accumulated effects of interference from 31 other threads when it is scheduled on the core again.

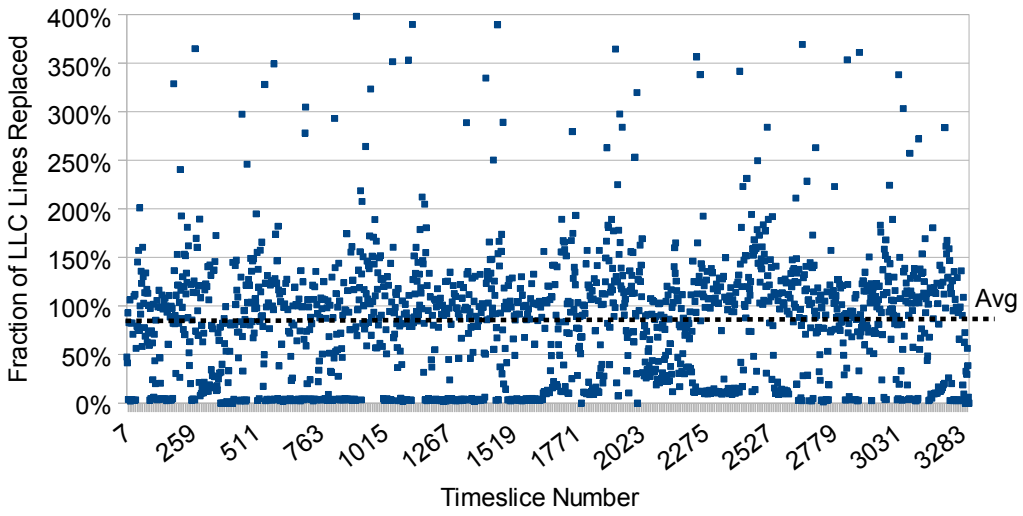
Just how severe is this penalty? If we can estimate what percentage of the LLC gets overwritten during the time a thread is off the CPU, it is possible to tell approximately how much of the thread's working set is retained in the cache between context switches, and thus predict its performance.

Figure 3.2 presents the cumulative LLC misses, as a fraction of the total LLC size, caused by interfering threads during the interval while an observed thread is off the CPU. E.g. in a system with four threads per core, this interval would amount to three timeslices. The X axis depicts the samples (every interval between reschedulings), and the Y axis shows the percentage of the cache lines overwritten since the thread's previous timeslice. This data was obtained using the CPU performance counters, for a *SATSolver* application between consecutive context switches [1]. The experiment was performed on a 2-core system with a 4MB LLC (65K cache lines). Results are presented for both two threads per core and four per core (4 threads and 8 threads total, respectively).

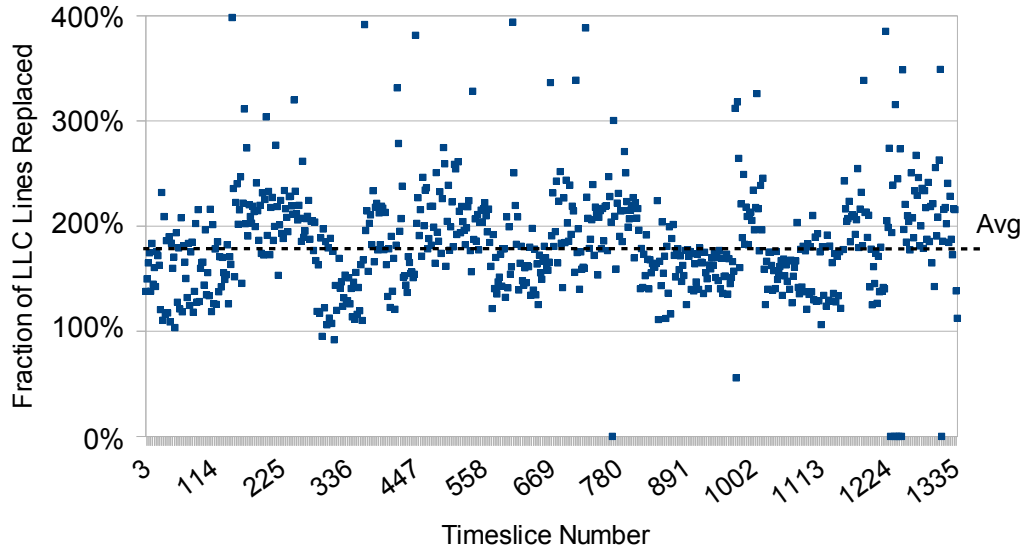
The figures indicate that with as little as two applications per core, the whole LLC is effectively fully replaced, with 84% of the cache lines being overwritten on average between thread's rescheduling. Worse, with four applications per core (Figure 3.2b), the LLC is fully replaced in all of the cases. Even taking into account the possibility of cache interference (when several threads have misses on the same cache line), the observation that 170% of the LLC lines are overwritten, enables us to expect with high confidence that once a thread is context-switched off the CPU, it then returns to a cold cache.

Further analysis, using data access traces obtained with the PIN toolkit [57], shows that, between two consecutive schedulings, each thread re-accesses an average of about 75% of the data it touched during the preceding timeslice. These results indicate that (a) **every** thread starts with a cold cache after **every** context switch, and (b) by increasing timeslice length, we could allow threads enough time to more effectively utilize the cached portion of their working set, thus amortizing the cold

[1]Here we measure the amount of cache thrashing due to all the other threads in the system, thus application choice is irrelevant.



(a) Dual-core system with 2 applications/core.



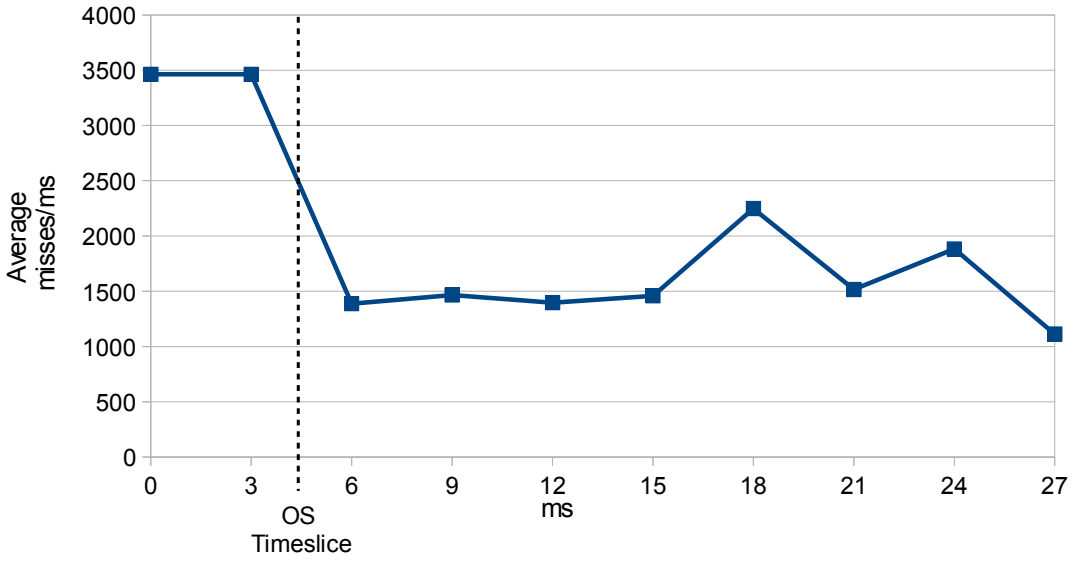
(b) Dual-core system with 4 applications/core.

Figure 3.2: Number of LLC misses between reschedulings, as a percentage of LLC lines replaced.

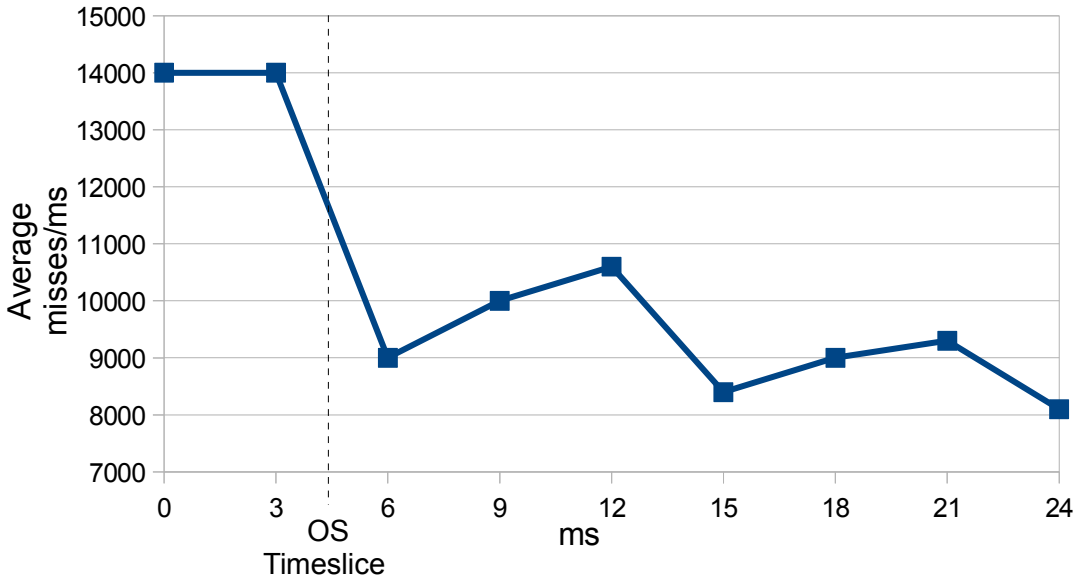
cache effect. An alternative approach of partitioning the cache to keep the working sets from getting thrashed, requires extra hardware and, as we show later, does not provide a substantial performance gain beyond what timeslice aggregation can achieve.

The LLC interference is aggravated by short OS scheduling timeslices. Once context-switched out, and then rescheduled back onto a core, a thread does not execute long enough to fully load its working set into the cache and is thus forced to run with high cache miss rate (low IPC). Figure 3.3 illustrates this point. Figure 3.3a shows the LLC Misses per *ms* for a thread running the *Raytrace* application in a time-shared multicore system inside a single, extended time slice. Figure 3.3b shows analogous data for a thread running *Xalancbmk* application. The data was sampled every 3ms using the CPU performance counters. We note that, during the first 3ms of execution (which is approximately the normal OS timeslice length in this case), each thread experiences a high number of misses due to the cold cache effect described. Note, after the working set is filled into the LLC (from *6ms* on), the thread experiences a much lower and stable miss rate. Thus at least $3-5$ ms of execution is required before cold misses become insignificant for **every timeslice**. At least 2–3$\times$ that time is needed to amortize the high initial miss rate. Unfortunately, in a conventional system, the threads would be de-scheduled from the core after $\sim$5ms of execution. Utilizing longer OS timeslices, however, the system can amortize the cold cache effect of context switches, allowing it to more effectively leverage the LLC space, boosting the thread performance and decreasing the memory bus contention due LLC misses.

One potential side-effect of increased time slices is that threads will more thoroughly fill the LLC, which in turn, increases the temporal cache thrashing with respect to all the other threads which run later. However, because the cache data



(a) Raytrace.



(b) Xalancbmk.

Figure 3.3: Misses/ms for an extended timeslice, sampled every 3ms.

retention for each distinct thread between schedulings is very low to begin with (i.e. <10%), this side-effect is far outweighed by the improvement in LLC miss rate.

3.3 Design

Our design is illustrated on the Figure 3.4. The figure shows a snapshot of the three types of OS schedules on one core, with two threads being scheduled. We start from a conventional CFS timeslice schedule, as shown in the top of the figure. Note how after each context switch, threads are paying the cold cache penalty, hence suffering degraded performance.

The naïve approach to increasing the OS timeslice length (illustrated in the middle of Figure 3.4) is to tweak the constants in the Linux kernel. Note that the cold cache penalty is amortized across the longer thread running times. This approach is, however, imprecise and requires frequent re-tuning in order to maintain the desired timeslice length. The scheduler constants only provide an upper bound for the scheduling slices, and when the system load changes, the kernel automatically shortens the timeslices in response to the increasing number of threads in the system.

An alternative approach, presented in the bottom of Figure 3.4, which we call *timeslice aggregation*, is to modify the scheduler code in such a way that it would consecutively reschedule a given thread for a set amount of time before switching to the next thread (selected utilizing the long-run time sharing fairness of the Linux CFS scheduler). In such a scheme, instead of having one long, monolithic timeslice, several timeslices are effectively aggregated. This allows precise timeslice control, as well as allowing the scheduler to switch to a higher-priority thread (such as an interrupt bottom-half), if needed, without having to wait for the current low-priority thread to finish its full extended timeslice. Aggregation guarantees low impact on system interactivity, although it may provide lower performance boost in highly-interactive

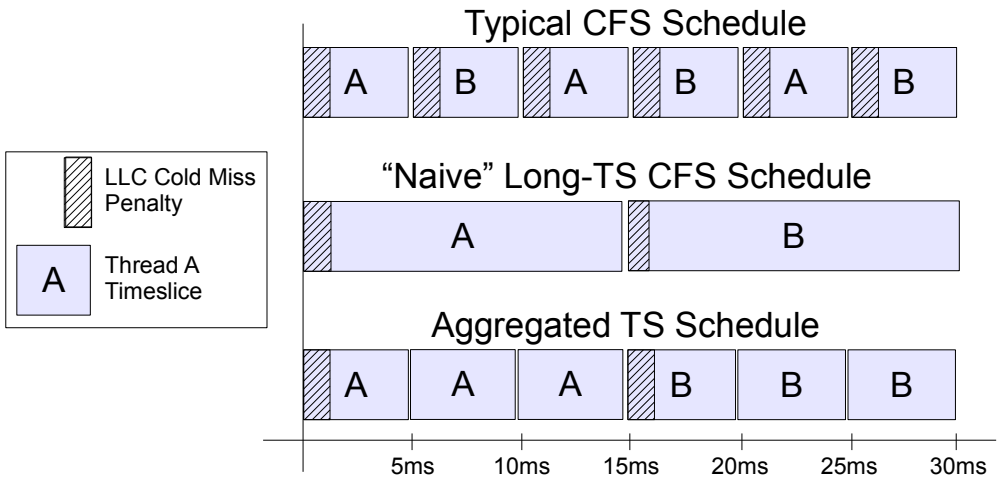


Figure 3.4: A comparison of the conventional CFS timeslices vs. proposed aggregated timeslices.

systems with frequent interrupts.

3.4 Evaluation

3.4.1 Methodology

We implemented our proposed the timeslice aggregation scheme in the Linux kernel. Standard Linux timeslice length was used as a baseline. Note, in Linux, timeslice length varies between 1ms and 15ms dependent on system load (number of threads running) and number of CPUs in the system, with shorter time slices as the load increases.

We ran our experiments on an an Intel Xeon E5 with 16MB LLC and 90GB of RAM. We emulated double- and quad-core configurations by pinning the threads to specific cores. Additionally, we ran experiments on a smaller system, an Intel Core 2 duo E6550 processor with 4MB last-level cache, and 8GB of RAM, in order to gain some insight on how the LLC size affects the performance degradation due to thread contention. We utilized memory-oriented applications from the PARSEC 2.1,

SPLASH-2x, and SPEC CPU 2006 suites [5, 4, 26] running with their native (or equivalent) input sets. Namely, barnes, bodytrack, canneal, dedup, facesim, ferret, fft, freqmine, mcf, omnetpp, raytrace, soplex, streamcluster, vips, water.nsquared, water.spatial, x264, and xalancbmk. Additionally, we used a SATSolver benchmark with an input set from a SAT contest [2]. For these experiments, we selected those applications which have working sets larger than the L2 cache, *i.e.* those which show at least 3% performance degradation when co-scheduled with a memory-intensive microbenchmark. For two-core experiments, we arranged 16 mixes of four threads per core, with a total of eight threads per mix. In four-core experiments, we utilized four threads per core for a total of sixteen threads per mix.

3.4.2 Performance Improvement

Threads running in a multi-core multi-process system experience severe LLC interference. One of the major implications of this interference is that each thread starts each timeslice with a cold cache. By allowing threads to run longer before a context switch, we can effectively amortize the performance lost due to cold cache startup, thus boosting the system throughput without any hardware modifications.

We present two sets of experiments on a quad-core machine with 16MB LLC. First set consists of 16 mixes of 4 applications per core, run on two cores, for a total of 8 applications per mix (applications are pinned so two cores are held idle). The second set consists of 6 mixes of 4 apps per core, run on four cores, for a total of 16 applications per mix. In both experiments, we compare the geometric mean runtime improvement of the applications with four aggregated timeslices (15ms, 30ms, 45ms, and 70ms) versus the conventional CFS scheduler short timeslice (∼4ms).

Figure 3.5 shows the geometric mean runtime improvement figures for application mixes in two-core experiments, normalized to a system with conventional timeslices.

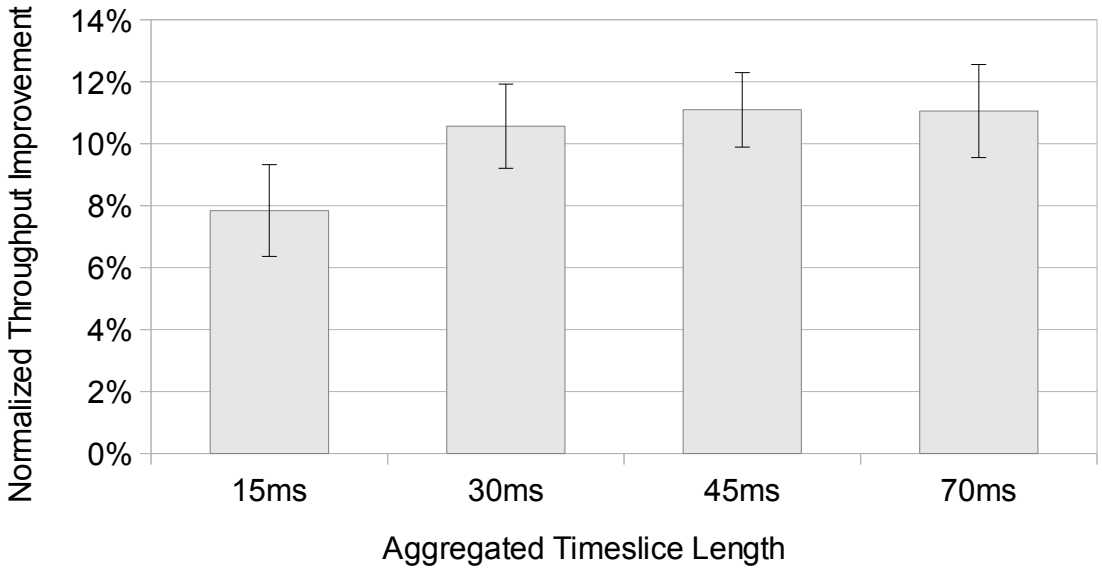


Figure 3.5: Runtime improvement versus CFS baseline in a dual-core system with 16MB LLC. Geomean across sixteen application mixes, four applications per core.

The error bars depict the variance in performance across the applications (±1 standard deviation). We observe a gain of 8–11% on average. With the large 16MB LLC, it takes a considerable amount of time for applications to load their working sets in the cache, thus the performance improvement increases from 15ms to 30ms timeslice length due to the increased amount of time to amortize LLC cold start cache misses. However, beyond 30–45ms timeslices, there is only marginal performance improvement since the applications' working set data is then loaded in the LLC and the cold start penalty amortized.

Figure 3.6 presents the runtime improvement figures for application mixes in a quad-core system, normalized to a system with conventional scheduler. The error bars depict ±1 standard deviation. Again, we observe diminishing returns on throughput improvement beyond 45ms timeslices. Here the benefit is slightly larger than in the dual-core case, in part due to the more thorough replacement caused by

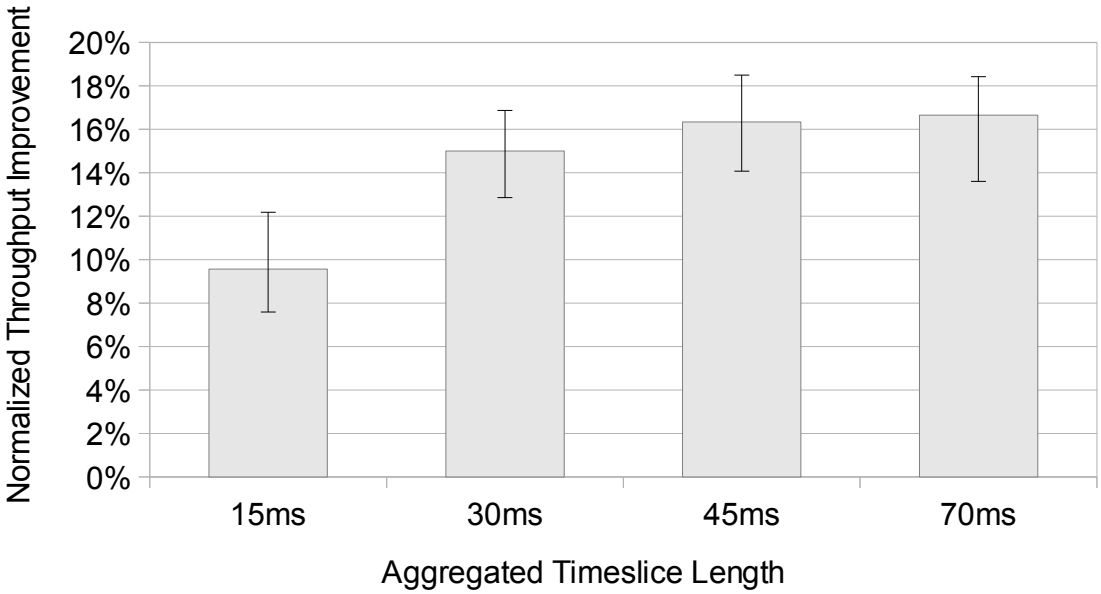


Figure 3.6: Runtime improvement versus CFS baseline in a quad-core system with 16MB LLC. Geomean over six application mixes, four per core.

running more simultaneous applications.

3.4.3 Effect of LLC Size on Optimal Timeslice Length

Intuitively, larger caches take longer to fill than smaller ones, so the threads need longer timeslices when running with a larger LLC, in order to amortize the cold cache penalty. We conducted the same set of two-core experiments as in Section 3.4.2 in a true two-core system with 4MB cache. Figure 3.7 presents the geometric mean runtime improvement figures, for sixteen application mixes in a dual-core system with aggregated timeslice lengths from 5 to 30ms. Each bar reflects the performance improvement normalized to running the mixes on a system utilizing the conventional CFS Linux scheduler with standard timeslices ($\sim$2ms in this case due to a smaller number of cores in the system and initial CFS settings). Again, the error bars depict $\pm$1 standard deviation.

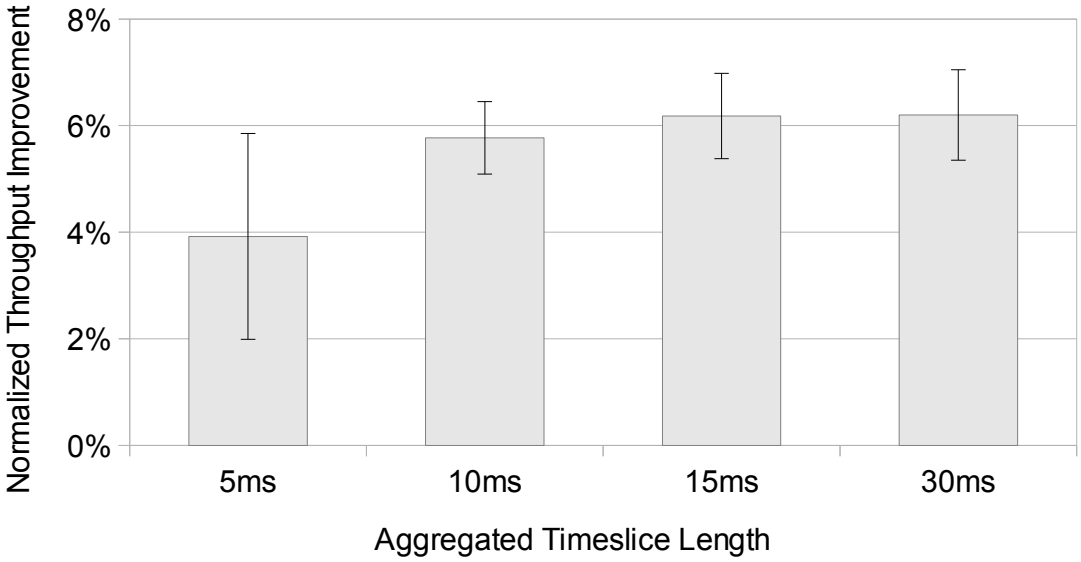


Figure 3.7: Runtime improvement over CFS baseline in a dual-core system with 4MB LLC. Geomean across sixteen application mixes, four applications per core.

In the figure we see that here timeslice aggregation with 5ms-long timeslices yields about 4% geomean runtime improvement, while 15ms-long timeslice aggregation boosts the performance by 6.2%. Versus the previous results with a large LLC (Section 3.4.2), we note that: (a) the overall performance improvement is less, and (b) the performance benefit saturates at a shorter relative timeslice length. The differences primarily derive from the fact that a smaller cache takes less time to refill on cold restart than a larger cache, thus the total performance penalty is less and it takes less time to amortize (about 15ms versus 30-45ms with larger LLC). We conclude that the optimal timeslice length for a given system depends on the size of the LLC.

With caches getting larger and the number of threads increasing, the data seems to suggest that longer timeslices are needed to amortize the cold start penalty and improve the system performance.

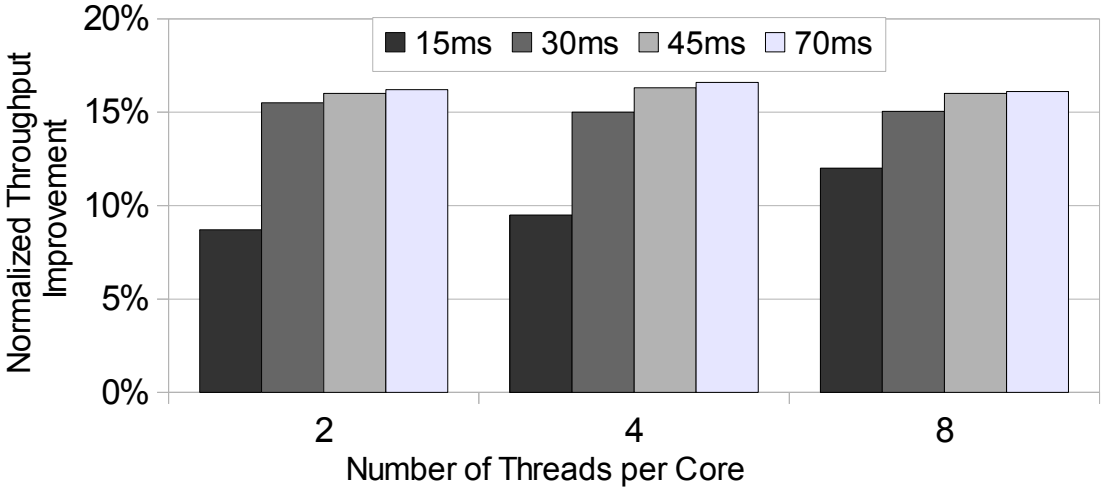


Figure 3.8: Runtime improvement over CFS baseline with 2, 4, and 8 applications per core, in a quad-core system.

3.4.4 Performance vs. Application Count

Here we examine sensitivity to the number of applications per core. Figure 3.8 shows the results for experiments with two, four, and eight applications per core. Across all experiments, the performance improvement saturates at ∼15% for aggregated timeslices larger than 30ms. This is in line with our reasoning that shorter timeslices are not quite sufficient to amortize the cold cache penalty. The slightly better results in 8 thread-per-core experiments can be explained by the fact that additional threads more thoroughly replace the contents of the LLC, leading to more performance loss on cold restart.

3.5 Discussion

Figure 3.9 presents the per-application runtime improvement for one application mix in a quad-core system, with 45ms aggregated timeslices. The figure shows that all the applications benefit from longer timeslices, with an average improvement for this particular mix of ∼12%.

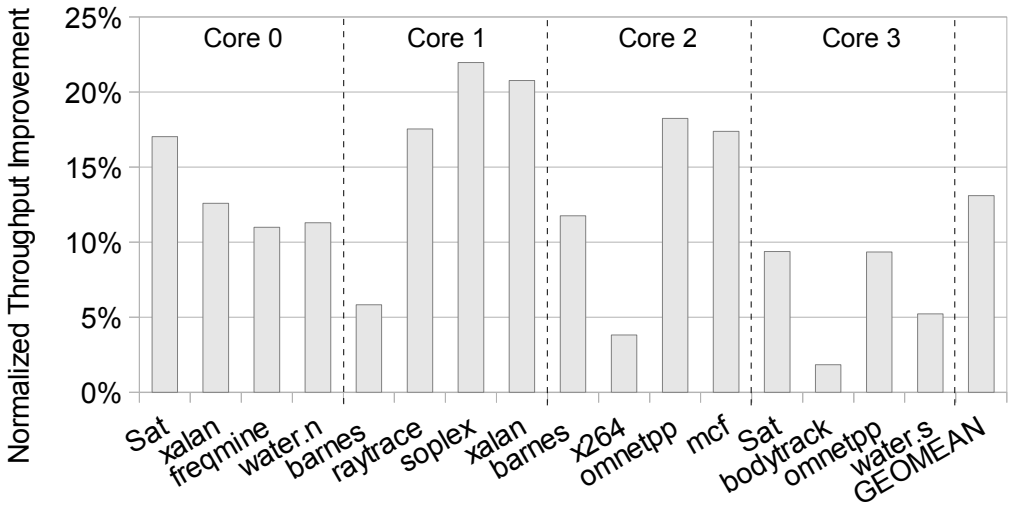


Figure 3.9: Detailed runtime improvement, 45ms aggregated timeslices versus CFS baseline in a quad-core system with 16MB LLC. Four applications per core, sixteen total.

3.5.1 LLC Miss Reduction

Figure 3.10 presents the total number of LLC misses for one mix of applications with aggregated timeslices, as a fraction of misses with conventional CFS. We observe that timeslice aggregation helps alleviate about 15% to 18% of the total LLC misses. Note that these misses are forced cold cache misses due to multitasking, hence reducing them leads to pronounced improvement in system throughput, as we have seen in Section 3.4.2. The geometric mean LLC miss reduction per application is 30% (we do not present the figures for brevity). Again, we observe that about 30-45ms timeslice length is enough to amortize the cold start penalty, and beyond that we hit diminishing returns (*i.e.* the rest of the misses are conflict and compulsory misses intrinsic to the application mix running in the system).

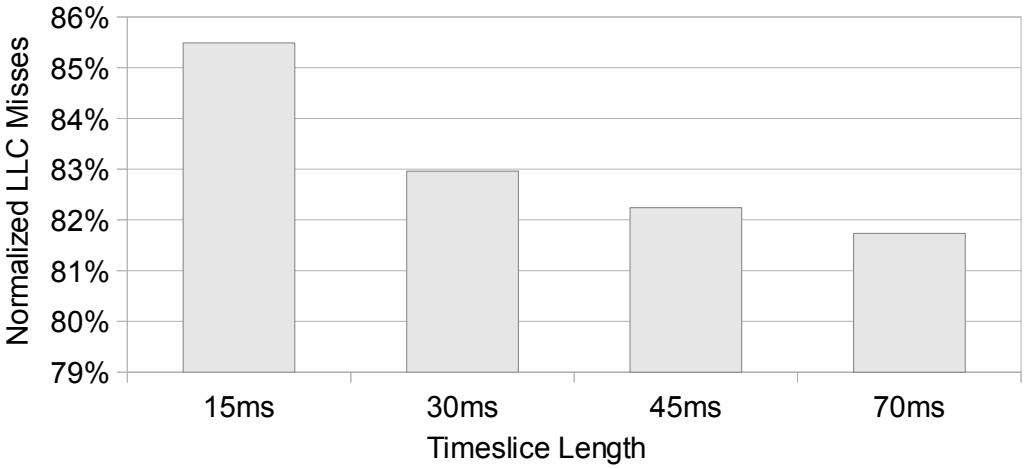
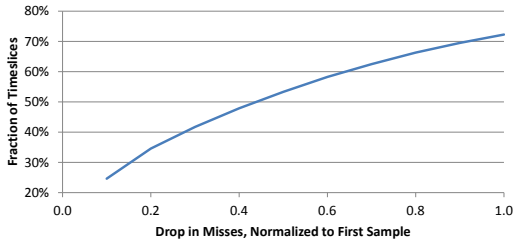


Figure 3.10: LLC misses with aggregated timeslices, normalized to CFS in a quad-core system with 16MB LLC.

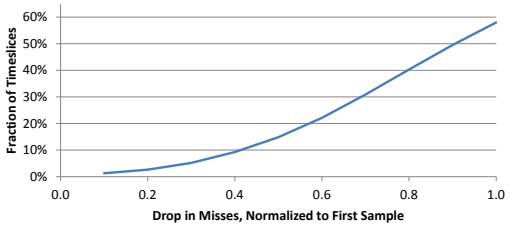
3.5.2 Cache Partitioning vs. Timeslice Aggregation

We notice that timeslice aggregation scheme improves the application performance by 10–15% in multi-core systems. The interesting question is, can this technique be augmented by partitioning the cache? Longer timeslices help to amortize the cold cache effect, but they do not prevent it. With cache partitioning, each thread would retain its cache contents across timeslices thus avoiding the cold-start effect completely. The catch here is that it is challenging to effectively partition a 16-way cache among 10–20 threads. Further, a partitioned cache gives a fraction of its space to each application, while with timeslice aggregation, all the cache is available to the application once it has been refilled. Intuitively, splitting applications into groups based on their cache size demands, one would need to enforce partitioning for a much smaller number of threads at a time (*i.e.* partition the cache into 4 pieces where 3 applications get 1 full piece each, while the rest of the applications share a single partition; rotate the partition assignment every 200ms).

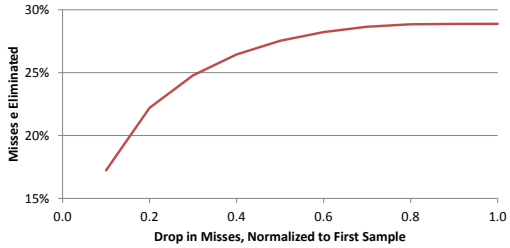
Here we analyze an ideal case, assuming that the cache is equipped with a parti-



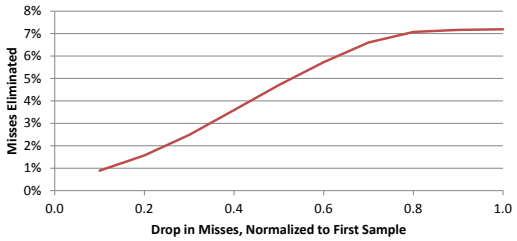
(a) Freqmine cold cache effect.



(b) Soplex cold cache effect.



(c) Freqmine miss improvement beyond 15ms aggregation scheme.



(d) Soplex miss improvement beyond 15ms aggregation scheme.

Figure 3.11: CDF functions for cold cache effect, and estimated LLC miss improvement beyond 15ms aggregation scheme.

tioning scheme which is able to split the cache among all the threads in the system, based on cache utility or working set sizes, etc., such that each thread achieves the maximum benefit. We gathered the statistics of the Miss rate fall-off behavior among four mixes of applications. Looking at Figure 3.3a as an example, let us take the first data point (at 3ms) as the maximum (100%). Now, the fifth data point (at 15ms) in this example is $\alpha = 0.42$ of the maximum miss rate. We are interested in the fraction of misses that could be avoided if a perfect partitioning scheme was implemented in the cache. Let us assume a best-case scenario where the number of misses is high during the first 3ms, but rapidly drops and stays at the level of the 15ms point. We can approximate the total number of misses the thread encountered in 15ms as follows: $3ms \times Xmisses + 12ms \times \alpha Xmisses = (3ms + 12ms \times \alpha) \times X$. Now with perfect partitioning, there would be no spike in misses during the first

3ms, thus in this case the number of misses in 15ms is: $15ms \times \alpha Xmiss/ms$.

The improvement in cache misses with cache partitioning beyond 15ms aggregated timeslices can be thus calculated as follows: $1 - \frac{15\alpha}{3+12\alpha} = 1 - \frac{5\alpha}{1+4\alpha} = \frac{1-\alpha}{1+4\alpha}$. Note that the multiplier 4 in the denominator increases with the increase of aggregated timeslice length, thus the maximum miss rate improvement with long timeslices is decreased.

Figure 3.11 depicts the CDF functions for the two most representative applications from our application mixes. The two charts on the top show, for each α, the fraction of timeslices that the thread experienced a corresponding drop in misses from the 3ms to the 15ms data point. For instance, with *Freqmine* benchmark (Figure 3.11a), the miss rate drop is smaller than $\alpha = 0.6$ for 40% of the timeslices, and smaller than $\alpha = 0.9$ for 30% of the timeslices. The charts on the bottom reflect the corresponding estimated improvement in total number of misses with the data obtained from the top charts. Continuing the *Freqmine* example: with partitioning, we would alleviate 90% misses (α=0.1 drop in Figure 3.11a), which is $\frac{1-\alpha}{1+4\alpha} = 0.64\times$ better than 15ms aggregated scheme, for 25% of timeslices. The timeslices with α=0.1 drop with partitioning would contribute to 16% misses alleviation beyond the 15ms scheme (Figure 3.11c). Then, 80% misses (0.44× better than 15ms scheme) for additional 10% of timeslices, which would contribute to additional 6% misses alleviation beyond the 15ms scheme. In other words, we integrate the upper graph with the $\frac{1-\alpha}{1+4\alpha}$ multiplier, obtaining a total of about 27% Misses reduction beyond the 15ms scheme. Similarly, *Soplex* application would achieve 7% Misses reduction with cache partitioning, beyond the 15ms scheme. The more timeslices exhibit the larger drop in miss rate, the more can be gained with partitioning.

Our estimates show that for most of the applications, partitioning provides only about 5–10% additional benefit beyond 15ms aggregated timeslices. With longer

timeslices, the added gains rapidly drop. Our results thus indicate that partitioning does not seem to offer substantial additional benefit beyond longer time slices. We further note that time slice aggregation is a purely software solution without any additional hardware support, thus the overhead of implementing this scheme is much lower than partitioning. With partitioning, however, there is no cross-core interference, *i.e.* the steady state miss rates in Figures 3.2a and 3.2b would be lower with cache partitioning. Generally we find these applications are not particularly sensitive to inter-core interference, with only a $\sim$5% miss rate increase when run simultaneously with other applications.

3.5.3 Fairness

Since our scheme is based on the Completely Fair Scheduler, by increasing the timeslice length, the system fairness should not be affected. We observe that by aggregating the timeslices, the system achieves similar fairness among the applications. Note however that the aggregated timeslices allow fairness over longer time scales than the original CFS, while potentially being unfair over timescales smaller than the aggregated scheduling times.

3.5.4 Dynamic Timeslice Extension

A natural extension of our approach is to dynamically control the timeslice length, adapting to the application execution phases. The intuition is that overlong timeslices can cause more LLC interference where applications on different cores will fight for the cache space. We implemented a dynamic scheme as a modification of the CFS scheduler in Linux Kernel.

The basic idea of our dynamic scheduler was to monitor the threads' LLC fill rate and individually adjust the timeslice aggregation so that each thread is only able to fill a preset fraction of the cache. We leveraged the kernel interface to the

CPU performance counters to compute the MPTS (misses per timeslice) metric. The overhead of accessing the counters is approximately 2 μsec per read. Compared to the time one scheduler invocation and context switch takes, this overhead is negligible.

Because the dynamic scheme is an extension of CFS, regardless of each thread's timeslice length, all threads are given a fair share of CPU time. All runnable threads are stored in a tree where the key is their virtual runtime length, *vruntime*. After a thread has been scheduled and finished its timeslice, the *vruntime* variable is incremented by the timeslice length, and the thread gets put back into a tree. A new thread with the lowest *vruntime* is picked from the tree. When we are aggregating the timeslices for a thread, after one scheduling of such thread, its *vruntime* variable will get much larger than other threads' *vruntimes*. As a result the thread will remain unscheduled for a longer time. Thus the scheduler automatically takes care of fairness.

Our experiments show that, while the dynamic scheme is adapting to individual applications' behavior, it does not provide any noticeable performance improvement beyond the Systemwide aggregation. We calculated the average timeslice lengths produced by the dynamic scheme, and based on that data, it is marginally better than Systemwide (by about 0.7–1% on average) when the timeslices are shorter than 6 ms, but beyond that, there appears to be no gain from adaptivity.

3.5.5 Even Larger LLCs

It is tempting to think that with larger LLCs multiple working sets might be able to fit simultaneously and hence the benefits from longer timeslices may not persist. First, our results show that longer timeslices benefit both 4MB and 16MB LLCs, with the trend towards more benefit in larger LLCs. Second, application sizes and data sets tend to grow as larger systems become available. Virtualization and server

consolidation continues the trend of increasing resource sharing and the resultant contention at the LLC. As we have seen (Figures 3.2a and 3.2b), four applications in a system are able to overwrite a 4MB LLC leaving each thread with a cold cache every timeslice. Similarly, in an 8-core system, with four threads per core, a 32MB cache may easily be overwritten. Note that, with larger cache sizes come longer refill latencies, aggravating the cold start penalty. We thus expect the timeslice length required to efficiently amortize the cold cache penalty in systems with large LLCs to be even greater.

3.6 Summary

System performance in time-shared, multitasking systems is degraded by the cold restart fill time after every context switch seen in modern, large, last-level caches. Modern cache sizes have grown so large, and the OS scheduling timeslices so small, that the cache fill time is greater than the scheduling window. It is possible to amortize the cold cache penalty by allowing the applications to run longer between reschedulings. Here we present a simple scheme for timeslice aggregation, and show that it is possible mitigate the system slowdown due to the multitasking, boosting the throughput while observing fairness.

4. FTCAM: AN AREA-EFFICIENT FLASH-BASED TERNARY CAM *

As an unconventional approach to utilizing emerging memory technologies, we present a Ternary Content-Addressable Memory (TCAM) design with Flash transistors. TCAM is important in modern high-speed network routing. We have chosen internet routing application as a demonstration of the applicability of our design, however, such design could be utilized in Virtual Memory accelerator applications. For instance, TCAM could be leveraged for storage of the Page Table and hence fast Virtual-to-Physical address translations, obviating the need of the costly Page Table walks. Adoption of this scheme would require significant modifications to the existing processor architectures. Also the limited re-write endurance of Flash in such an application would need to be addressed. We leave this for the future work.

4.1 Introduction

For routers that serve the internet core, the routing lookup operation needs to be performed on data that arrives on optical interfaces operating at a few 100 Gb/s. As a result, it is imperative that the routing table lookups for these routers be done in hardware, in parallel, rather than in software. The circuit of choice for hardware routers is Ternary Content Addressable Memory (TCAM) [58, 68, 99]. TCAMs ideally store an entire routing table, and perform a simultaneous comparison for *all* routing entries against the destination address of the packet being routed. A TCAM is a variant of a cache (which is also referred to as a CAM [79]), with the added ability to disregard a subset of address bits while performing the lookup. The address bits that are disregarded during a routing table lookup operation correspond to the mask

bits of the routing table entry (which have 0 values).

Traditionally, TCAMs are implemented using CMOS integrated circuits (ICs), in which each TCAM cell requires 17 transistors, and each SRAM cell (which stores the interface or next hop port) requires 6 transistors. In our work, we realize each TCAM cell using 2 flash transistors, and each "SRAM" cell (we refer to it as a *port cell*) requires 1 flash transistor. As a consequence, our flash-based TCAM (FTCAM) block [16] is significantly more dense than the traditional CMOS-based TCAM block (by a factor of about 7.9×), with a lookup delay which is 2.5× larger, and a power consumption which is 1.64× lower than the CMOS-based TCAM.

The key contributions of this work are:

- To the best of our knowledge, this is the first work that utilizes flash technology to realize a Ternary CAM design.

- We implemented the flash-based TCAM block in HSPICE [60], using a 45nm PTM [70] CMOS technology, and a 45nm flash technology, yielding accurate delay and power, compared to a CMOS-based TCAM block [20], which was also implemented.

- The layout of the flash-based TCAM was generated. The TCAM block area of the flash-based design was compared with that of a CMOS based design [20], and we demonstrate that the flash-based design is about 7.9× more dense.

- Our FTCAM was simulated using a real trace of a backbone internet router, to evaluate the system lifetime and to ensure no packets were lost during real-time operation.

The remainder of this chapter is organized as follows. Section 4.2 discusses some previous work in this area. In Section 4.3 we briefly describe the architecture of our

envisioned flash-based TCAM, and provide details of the flash-based TCAM block we employ. In Section 4.4 we present experimental results comparing our flash-based TCAM block with an implementation of an existing CMOS TCAM block. Conclusions are discussed in Section 4.5. A brief background on internet routing and TCAM operation, as well as a discussion of Flash transistor basics, erase and program operations of the proposed cells, and the cell layouts of our design can be found in the Appendix.

4.2 Previous Work

A good overview of existing TCAM approaches can be found in [58, 65]. Most TCAM implementations store routing entries in blocks [87, 99], where each block contains routing entries of a particular mask length. This allows for fast lookups, since the IP address would be looked up in all blocks, and only the match from the block with the highest match length would be selected. Another implementation [40] allows routing table entries to be stored at any locations in the TCAM, but require two cycles to perform a lookup. Routing table lookup in this approach is done in a non-pipelined, two stage manner. In the first phase, the TCAM performs the lookup and performs a bitwise OR of the matching entries' masks. This produces the longest mask, which is fed back to the TCAM and further constrains the original matching entries to produce the entry with the longest prefix. The main drawback of this approach is that in lowering the cost of insertion, the cost of each lookup is doubled.

Most TCAM designs utilize a *priority encoder* [100] circuit to perform the Longest Prefix Match (LPM). The LPM computation is usually done in hardware, either using dedicated hardware [40], or by arranging the routing table entries in a specific order as described in [87].

In [50], the authors discuss techniques for reducing power in a TCAM, including

3D stacking and the use of programmable vias to save area in the port memory. As such, the techniques described are orthogonal to the ideas we present in this work. STT-based TCAM circuits were proposed [105], however due to the resistance variation of the magnetic junctions their design utilizes three memory elements per cell, as well as $3 + 11$ CMOS devices, whereas our FTCAM cell utilizes only two flash transistors. Our FTCAM uses 0.6 fJ/bit/search while the TCAM in [105] requires 7.1 fJ/bit/search. The lookup energy ratio roughly tracks the number of devices per TCAM cell. Memristor-based CAM utilizes two memory elements and three CMOS transistors [15]. However, the focus of their paper is not on TCAMs, but rather on the cell design of a memristor-based CAM. The authors of [25] present a resistive TCAM cell, to be integrated with the virtual memory, making the physical address space content-addressable. In general, the focus of this paper is at a higher level of abstraction, unlike our work.

In [20], the authors implemented an efficient TCAM, in which routing table entries are stored in any order, thus eliminating the large worst-case insertion cost of typical TCAM implementations, as described in [87]. In addition, they used an efficient Wired-NOR based LPM circuit, whose delay scales logarithmically with n, thus improving over the linear complexity (in the size of the TCAM) of priority encoder based circuits. All the above approaches utilize a CMOS implementation of the TCAM.

There have been several research efforts which study the flash devices and their use in memory. A shortened list includes [47, 1, 66, 10, 89]. These papers report details of flash devices and their characterization. However, they do not describe the use of flash transistors for TCAM like circuits. To the best of our knowledge, there has been no prior work that uses flash devices to realize TCAM structures.

In our approach, we utilize flash transistors to realize the TCAM cells. We

assume that the TCAM is realized in blocks (with 256 entries per block). Each entry consists of 256 TCAM bits (thereby supporting IPv6 routing tables), and 512 data bits. Although the focus of our work is on the design of a TCAM block, we also discuss the architecture of the entire TCAM, discussing how routing updates and route flaps would be handled. The flash-based TCAM block is compared in terms of layout area, delay and power with an efficient CMOS TCAM design [20], which was re-implemented in the 45nm PTM [70] technology for a fair comparison with our flash-based TCAM block.

4.3 Our Approach

4.3.1 Definitions

We first provide the definitions of terms used in this work. The proposed design consists of TCAM blocks, each block containing a number of Flash TCAM cells and Port cells.

Definition 1. TCAM block: *A block consisting of 256 rows of 256 flash TCAM cells and 512 flash port cells per row.*

Definition 2. Shadow TCAM block: *A CMOS TCAM block consisting of 512 rows of 256 CMOS TCAM cells and 512 SRAM (port) cells per row.*

Definition 3. LPM block: *A special block performing the Longest Prefix Match operation among multiple matching flash TCAM blocks.*

Definition 4. Flash TCAM cell: *A circuit consisting of two flash transistors, capable of storing a ternary value and comparing against the stored value.*

Definition 5. Flash Port (Memory) cell: *A single flash transistor circuit holding one bit of port (next hop) information.*

4.3.2 Overview

In the next two sections, we discuss the design of out FTCAM router following a top-down approach.

In Section 4.3.3, we discuss the architecture of the TCAM at the chip level, along with a discussion of how insertions and deletions are handled. In the proposed design, the routing entries are stored in *blocks* of a fixed size.

The focus of this work, however, is the design of the TCAM block. The design of each block is discussed in Section 4.3.4. This section covers the design of the TCAM cell and the port cell. The FTCAM block in our approach was simulated in HSPICE [60] , with wiring parasitics extracted via Raphael [81]. The layouts of the FTCAM and port cells were generated as well. The delay, area and power of the FTCAM block were compared with those of a CMOS TCAM block [20].

In existing flash memory ICs (which are used in SDCards, memory sticks, and SSDs), both CMOS and Flash transistors co-exist on the same die. The CMOS transistors are used for control operations, pulse generation, and readout management. Similarly, our FTCAM based router also uses CMOS and flash devices together, on the same die.

Whenever a TCAM based router comes online, a protocol called the BGP (Border Gateway Protocol) is run to announce its routes to neighboring routers, and, conversely, to discover new routes from them.

4.3.3 TCAM Architecture

Our design stores routing entries in blocks, with each block used for the storage of entries with a specific mask length M. For each mask length M, a fixed number of blocks N_M are employed, based on the mask length statistics of the routing table entries being stored in the router.

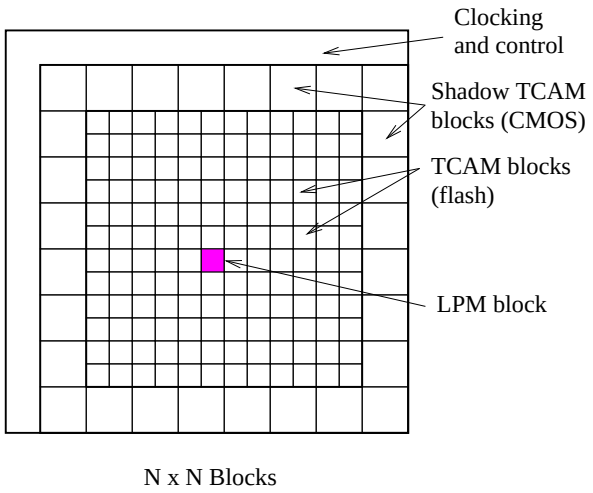


Figure 4.1: Floorplan and block arrangement of our TCAM.

The floorplan of our proposed TCAM design is shown in Figure 4.1. The total number of flash TCAM blocks in this design is $N^2 - 1$, and these are divided among mask lengths, such that $\Sigma_{M=1}^{M=128} N_M = N^2 - 1$. The largest mask length is 128 (assuming IPv6 IP addresses). A single central block is used as an LPM block, to select one entry with the longest prefix, from all the matching entries from several TCAM blocks. The LPM block can be implemented as a priority encoder [100], or a wired-NOR based circuit of [20], which exhibits a logarithmic delay instead of a delay that is linear in IP address length.

In our TCAM, as in the scheme of [87], memory management was performed external to the TCAM, in software. The flash TCAM entries can be thought of as NAND flash memory, but with extremely short device stacks (we utilize two devices in series, unlike a NAND flash memory in which there are typically 100s of series devices in a NAND stack). Each short stack corresponds to a TCAM entry, and can be erased and programmed independently. However, flash devices have long erase and program times (in the 100s of microseconds [13]. Therefore, our flash-based

TCAM has *shadow* blocks which are implemented using CMOS cells (see Figure 4.1). These CMOS TCAM shadow blocks are used to perform lookups while flash TCAM blocks are being modified (and are consequently off-line). The CMOS TCAM shadow blocks are implemented in a manner similar to [20], with each block being able to store entries of variable mask lengths. As we show later, 48 512-entry CMOS shadow blocks are sufficient to support the operation of our TCAM.

The router firmware contains the *golden* state of each block of the flash-based TCAM, in its DRAM memory. The DRAM is not used for lookup (this would be prohibitively slow), but rather as a means to ensure consistency of TCAM entries. We next discuss how route additions and deletions are performed, using the DRAM, shadow TCAM blocks and flash-based TCAM blocks. The following discussion assumes that each TCAM block has a pointer indicating an unused entry (a *hole*), and each TCAM entry has a flag to indicate whether it's contents are valid or invalid (this flag is implemented in CMOS, and can be written at the speed of the TCAM clock).

The hole pointer would require a small amount of memory per TCAM block. Each pointer requires one byte as there are 256 potential hole positions. This translates into a total of 6 KB of memory (since we can accommodate 6000 FTCAM blocks in a 1.5 cm $\times$ 1.5 cm die). This amount of memory would fit into an area corresponding to one FTCAM block. Invalid/valid information is stored as a single bit in the FTCAM array, and utilizes 2% of the FTCAM block area.

Flash transistors typically have a finite (10k - 100k) number of times they can be written [36]. In traditional flash memory, *wear leveling* is performed at the architectural level to spread the wear of the cells. In our approach, the same wear leveling techniques would be used for blocks of a particular prefix size.

4.3.3.1 Route Addition

When a new route R with mask length M is to be added, it is first written by the firmware to the appropriate location in DRAM. This location corresponds to a vacant position in a flash-based TCAM block which stores entries with mask length M. The entry R is then copied to a CMOS shadow block which is on-line and available to do lookups.

This technique filters route flaps and reduces write-stress and wearout of the flash-based blocks. Once the CMOS block is full, the process of copying to flash-based TCAM is initiated. The corresponding flash-based TCAM blocks are now taken off-line, the contents of the DRAM blocks with the new route entries from the CMOS shadows are copied to the corresponding flash-based TCAM blocks, and the flash-based blocks are then again brought on-line. At this time, the CMOS shadow block is reset.

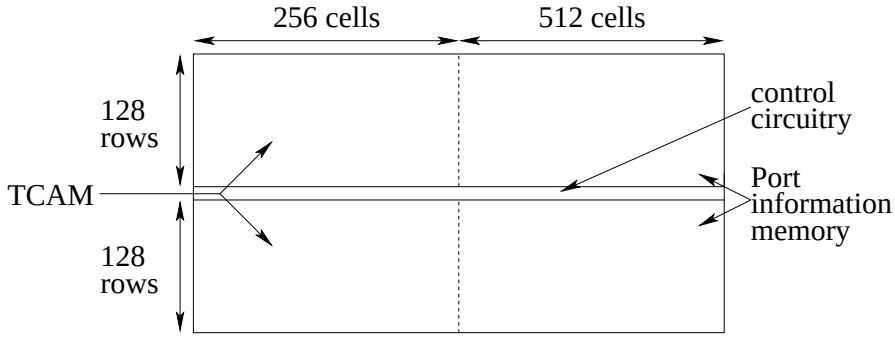


Figure 4.2: TCAM block organization.

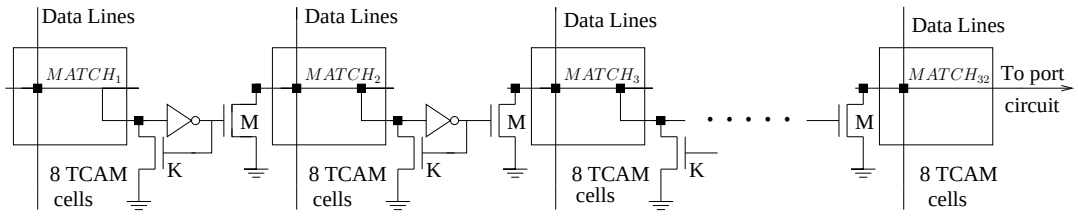


Figure 4.3: TCAM row split into 32 sections.

4.3.3.2 Route Deletion

When a route with a mask length M is to be deleted, we first do a lookup. Assuming that the route is present in the TCAM, there is a match from one of the flash TCAM blocks which store routes of mask length M. Note that this route may be present in a CMOS shadow TCAM block as well, and the next steps are identical in that case. The row number of the route entry is hard-wired in the port memory, and is recorded along with the match. Now, we invalidate this entry, and update the hole pointer to this invalid row in the flash-based (or CMOS shadow) TCAM block. Simultaneously, the deleted route is removed from the appropriate location(s) in the DRAM as well.

In this way, the latency of writes to the flash-based TCAM blocks is hidden, and consistent operation is ensured at the router level. Note that the number of CMOS shadow TCAM blocks is significantly smaller than the number of flash-based TCAM blocks.

4.3.4 TCAM Block Implementation

In existing flash memory designs, flash as well as CMOS transistors are used on the same die [97]. Our work assumes that both flash and CMOS devices are present on the same die.

As discussed in the previous section, the flash-based TCAM consists of $N^2 - 1$ TCAM blocks. Each block (illustrated in Figure 4.2) consists of 256 routing table entries (split into an upper group of 128 entries, and a lower group of 128 entries). In order to minimize wire lengths, the control circuitry (bitline drivers and keepers) are situated between the upper and lower groups of entries, a technique employed in most high-speed memories. The number of TCAM cells per routing table entry was chosen to be 256, which is twice the length of an IPv6 address. By disabling all but 32 TCAM cells, backward compatibility with IPv4 is also supported. The number of port cells per routing table entry is 512. Eight hard-wired bits are used to store the row number of each entry (required during route deletion, as previously described). Other bits can be used to store port (next hop) information, QoS (quality of service) data, etc.

The number of entries per block was arrived at after significant circuit level optimizations, using HSPICE [60]. For the TCAM block design, we generated the layout of the flash-based TCAM and port cells, and based on layout dimensions, extracted accurate 3-D wiring parasitics (R and C) using Raphael [81], for the HSPICE simulations. We held the number of FTCAM cells fixed at 256 to support IPv6 address lookups (with 2× over-provisioning), and varied the number of rows. Beyond 256 rows, the lookup delay (our metric for optimality) gets drastically higher. This is because in the worst case, one port cell has to pull down an entire column in the T-CAM block. We found that 256 rows present a reasonable compromise in size versus speed.

Each TCAM block performs a parallel lookup of the applied destination address among the 256 routing entries stored in it. Because all the routing entries have the same mask length, it is guaranteed that exactly 0 or 1 entry will match the applied destination address. The TCAM block has 256 match lines, with each precharged

match line being pulled down if one or more TCAM cells mismatch the destination address. As a consequence, each match line spans 256 TCAM cells horizontally. However, this results in a fairly long $MATCH$ line, resulting in a large match computation delay for any row. The parasitic resistance and capacitance of the $MATCH$ line are proportional to its length, and therefore its RC time constant increases quadratically with length. In order to minimize this delay the $MATCH$ line is split into p smaller *sections*, as illustrated in Figure 4.3. In this figure, the match line is shown as 32 smaller lines. If any section i determines a mismatch condition, it pulls down its $MATCH_i$ signal, which turns on device M for section $i+1$, which pulls down the $MATCH_{i+1}$ signal for section $i+1$. This effect cascades, until $MATCH_{32}$ is pulled down. There is also a keeper device K in each section, and it serves to speed up the pulldown of the $MATCH_i$ signal once a mismatch condition is detected. In other words, if the $MATCH_q$ signal of section q $(1 < q < 32)$ is pulled low, then sections $r > q$ $(q < r \leq 32)$ are automatically pulled low by the NMOS devices labeled M, in each of the sections with index r.

We performed several SPICE simulation sweeps to determine the optimal number of sections p for the match line, and found this number to be 32 (with 8 TCAM cells per section). For the CMOS TCAM block we use for comparisons, the value of p was found to be 4.

Each TCAM block performs a parallel search on the applied destination address, independent of all other blocks. The pipelining of the operation is illustrated in Figure 4.4. The duration for the TCAM lookup (T_2) is larger than that for port memory lookup (T_1), and the total cycle time $T = T_1 + T_2$ is one of the figures of merit of our design.

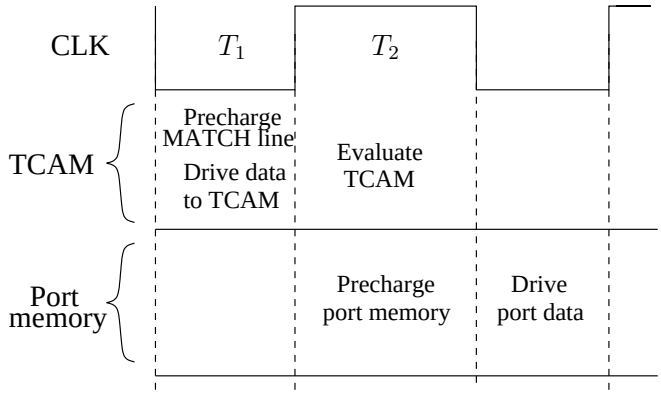


Figure 4.4: Pipelined implementation of lookup functionality.

4.3.4.1 Flash-based TCAM Cell

For a primer on Flash transistor operation the reader is referred to the Appendix.

Our flash-based TCAM cell is illustrated in Figure 4.5. This figure refers to a cluster of four TCAM cells. Cells in row j $(j+1)$ are all connected to the $match(j)$ $(match(j+1))$ match line as shown. Each match line illustrates two TCAM cells connected to it. Each TCAM cell consists of two flash FETs connected in series to the match line. The match line is precharged, and the control gates of the two flash FETs are connected to signals a_i and b_i, respectively.

We now describe the encoding of the signals a_i and b_i that are used to perform a routing table lookup, along with the corresponding threshold voltage settings for the two flash FETs in each TCAM cell. Consider $cell(i,j)$ in Figure 4.5. The control gate of transistor M_1 (M_2) is connected to a_i (b_i).

Consider the Karnaugh-map [100] for the state of M_1 in Figure 4.6 a). When R_H is applied on a_i, transistor M_1 conducts regardless of the threshold voltage of M_1, since $R_H > T_H$ and $R_H > T_L$. When R_L is applied to a_i, M_1 conducts only when the threshold voltage of M_1 is T_L, since $T_H > R_L > T_L$.

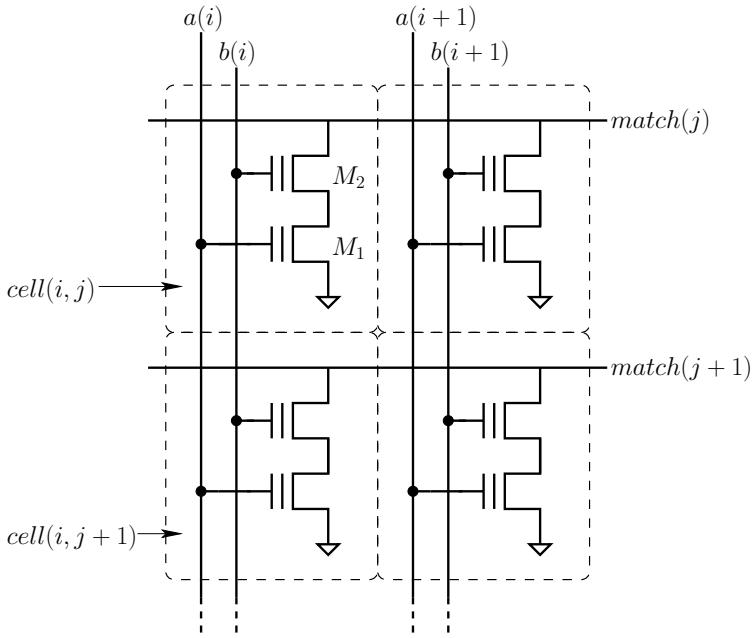


Figure 4.5: Flash-based TCAM cell.

In a similar manner, we construct the Karnaugh-map for the state of M_2 in Figure 4.6 b), as a function of b_i and the threshold voltage of M_2. Note that the rows (columns) of Figure 4.6 b) have values in the reverse order as compared to the rows (columns) of Figure 4.6 a).

Now consider Figure 4.6 c). This figure is the logical intersection of Figures 4.6 a) and b). The row (column) labels of Figure 4.6 c) are the concatenation of the row (column) labels of Figures 4.6 a) and b). In other words, the top left box in Figure 4.6 c) corresponds to the situation when R_L is applied on a_i *and* R_H is applied on b_i, when M_1's threshold value is T_L *and* M_2's threshold value is T_H. The meaning of the other 3 boxes in Figure 4.6 c) can be derived similarly.

Since M_1 and M_2 are connected in series, the entry in any box of Figure 4.6 c) is ON iff the entries in *both* the corresponding boxes of Figures 4.6 a) and b) are ON.

This yields the logic function of the cell state in Figure 4.6 c).

Now let us correlate the above discussion to the TCAM cell operation. Consider the threshold voltage of the $\{M_1, M_2\}$ pair. Assume that a value of $\{T_L, T_H\}$ refers to a "1" value being stored in the TCAM, $\{T_H, T_L\}$ refers to a "0" value being stored in the TCAM and $\{T_H, T_H\}$ refers to a "X" value being stored in the TCAM. Now assume that $\{a_i, b_i\}$ value of $\{R_L, R_H\}$ refers to the "lookup 0" value of the applied destination IP address, and $\{R_H, R_L\}$ refers to the "lookup 1" value of the applied destination IP address.

Before a lookup is performed, the matchline is precharged to (R_L). When a "lookup 0" value is applied to the TCAM cell, the cell state is ON only when a "1" value is stored in the TCAM (thereby pulling down the precharged match line, and declaring a mismatch condition). When the "0" or "X" value is stored in the TCAM, the cell state is OFF, and the match line is not pulled down, as required. Similarly, when a "lookup 1" value is applied to the TCAM cell, the cell state is ON only when a "0" value is stored in the TCAM (thereby pulling down the precharged match line, and declaring a mismatch condition). When the "1" or "X" value is stored in the TCAM, the cell state is OFF, and the match line is not pulled down. This discussion describes the construction of the TCAM cell, and simultaneously serves as a proof-by-construction of correct operation of the TCAM cell.

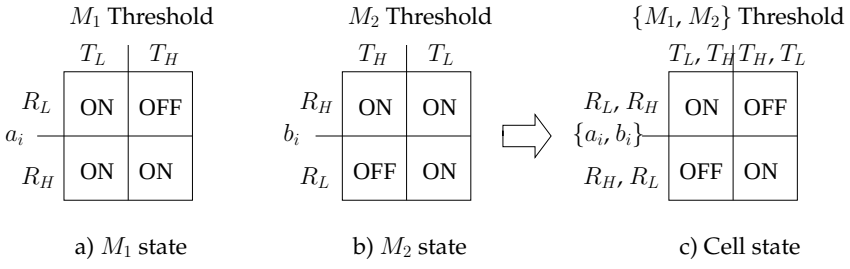


Figure 4.6: TCAM cell logical construction.

Theorem 4.3.1. *The proposed flash-based TCAM cell correctly performs a ternary lookup*

Proof. Follows from the construction of the TCAM cell described above □

The traditional CMOS TCAM cell utilizes 17 transistors (instead of 2 flash transistors in our design) [20]. This CMOS TCAM cell consists of 2 6-T SRAM cells. One SRAM cell stores the prefix bit, and the other SRAM cell stores the mask bit.

4.3.4.2 Flash-based Port Cell

Our flash-based port cell is illustrated in Figure 4.7. This figure refers to a cluster of four port cells. Each port cell consists of a single flash FET. The control gate terminals of the cells in row j ($j+1$) are all connected to the $match(j)$ ($match(j+1)$) match line. Each match line in the figure illustrates two port cells connected to it. Hence, when the match signal of any row is high, the corresponding port cells for that row are read out on the bitlines of the port memory.

The flash FET of each port cell is programmed with one of two threshold voltages T_H or T_L (the same threshold voltages were used for the flash FETs of the TCAM cell) and driven with one of two voltages 0 or R_L depending on whether there is a match (R_L) or no match (0). To perform a read, all bitlines are first precharged to VDD. Now, assume that *match(i)* is driven to a value R_L, indicating that the i^{th} row of the port memory is to be read out. If *cell(i,j)* has a threshold of T_H, the flash FET in this cell will not turn on, and *bitline(i)* will stay precharged. If *cell(i,j)*, on the other hand, has a threshold of T_L, the flash FET in this cell will turn on, and *bitline(i)* will be discharged to ground.

Our HSPICE simulations of the port memory indicated that it was faster than the TCAM cells of the TCAM block, and hence we did not split the bitlines of the port memory into sections. Also, to assist in the discharge of the port memory, each

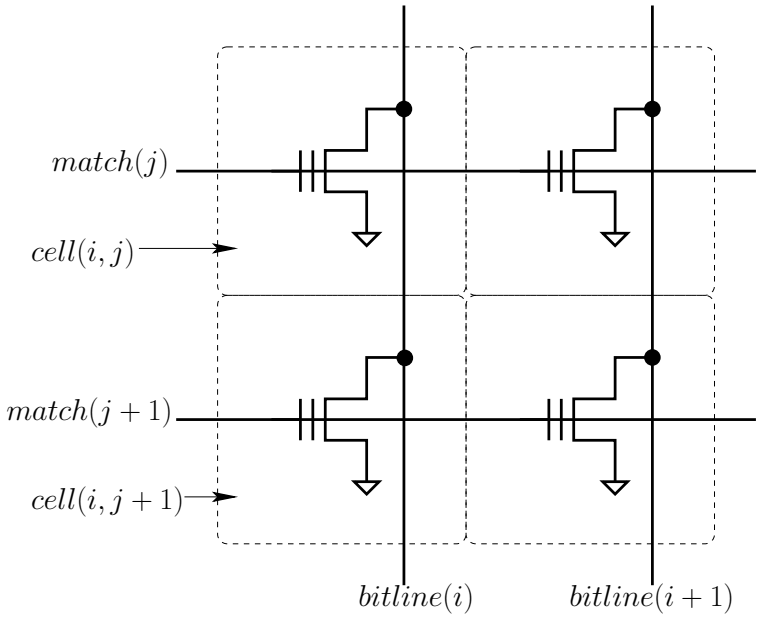


Figure 4.7: Flash-based port cell.

bitline had a single inverter which drove a NMOS keeper transistor. The flash-based port memory does not use a traditional sense amplifier. The structure utilized is identical to the keeper circuit employed in each section of the matchline (shown in Figure 4.3), and is located in the control circuitry bay between rows 128 and 129 of the TCAM block (see Figure 4.2).

The corresponding port cell for the CMOS TCAM block is a standard 6T SRAM cell, and is not shown for brevity. Note that the port cell of our flash-based TCAM block uses just one flash FET, as opposed to 6 CMOS FETs for the CMOS TCAM block.

For the CMOS port memory, sense amplifiers [102] are inserted at the ends of the bitlines, between rows 128 and 129 of each block.

These sense amplifiers are used to sense the value driven out of the SRAM during a read operation, and reduce the delay of the read operation. The sense amplifier

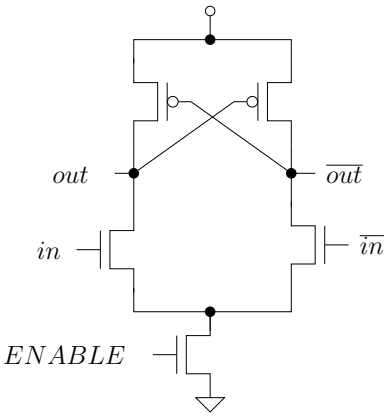


Figure 4.8: Sense amplifier used in CMOS port array.

used is shown in Figure 4.8. The *in* and $\overline{in}$ terminals of the sense amplifier are connected to *bitline* and $\overline{bitline}$ of the port memory array respectively. The bitlines of the CMOS port memory are precharged to VDD/2 before a read.

We refer the readers to the Appendix for erase and program operations on the proposed Flash-based cells.

4.4 Evaluation

	TCAM cells			Port memory			Total		
	Delay	Power	Area	Delay	Power	Area	Delay	Power	Area
CMOS TCAM block	218 ps	96 mW	127402.0 μ^2	174 ps	33 mW	159252.5 μ^2	393 ps	129 mW	286654.5 μ^2
FTCAM block	679 ps	65.2 mW	19398.6 μ^2	306 ps	14.1 mW	16731.3 μ^2	985 ps	79.3 mW	36129.9 μ^2
Ratio (Flash-based/CMOS)							2.51×	0.61×	0.126×

Table 4.1: Comparing delay, area and power of CMOS TCAM and FTCAM blocks.

We implemented the FTCAM block in HSPICE, using accurate resistive and capacitive parasitics obtained from a 3-D parasitic extraction tool, Raphael [81]. A 45nm process was used for our simulations. We started by generating a layout of the TCAM and port cells, and used these layouts to estimate the wire lengths and

spacing for the bitlines and matchlines, which were then fed into Raphael for 3-D parasitic extraction.

For CMOS devices, we used a 45nm PTM process [70], while for flash devices, we derived our model card from the device-level measurements presented in [47, 1]. The basic idea is to emulate the states of a floating-gate device with two separate PTM model cards, one that models the flash FET in the low V_T state (we call this value T_L), and the other for the flash FET in the high V_T state (we call this value T_H). We used the gate and oxide thicknesses, and doping levels from [47, 1]. We then took a base 45nm PTM CMOS model card and modified it so that the threshold voltages of the two derived model cards would be T_H and T_L, respectively, and the I_{ds}-V_{gs} curve slopes matched in [47, 1].

In our experiments, VDD was set to 1.1V, and T_H and T_L were 760 mV and 210 mV respectively. Our R_H was VDD, while R_L was selected to be 600 mV. The values of R_L, T_H and T_L were chosen based on several HSPICE sweeps which aimed at minimizing TCAM lookup delay primarily, and overall power consumption secondarily.

The number of sections per match line was chosen to be 32. The output of the 32^{nd} section was buffered, registered and then driven to the port memory array with a buffer chain. The port memory bitlines were not split into sections. The bitlines of the port memory, as well as the match lines of the TCAM cells were precharged to VDD.

The FTCAM block occupied an area of 190.72 μ $\times$ 187.44 μ. About 46% of this area was used by the port memory. We refer the reader to the Appendix for flash-based TCAM and port cell layouts.

For the CMOS TCAM block, we used the same specifications as for the FTCAM block (256 entries, 256-bit wide TCAM entries and 512 bit wide port entries). We

implemented the CMOS TCAM block in HSPICE, with resistive and capacitive parasitics obtained from Raphael [81]. A 45nm PTM process [70] was used for our simulations. The cell layouts for the TCAM and SRAM blocks were obtained from [20], and were scaled to obtain an estimate of the wire lengths for the matchlines and bitlines. These wire dimensions were used for 3-D parasitic extraction for the HSPICE simulations of the CMOS TCAM block.

The number of sections per match line was chosen to be 4, based on HSPICE experiments which were aimed at minimizing TCAM lookup delay. The output of the fourth section was buffered, registered and then driven to the port memory array with a buffer chain. The match lines of the TCAM cells were precharged to VDD. The port memory transitions were sped up by using a sense amplifier for each port memory cell. The port memory bitlines were not split into sections. The bitlines of the port memory were precharged to VDD/2 and equilibrated before the port memory read operation.

The CMOS TCAM block occupied an area of 311.04 μ × 921.6 μ, with ∼55% used by the port memory. These numbers were obtained from the layouts in [20], after scaling to a 45nm fabrication process.

The results of our comparison of the FTCAM block with the CMOS TCAM block are presented in Table 4.1. We note that our FTCAM block occupies about 8× less area than the CMOS TCAM block, with a power reduction of about 1.64×. The speed of the FTCAM block is about 2.5× lower than the CMOS TCAM block, but it can sustain the line rates of the fastest current routers in the internet core, as we will see next. The FTCAM cells delay (679 ps) shown in the table constitutes the critical part of the total delay, since it is 69% of the total (985 ps).

The increase in FTCAM block delay compared to a CMOS TCAM does not pose a problem at the system level since our FTCAM can support ∼400 Gb/s link speeds,

as we show next. The current backbone routers need to operate at ∼100 Gb/s. We conclude that, despite the 2.5× larger delay, the proposed design is fast enough to be utilized in modern systems.

To compute the highest serial line rate supported by our TCAM, we assume a 48 byte packet, and that the FTCAM operates at ∼1 GHz clock speed (based on the total delay of 985 ps from Table 4.1). The system is able to perform 1 billion lookups per second, which translates to a $10^9 \times 48 \times 8 = 384$ Gb/sec link speed. Further, if the FTCAM lookups were pipelined, a clock speed of ∼1.4 GHz is achievable (based on the cycle time of 679 ps, the larger of the FTCAM cells and the Port memory delays in Table 4.1), allowing a $1.4 \times 10^9 \times 48 \times 8 = 537$ Gb/sec link speed.

Thus, our FTCAM supports link speeds of 384–537 Gb/sec (which is an undefined standard as of yet). This allows our FTCAM to do about 10–13 OC-768 (whose data rate is 38.4 Gb/sec) lookups in a clock cycle.

We also computed the sensitivity of the delay and power to variations in the T_H and T_L values. Both T_H and T_L were varied by ± 20 mV independently, to simulate variations in the floating gate charge that occur during the flash programming step. With these variations in T_H and T_L, the delay numbers reduced (increased) by 2.8% (3.8%). The power consumption varied by ± 0.2%.

4.4.1 Lifetime Estimation

We estimate the flash-based part of our chip to have the area of 1.5cm×1.5cm. This allows us to have 6000 FTCAM blocks, for a total FTCAM size of 1.5 million entries. We assume that there are 32 + 15 CMOS shadow blocks (32 for each prefix, and 15 for double/triple buffering). The area of shadow CMOS blocks is estimated at 0.52cm×0.52cm, using the area numbers reported in Table 4.1. (By increasing the CMOS shadow block capacity further, it is possible to boost the FTCAM lifetime,

but at the expense of the total chip area.) Hence the total chip area is about 4cm^2, including the area of the control circuitry. The interconnect is accounted for by arranging extra chip area to realize wiring. We assume that 37% of the total die area (4 cm^2) is used by the interconnect and control logic. There is no performance impact of the interconnect, since the pipelined nature of TCAM lookup and Port memory lookup allows us to hide the interconnect delay.

We conducted an experiment to estimate the lifetime of our TCAM under real workloads. For this purpose, we developed and used an in-house TCAM-based router simulator based on the architecture described in Section 4.3. The contents of the routing table were populated using the Routing Information Base (RIB) snapshots [83] of a real internet router on July 1, 2014. The size of the internet routing table was about 500K entries. We "replayed" the UPDATE traces for one full day following the RIB dump and recorded the number of writes to each FTCAM block. We also tracked the CMOS shadow block utilization.

Figure 4.9 shows the number of flash entries that are used over the duration of one day, as a function of their prefix length. "Base Size" refers to the number of entries in the FTCAM at the start of the day. "UPDATES" refers to the number of entries updated (with and without shadow CMOS blocks). Prefixes with less than 2000 entries are not shown.

The results show that CMOS shadow blocks (without double/ triple buffering) filter about 61% of all the UPDATEs to the FTCAM, which increases its lifetime. There were 45 cases when a CMOS shadow block was flushed twice within one second, and for the prefix length of 24, there were two cases when the CMOS shadow block was flushed 7 times in one second. The average time between consecutive flushes was 290 seconds. We estimate the erase/write delay of the FTCAM blocks to be similar to the traditional flash storage, meaning that they would support up to 3-5 writes

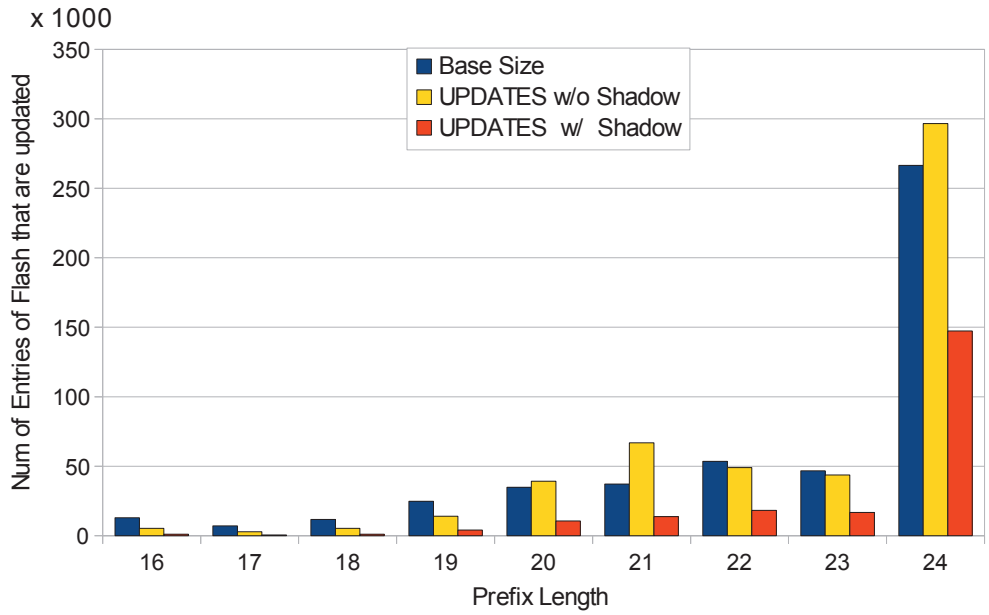


Figure 4.9: FTCAM utilization over one day (with and without CMOS shadow blocks).

per second [48]. Thus, in case we did not use double buffering, prefix 24 would not be possible to update correctly. Hence we use double/triple buffering.

Since some prefixes are written more frequently than 3 times a second (prefix 24 in our example), we need a pool of extra CMOS shadow blocks to double-buffer (or triple-buffer) the UPDATEs to such prefixes. From our simulation, the total number of CMOS shadow blocks required is about 32 + 15 (as explained before). With this choice, no UPDATEs are lost. The minimum number of shadow blocks necessary to sustain the FTCAM operation from our simulation was 32 + 5. With less than 37 shadow blocks the system experienced packet loss due to insufficient flash write bandwidth. We over-provision the double-buffering blocks by 3× (with a total of 32 + 15 shadow blocks) to provide a safety margin.

The observed number of UPDATEs to FTCAM blocks was 210K per day (out of a total of 500K per day). To estimate the lifetime of the proposed scheme, we

made the following assumptions: (1) The FTCAM has 1.5M entries, with 500K of them occupied by the routing table. (2) In the worst case, rewriting each entry needs to erase and copy the whole 256-entry block. (3) Flash endurance is 100K write cycles [7]. (4) Randomized wear leveling techniques are used in flash-based blocks [3, 74].

In our FTCAM, 500K out of 1.5M entries are occupied. The remaining 1M entries correspond to 4000 FTCAM blocks of 256 entries. The FTCAM experiences 210K UPDATEs, which in the worst case requires 210K block writes per day. Assume any block out of the pool of unoccupied blocks can be used for (uniform) wear leveling, thus the free blocks will be fully overwritten $\frac{210K}{4000} = 52.5$ times a day. With the flash endurance of 100K rewrite cycles, the FTCAM lifetime is therefore $\frac{100K}{52.5} = 5.2$ years until wear-out. If we now assume that FTCAM blocks are statically allocated for the life of a router, it is easy to notice (see Figure 4.9) that the prefix 24 will be the most stressed. Hence it is reasonable to allocate more entries to that prefix. If we were to allocate 500K free entries (2000 blocks) to prefix 24 (i.e. about 750K entries total), then the estimated lifetime would be computed as follows. For prefix 24, there are 150K UPDATEs per day (see Figure 4.9), requiring 150K block writes per day in the worst case. There are 2000 free blocks for prefix 24. Hence the free blocks will be overwritten $\frac{150K}{2000} = 75$ times a day. Assuming a flash endurance of 100K cycles, the FTCAM lifetime can be estimated as $\frac{100K}{75} = 3.65$ years until wear-out. By adding uncommitted spare replacement FTCAM blocks, this lifetime can be further increased.

4.5 Summary

Ternary Content-addressable Memories are commonly used for high-speed IP packet routing applications in the internet core. We present a flash transistor based

design of a TCAM block. Our FTCAM cell utilizes only 2 flash transistors, while our flash-based port memory cell utilizes a single flash transistor. In comparison, the traditional CMOS TCAM cell requires 17 transistors, while the CMOS port memory cell requires 6 transistors. Based on our layout and circuit simulation experiments, we conclude that our FTCAM block achieves an area improvement of 7.9× and a power improvement of 1.64× compared to a CMOS approach. The estimated lifetime of such FTCAM implementation is about 5 years. The speeds accomplished by our flash-based TCAM can meet current (∼400 Gb/s) data rates that are encountered in the internet core.

5. OSVPP: OS VIRTUAL-MEMORY PAGE PREFETCHING

5.1 Introduction

Modern computing trends are increasingly turning towards Big Data, Cloud Computing, and Virtualized Systems. Tremendous computation and storage power is required to support the existing applications, with the demand growing as the use of virtualization is expanding. With the backing store orders of magnitude slower than the Main Memory, applications demand more DRAM in order to execute completely inside the memory, avoiding the costly disk or network reads and writes. Several applications at Facebook, for example, run in memory, using such tools as memcached [64]. Main Memory capacity has become a major bottleneck.

Conventional data center architectures rely on the use of DRAM, which is costly and wastes energy [23]. We aim to reduce the costs and energy expenditure by extending the expensive DRAM with a cheaper alternative, such as NVMe Flash [104].

Historically, storage devices used to be much slower than DRAM and thus only suitable for longer-term storage (such as a replacement for the spinning disks). On the other hand, the modern Non-Volatile Memory technologies (NVM) [59] have significantly improved performance and are now steadily approaching the speed of DRAM. However, only NVMe Flash devices are currently widely available in large capacities, and their latencies are still several orders of magnitude higher than the traditional Main Memory. We note that naively replacing a portion of DRAM with such NVMe devices will, while reducing the system cost, lead to a significant performance loss which will be unacceptable.A mechanism is thus needed to actively manage such DRAM + NVM combinations in order to successfully lower the costs but keep the performance at a competitive level.

In current systems, when data does not fit in the main memory, it is swapped out to a storage device behind the memory. OS manages the DRAM and the storage device to create the impression of a larger memory. However, the application performance is significantly impacted by the slower storage devices. OS in current systems employs "demand paging" to bring pages from the swap device to memory on a "page fault". With emerging NVM devices, the performance gap between main memory and the swap devices will be much smaller, potentially enabling other approaches to managing the swap space. Second, the emerging NVM devices exhibit much higher parallelism than magnetic disks. These characteristics warrant revisiting the problem of managing the swap devices behind the main memory. We propose a prefetching approach, i.e. a scheme that predicts the application's demand for certain memory addresses and actively fetches those addresses into the faster memory in the background, so the application is operating as if the whole memory were fast DRAM while in fact only a portion of the memory is DRAM.

We chose to base our approach on the OS page swapping mechanism. There are several considerations for this:

- The swapping is simple, i.e. the support is already built-in to the hardware and software, thus there is no implementation penalty and the approach can be quickly adopted by the industry;
- The OS already manages the page loading and unloading from the swap device, hence no need to come up with a custom system that would be compatible and portable;
- However, the conventional OS mechanisms are not proactive enough, and are based on old assumptions about slow spinning disks and small application working sets.

To the best of our knowledge, this is the first work to address the OS Page prefetching with low-latency NVM storage. The main contributions of this work are as follows:

- We implemented a framework in the Linux kernel for swap page prefetching,
- We augmented the Linux RAMDisk driver to emulate high-speed low-latency backing store NVM devices,
- We proposed a collection of approaches to page prediction, based on the observed Page Fault patterns, without any extra overhead (i.e. the predictors themselves are not increasing the number of Page Faults), and evaluated them on a real system,
- We built a mechanism of ranking the prediction algorithms on-the-fly and choosing a combination thereof for the best performance.

Our approach reduces the application running time by 18% on average, compared to the conventional OS swapping. We are able to achieve 50% of the maximum potential speedup.

5.2 Background

Virtual Memory (VM) is a powerful abstraction used in practically every modern Operating System (OS) [12]. It is attractive from many points, but some of the main advantages are:

- VM allows dynamic and efficient sharing of Main Memory among applications in the system;
- By abstracting Main Memory and Backing Storage (hard disk) capacity, VM

creates an illusion of "infinite main memory" available to each application[1] (see Figure 5.1), while the Physical Main Memory capacity may be several orders of magnitude lower;

- Since applications generally use only a fraction of their full working set at any given time, VM allows more applications to run in the system without adding the Main Memory capacity to the machine;

- VM guarantees protection for the applications' memory contents from each other. Each application is provided with a contiguous block of "virtual" memory addresses and thus it cannot access the addresses of other applications.

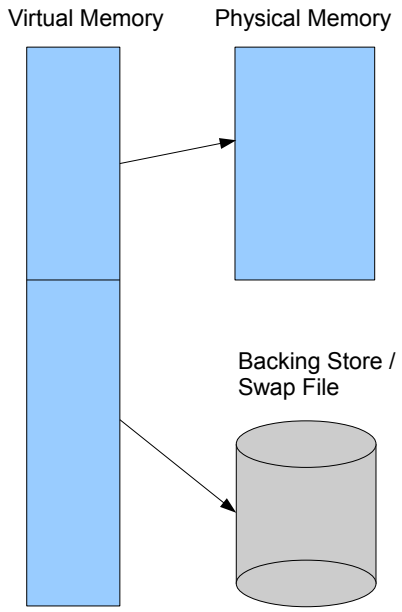


Figure 5.1: A diagram explaining the virtual memory concept.

[1]In the current x86-64 CPUs, the total amount of available virtual memory is limited by the implementation at $2^{48} = 256TB$.

Here we are going to briefly explain the operation of Virtual Memory at a basic level necessary for understanding the following discussion. For more detailed information the reader is referred to the Linux Kernel books [8, 56]. In x86 systems family, Virtual Memory is implemented as a hardware-software system where the OS manages the Virtual-to-Physical address mapping for each distinct application, while the CPU is responsible for actually translating Virtual Memory addresses into Physical Main Memory addresses. The granularity of such mappings is 4 kB and in x86 each such region is called a Page. The interface between the OS and the CPU is provided through a Page Table (PT) and is standardized in x86 systems to enable the hardware page table walker in the CPU core to traverse the entries and get virtual address translations without the OS intervention. The PT is indexed with the 36 high-order bits of the Virtual Address (excluding the lowest 12 bits as they index within the 4 kB page itself). When the OS needs to establish a relationship between a given Physical memory page and its Virtual address in VM, it sets up an Entry in the PT to contain a translation to the Physical address in question. The CPU then performs the PT lookup for every memory operation, and if there is no write protection violation and the translation is valid, the operation is carried on. Conversely, in the case of a violation, a Page Fault is triggered and the OS is responsible to deal with the fault, whether by terminating the application or allocating a translation, respectively.

The mechanism by which the Hard Disk capacity is abstracted in VM is called Page Swapping (or just Swapping). The OS maintains up to 32 swap regions, or "swap files" where the memory pages can be stored. When swapping, a 4 kB page is written (swapped out) from Physical Memory to the swap file in Backing Store, freeing space for some other Virtual Page. Note that the new Virtual Page will now have the same Physical address as the swapped-out one. The original Virtual

Page is not lost, however. Rather, the application still has full access to it. The OS makes sure to catalog into the PT entry, the swapped page's location in the backing store instead of the previous physical memory address. The bet is that this freshly swapped-out page will not be needed in the near future. If the bet is wrong and the application tries to operate on the swapped-out page, it has to be read back from the disk and into the Physical Memory. The CPU will try to translate the Virtual Address, but upon reading the relevant PT entry will discover that the page is not in Physical Memory anymore, and the Page Fault will be triggered. The OS will once again check the PT entry, find out where and in which swap file the page is residing, then it swap the page back in to the physical memory, and update the PT entry with the new translation. (Note that this time the page's Physical Address will most definitely be different than previous address, which is now occupied by another page, and it will be reflected in the appropriate PT entry.)

We are thus starting to see the main drawback of this capacity abstraction, namely, page Access Latency degradation: an access that usually takes nanoseconds (assuming data is in the cache), now costs several milliseconds, in the worst case a 10 orders of magnitude increase (if a page is only accessed once after being swapped in). Even amortized across 4K accesses to the page (accessing every byte as a separate transaction), the effective latency of each access is still 1–2 orders of magnitude worse than if the page were in the Main Memory in the first place.

To control this effect, the OS tries to be smart when choosing which pages to swap out as well as swap in. All the pages in the system are accounted for in a type of an ordered list which in theory gives the OS information about the frequency and recency of page usage and allows it to pick the least recently/frequently used pages for swap-out. In Linux, there are two lists, the Active List and the Inactive List which represent, respectively, a collection of "more-recently used" pages and

"less-recently used" ones. The OS periodically samples the hardware flag within each page to determine whether it was recently accessed, for the purpose of moving pages between the two lists. There is no effort made at keeping the relative recency information within each list, other than the fact that pages from the Inactive list, if touched again, are placed at the top of the Active list.

Conversely, when swapping-in, if the OS is able to bring several extra pages together with the demanded page back into the Main Memory, and if those pages happen to be required by the application, the extra latency is mitigated. This mechanism is called Swap Readahead in the Linux Kernel.

Overall, it is reasonable to admit that the conventional spinning disks are so slow that VM Swapping when Main Memory is severely overcommitted is more of a crutch, and a guarantee that the system will drag along instead of completely crashing due to memory exhaustion, and the question remains whether the former is more beneficial for the operator.

5.3 Motivation

In order to lower the cost of Main Memory in systems with huge memory demand, it is becoming practical to substitute less-expensive NVM for some of the DRAM in the system, for a much lower system cost. The main challenge with NVM, however, is its relatively high access latency. We want to give applications the impression of having huge amounts of DRAM while seamlessly swapping pages back and forth between DRAM and NVM, with minimal performance degradation if possible.

In the case of Virtual Memory Page swapping, the application performance is largely driven by a combination of two factors:

- the number of Page Faults (which lead to swap-ins from the storage device), and

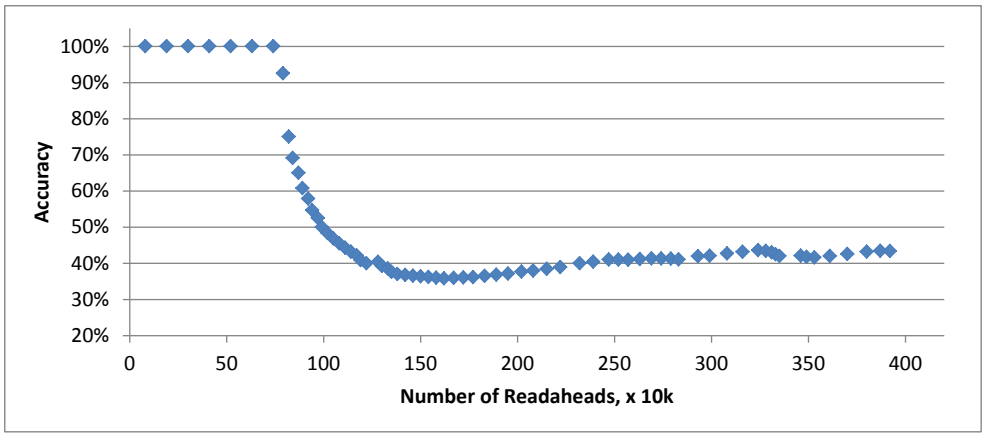


Figure 5.2: OS readahead accuracy with one application running, tunkrank app.

- the penalty, or cost, of each Page Fault (which is the latency of a swap-in operation).

Additionally, any overhead from the prefetching algorithm and the swap pressure from the speculative pages should also be taken into account.

These factors must be carefully considered together in order to maximize the performance in the presence of Page swapping due to insufficient RAM in the system.

Consider the OS Readahead mechanism. By blocking the faulting application and scheduling a number of speculative swap-ins on every Major Page Fault, it guarantees that the number of further Page Faults is significantly decreased. This is assuming that the utilization of those speculative pages is reasonably high, which is the case when only a few applications (i.e. 1–2) are running in the system, as our experiments show. Figure 5.2 depicts the running accuracy (i.e. how many of the read-ahead pages were useful) of the OS Readahead scheme with the Tunkrank application. At the beginning (first ∼900 k readaheads) the application exhibits linear pattern in page faults and the readahead is 100% accurate. However, as the

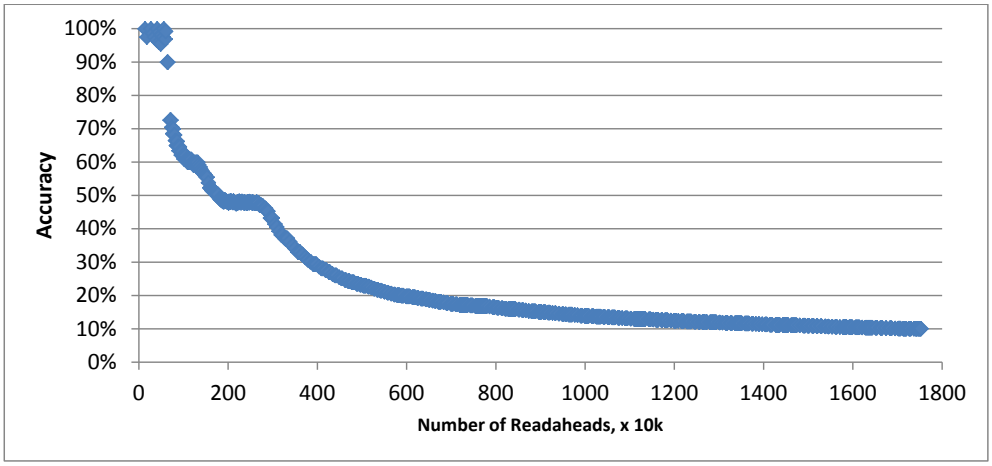


Figure 5.3: OS readahead accuracy with three applications running in parallel, tunkrank apps.

application continues running, the prediction accuracy of the readahead scheme drops to 40–45% on average.

On the contrary, as shown in Figure 5.3, with three or more applications, the OS mechanism is not only inaccurate (we observed down to 10% utilization on the read-ahead pages), but also, by filling the RAM with wasteful pages it increases the swapping activity, which in turn leads to the rising latencies of swap requests. We also note from the Figure that the total number of read-ahead pages is 4.5× greater as compared to the single application case, due to the increased number of faults.

This inaccuracy can be explained if we refer to the mechanism of page swapping in the kernel. With many applications in the system, the LRU lists will contain a thorough mix of pages from every application, so the adjacent pages in the swap file now may not belong to the same application, and thus the temporal relationship between such pages will generally not hold anymore.

In both of the above cases, the pressure on the swap device request queue is

artificially increased. While this was not an issue with traditional spinning disks, as we will show it can become quite severe when NVM devices are used as swap devices. In short, if there are 7 readahead requests ahead of the demand request, the effective swap-in latency observed by this request would be up to 7× the NVM device latency. Referring back to the performance as a product of the number of faults and their latencies, it is easy to see that just to break even vs. no extra swap-ins on Page Faults, the Readahead mechanism has to eliminate at least 85% of Faults. This is not taking into account the overhead that the mechanism introduces by blocking the application while it is scheduling those extra swap-ins.

With the observed OS Readahead prefetch accuracies of 35–50%, it is easy to see that this approach limits the potential performance achievable when using NVM swap devices. Our goal then is to attack both problems simultaneously, i.e. implement algorithms accurate enough to reduce the number of Faults, and to build up a smart framework that would allow us to carefully inject speculative requests so as to not disturb the demand requests and thus to keep their effective latency low.

5.4 Design

In this section we describe the considerations for OS Page prefetching and provide high-level design explanation. In order to improve the performance, the prefetches must be accurate, timely, and must incur minimal interference with the Demand requests.

5.4.1 To Prefetch or Not?

Unlike CPU caches where the block size is quite small (typically 64 Bytes [27, 63]) and thus temporal and spatial locality patterns are more apparent, the OS page sizes (4 kB pages are common) and allocation policies (on-demand, controlled by the OS memory allocator) make it challenging to detect and exploit locality. Pages adjacent

in virtual memory, when swapped out, may end up totally separate on the storage device. Moreover, pages that have been accessed together at some point in time may not end up swapped out together (e.g. if one of the pages were accessed shortly before the swap-out was triggered). This is particularly true in situations when multiple applications are running in the system, exerting high memory pressure.

As hinted before, in swapping situations, the conventional OS readahead mechanism in many cases leads to poor performance. It is even inferior to an approach of not prefetching at all. However, as we will show next, by adapting to the peculiarities of NVM devices and better exploiting macroscopic locality patterns in page accesses, it is possible to achieve a solid performance boost when the system is under memory pressure and swapping.

5.4.2 Prefetch Support for NVM Devices

In this work we focus mostly on swap-ins (reading pages from the swap device back into RAM). The Linux OS page swapping mechanism is developed with the assumption that spinning disk is used as a swap device, and while this is of little consequence to page swap-outs on an NVM device (since its performance far surpasses that of the traditional disk), page swap-in performance of this mechanism can be negatively impacted by such optimizations.

The three critical assumptions made in the Kernel are as follows:

- The latency of a swap-in request is as large as several milliseconds (driven by the spinning disk access times);
- For a penalty of one seek latency, multiple pages can be read from the disk with minimal extra overhead;
- The random-read performance is so low that it only makes sense to fetch se-

quential blocks of pages from the storage device.

Based on these assumptions, the OS implements a simple, effective readahead mechanism. On a demand page fault, a whole block of pages (typically 8 pages, but it can be adjusted via the sysctl mechanism during runtime) is requested from the disk. OS only requests aligned blocks, which means the demand page is not necessarily the first one to be read from the disk. This is done in the context of the original application. The application is suspended until the main demand page has been read, then it may continue. The extra pages are placed in the Swap Cache (i.e. readahead pages are not installed into the application's address space until there is a demand for them), and may even be swapped-out again without a chance to be used, if there is high memory pressure or of the readahead was inaccurate.

Obviously, with NVMe devices, none of the above assumptions is true. The read latencies of such devices are in low 100 μs [104], and if multiple pages are to be read, the latency penalty is added for each page. Additionally, due to their architecture, NVM devices perform as well on random reads as on sequential reads. It is easy to see how the latency for the demand request can exceed 500 μs if it happens to fall toward the end of an aligned block. When memory pressure is high, certain memory-intensive commercial applications will experience a large number of page faults (which means very short times between page faults – we observed as low as 100 μs). Note that the readahead requests from previous demand fault will still be residing in the device queue, and the effective latency for the next demand request could grow even further. With such implications, simply disabling the OS readahead might yield a considerably better application performance due to much lower effective demand fault latencies (our experiments show up to 6% performance improvement with readahead disabled, and $\sim$1% average improvement – see Figure 5.7).

It is thus critical that the readahead mechanism be modified with the view of NVM swap devices. The following fundamental changes are necessary:

- Utilizing a NOOP request scheduler. By default, Linux sorts the requests and uses the elevator mechanism to optimize the spinning disk performance. As discussed above, this can lead to increased latencies seen by demand requests. NOOP scheduler simply executes the requests in the order they arrive, allowing to retrieve data in the order intended by the prefetching mechanism.

- Critical-Page-First (CPF) demand page fetching. Instead of requesting the whole aligned block from the device, we first request the demand page and let the application continue as fast as possible.

- Separating readahead/prefetch logic from demand fetch logic. Since scheduling the read requests may easily take dozens of microseconds, it is necessary to perform them outside of the faulting application's context. We are using a different CPU core in this work, however, a hardware approach can be used.

- In order to keep the demand request latency as low as possible, we maintain two separate queues, one for demand requests and one for prefetches, and give higher priority to the demand queue. We schedule the prefetches only when there are no demand requests waiting.

- Prioritizing demand requests. The incoming demand request "jumps ahead" of any prefetch/readahead requests in the queue.

5.4.3 When to Prefetch

The OS Readahead mechanism injects 8 pages (the demand page plus 7 readahead pages) into the swap device queue on every Page Fault. This works well if the time

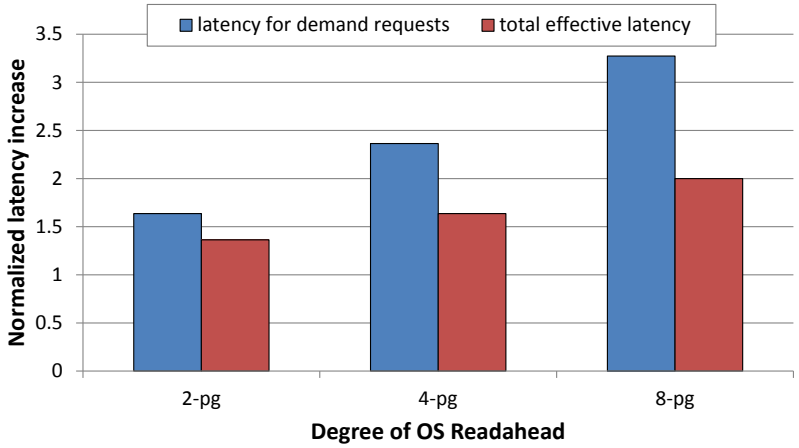


Figure 5.4: OS readahead latencies, normalized to no-readahead, on an emulated NVMe device.

to fetch the pages is shorter than the time between faults. In other words, this only works when the memory pressure is low and not much swapping activity is going on in the system. Lower-latency devices will help, but only to an extent of tolerating somewhat higher swapping activity. It is easy to note that the readahead approach may lead to a backlog in the swap device queue, and in turn cause the Page Fault latencies to increase. Figure 5.4 presents a comparison of demand page fault latencies and total effective request latencies between a baseline system with no readahead (i.e. where only the faulted page is swapped-in) and a system with various degrees of readahead (varied from 1 extra page to 7 extra pages per fault). For no-readahead case, these latencies are essentially the same as the device latency. We observe that the more speculative pages fetched per page fault, the higher the effective fault latency, which in turn affects the application running time. Schemes with very high prediction accuracy might benefit from injecting prefetches directly into the device queue, but as we show in Section 5.3, the OS readahead accuracy is on the order of 40%.

All is not lost, however. Assuming the 40% utilization rates of the readahead swap-ins (see Figure 5.2, assuming we can isolate the applications' pages into separate swap files), we simply need to find a way to keep the demand request latencies low, and the total latency will automatically be improved. (Recall that if a readahead request is accurate and completed in time, the latency of a potential demand fault for the given page is reduced to zero.)

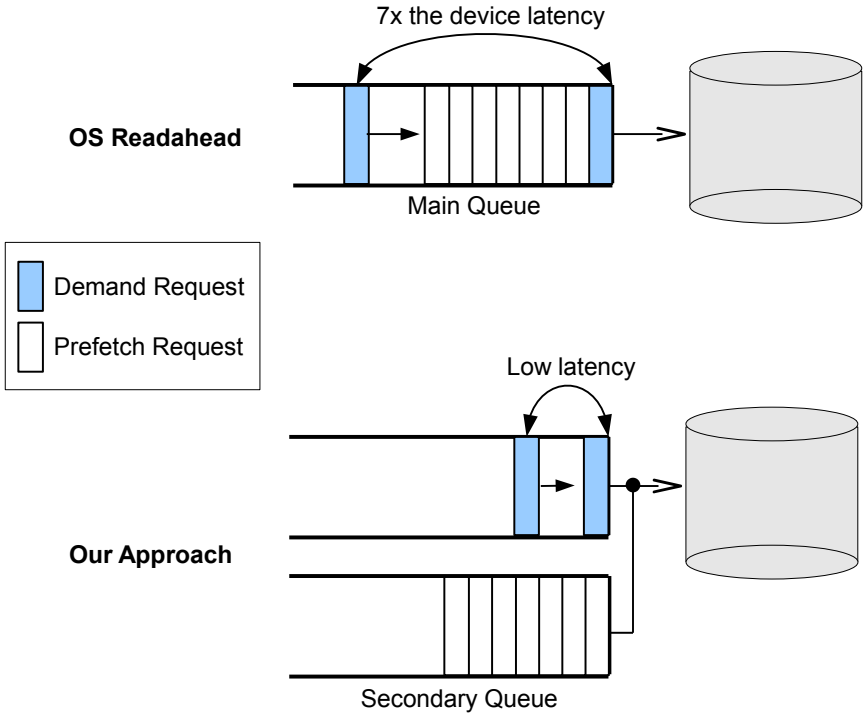


Figure 5.5: Secondary queue for the prefetch requests.

From Figure 5.4, the lowest latency for demand requests is achieved when there is no readahead pressure on the device. We note, however, that the application page fault behavior is bursty, with periods of high rates of page faults followed by phases

of relatively lower rates. The key observation here is that during the low phases we can issue more speculative prefetch requests, and back off of prefetching during the higher-demand phases. The best way to achieve this is by prioritizing the demand requests with respect to the prefetches. We implement a "secondary queue" to hold the prefetch requests, and the driver schedules requests from this queue only if there are no demand pages waiting in the main swap queue. Figure 5.5 illustrates the concept. Note how in the conventional OS approach the incoming demand request is forced to wait for all the prefetches to finish, which increases its effective latency, whereas with the secondary queue, this latency is minimal.

Now, the Linux OS prefetches speculative pages together with the demand page. Any new faults and prefetches have to wait until the older ones are executed. In other words, such policy favors old prefetches instead of the recent ones. However, with two separate queues for demand and speculative requests, we have a choice as to which prefetches to prioritize, based on which ones we believe have greater utility to the application. Intuitively, the prefetch requests from the past will be more out-of-sync with what the application is doing, than the fresh prefetches. It thus makes more sense to give priority to the new prefetches instead of the older ones. This is achieved by using a LIFO (last-in-first-out) request management policy instead of the FIFO (first-in-first-out) utilized by the conventional readahead.

Note that in Linux, before a swap request is enqueued, it has to be fully prepared for the swapped-in page. Namely, a fresh free Physical Page frame must be allocated and locked, the relationship between the swap file offset and the physical page established, and the page must be inserted into the OS Swap Cache. The enqueued request may be further decomposed and/or merged with adjacent requests by the I/O scheduler. Thus we are paying a penalty for the requests we may never issue! Because prefetches are speculative in nature, they require a means of reorder-

ing and/or canceling, or "rolling-back". It is easy to see how such roll-back makes canceled prefetches twice as expensive. In our experiments we have thus adopted a secondary queue which is one level higher than the main queue. I.e. the entities in the secondary queue are hints to the fetch mechanism, and are not allocated physical memory until the moment they are actually issued to the swap device. This allows nearly "free" roll-back (by simply erasing the entry in question), and only the actual prefetches get through the process of allocating physical memory.

In summary, to keep the Demand fault latency low, any demand requests must be fetched before the prefetches. The recent faults are more important than the past ones, consequently, priority among the prefetches must be given to the prefetch requests induced by the recent demand faults. We fulfill both these requirements by implementing a secondary queue for prefetch requests, with LIFO ordering.

5.4.4 What to Prefetch

A good prefetch algorithm should strive to predict pages that are soon to be utilized by the application, based on the recent history of page accesses. (Note that the prediction is somewhat complicated in that we can only observe the page faults, not the successful page accesses, without significant overhead. This is different from, say, CPU cache prefetching, where the full stream of accesses is visible to the prefetcher.)

Traditionally, there have been two approaches to such prediction: one based on temporal locality of accesses and the other one based on spatial locality. For our purposes, two pages are said to have temporal locality if, having been accessed together in the past, when in the future the first page is faulted, the second one is also faulted. Conversely, spatial locality implies that if a page is faulted, then a page adjacent to it in the virtual address space will also be faulted.

5.4.4.1 OS Readahead

Since the OS mechanisms are based on the assumption of the spinning disk as the main swap device which favors reading large contiguous chunks, it is only logical for the OS to exploit the physical space locality in the swapfile, in its readahead mechanisms. The physical locality in the swapfile is based on the temporal locality in evictions of pages from memory. The inactive pages are swapped out in larger blocks, moreover, the OS keeps the list of memory pages in an approximate LRU order which is known to capture temporal behavior.

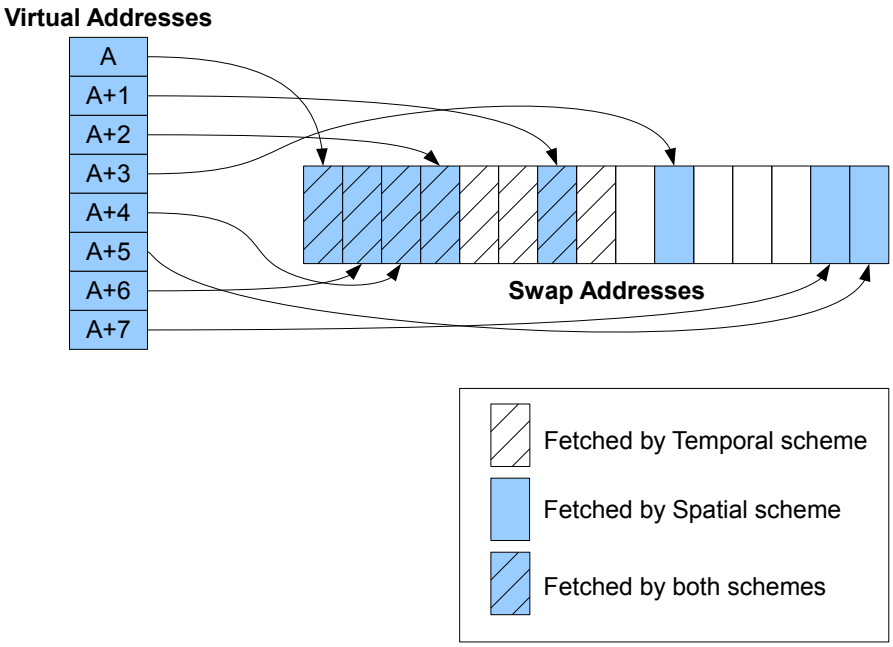


Figure 5.6: Temporal and spatial schemes illustration.

5.4.4.2 Temporal Prefetch

Based on the observation that the pages adjacent in the swap file were swapped out together, and thus were likely used together at some point in the past, it is reasonable to bet that they are likely to be again demanded together in the future. Our temporal approach thus prefetches pages surrounding the current faulting page in the swap file. The two main distinctions from Readahead mechanism are that our temporal prefetches have lower priority than the demand page faults, and the more recent temporal predictions have priority over the older ones. This reduces the latency of the demand requests, as well as avoids the potentially inaccurate stale prefetch requests from getting executed. Refer to Figure 5.6 for an illustration.

5.4.4.3 Spatial Prefetch

We tested a spatial approach which predicts that certain consecutive page addresses will be faulted following the current page fault. Figure 5.6 illustrates the concept. In the figure, following a page fault to address A, addresses A+1 ... A+8 are prefetched. In an ideal case, if the application exhibits such spatial patterns, the temporal prefetcher should also be able to capture them. However, due to the inability to precisely track the page access patterns beyond the first usage of any given page, the Linux' LRU mechanism is imprecise which leads to deviations in which pages are swapped out. Thus the temporal approach may not be able to capture the relationships between pages with adjacent Virtual Addresses.

5.4.4.4 Delta-Pattern Driven Prefetch

A delta pattern is a stream of differences between adjacent page addresses, e.g. if pages with addresses 0x4, 0x6, 0x9 and 0x8 were accessed, the corresponding delta-pattern would look like this: "+2, +3, -1". Our predictor is based on the

observation that the delta-patterns between page fault addresses are repeating over time. By keeping a history of such delta-patterns and the following deltas, we can look up any pattern and predict which page is likely to be faulted next. Moreover, by using the newly-predicted delta, we can loop around and speculatively predict the second page that is likely to be faulted, and so forth. These predictions have diminishing values, so it makes sense to have a decay mechanism in place that would stop infinite lookahead. We are using the observed accuracy of the scheme to discount each subsequent speculative prediction, and stop when such discount falls below a specific threshold. The approach is based on the "Variable length delta predictor" (VLDP) [88].

5.4.4.5 Combination Prefetch Scheme

We notice from our experiments that applications exhibit changing execution phases where different predictors achieve varying accuracies. It would thus be beneficial to combine several predictors into one scheme to take advantage of their strengths and amortize the shortcomings. We observed that the predictions from the Temporal and Spatial algorithms have a rather small intersection (i.e. about 10% of all the predictions are to the same pages for both algorithms). These intersecting predictions have a very high accuracy. We thus propose the following scheme of combining the predictors. First, find the intersection of all the predictions and add to the queue (again, refer to Figure 5.6). Then, merge the remaining predictions into the queue weighing the predictors by the observed accuracy in the preceding period.

We shall now describe the approach used to merge such remaining predictions, after the intersecting ones were taken care of. Let P_1 be the utilization (accuracy) of the first scheme, and P_2 the accuracy of the second one, also let n be the total number of the non-intersecting predictions that need to be inserted in the secondary

queue. The following formulas describe the relative weights, and the numbers of predictions that are added to the queue.

$$W_1 = n \times \frac{p_1}{p_1 + p_2}, \tag{5.1}$$

$$W_2 = n \times \frac{p_2}{p_1 + p_2}. \tag{5.2}$$

We note that in the theoretical ideal case, it is best to give a 100% preference to the more-accurate scheme. Clearly, if the utilizations were constant across the predictions within each scheme, we could expect that the total utilized prefetches for the combination scheme would be:

$$U_c = W_1 \times p_1 + W_2 \times p_2 = n \times \frac{p_1}{p_1 + p_2} \times p_1 + n \times \frac{p_2}{p_1 + p_2} \times p_2 = n \times \frac{p_1^2 + p_2^2}{p_1 + p_2}. \tag{5.3}$$

Assuming $p_2 > p_1$, i.e. the second scheme is more accurate, and comparing the number of utilized pages with the combination scheme above vs. just the second scheme, we get:

$$U_c - U_2 = n \times \frac{p_1^2 + p_2^2}{p_1 + p_2} - n \times p_2 = n \times \frac{p_1^2 - p_1 p_2}{p_1 + p_2} < 0. \tag{5.4}$$

However, in practice the accuracy of the predictions is not constant and diminishes towards the tail of the queue. Diminishing total page utilizations would thus be observed if only the predictions of the scheme with higher *average* accuracy were prefetched. Additionally, we want to keep fetching at least a few predictions from every scheme to be able to track utilizations. The weighted merging is thus a reasonable approach. In this manner, the more accurate predictions from both schemes get a higher chance of being prefetched, and we are constantly adapting to the appli-

cation behavior by leveraging a variety of predictors and choosing the best possible prediction approach.

We also experimented with an "overweighting" approach where the predictor with higher accuracy is assigned an additional 10% heavier weight, i.e. if the second scheme is more accurate, the weights are redistributed as follows:

$$W_1^{'} = 0.9 \times W_1, \tag{5.5}$$

$$W_2^{'} = W_2 + 0.1 \times W_1. \tag{5.6}$$

This approach provides an additional 0.8% performance improvement on average, and it is used in our proposed schemes.

5.4.5 How Many Pages to Prefetch

It is challenging to balance between the number of prefetched pages, given the limited bandwidth due to the demand faults having higher priority, and the diminishing utilization (the farther ahead we prefetch, the more the likelihood that the pages would not be utilized and/or would be already in the Main Memory). We have experimented with throttling the number of prefetches issued per demand fault based on the occupancy of the secondary queue. The intuition was that the number of entries waiting in the secondary queue directly correlates with the demand fault pressure in the system (since the secondary queue does not issue requests until the main queue is empty). However, our experiments show that this approach does not yield any significant improvement in performance. The major reason is that by throttling the newer, more useful prefetches we are in essence giving more weight to the older, potentially less useful, ones.

We have arrived at an optimal static number of 16 prefetches queued in the

secondary queues per demand fault. The actual number of prefetched pages is thus automatically controlled by the main queue occupancy. The more demand faults an application experiences, the less prefetches will reach the swap file. Additionally, the LIFO scheme of selecting prefetch requests across multiple page faults makes the actual number of queued requests not so critical.

5.5 Evaluation

In this section we evaluate the application performance improvement due to our proposed schemes. We also analyze the contributing aspects to the operation of our algorithms.

5.5.1 Methodology

We evaluated our schemes on a real system with 96GB DRAM and Intel Xeon six-core 1.9 GHz processor. We implemented our approaches in the Linux Kernel 3.13.0. We used applications from SparkBench suite [49]. The applications were configured with large working sets/memory footprints (8 GB for SparkBench). The conventional RAMDisk driver (BRD) was augmented to emulate an NVM device with a standard request queue and pre-settable operation latency[2]. We configured 10 RAMDisks to utilize a total of 92 GB of system RAM thus leaving about 4 GB for the applications , and set the devices as the system's "NVM swap devices". In that way, we induced memory pressure forcing the OS to swap the pages into the emulated NVM devices. The baseline run was done with one emulated NVM device, thus with a parallelism of 1 request per operation, and using the conventional OS readahead, like a freshly-installed system would be configured.

[2]We used 50 μs and 10 μs emulated device latencies for our experiments.

5.5.2 Performance Improvement

We show what performance can be achieved with the conventional OS techniques and compare that with our approaches. Figure 5.7 compares the baseline OS Readahead with 1 swap device to the OS Readahead with 10 swap devices available. One important technical contribution of our work is the technique of "prefetch offloading". This technique is first applied in Concurrent device Offloaded Readahead Prefetch (CORP) scheme, utilizing 10 swap devices, employing the readahead predictor, and offloading to a separate core in order to allow application to continue running as soon as possible after the demand page fault has been serviced ("CORP" bars in the Figure), which provides a significant performance boost. Note that this technique inherently contains the critical-page-first approach. The rest of this work assumes such mechanism of prefetch offloading utilizing concurrent devices in every proposed scheme. We note that the higher concurrency of NVM devices can be better exploited by this mechanism.

As shown in the figure, we also compared the performance of the OS readahead scheme utilizing the traditional "deadline" I/O scheduler vs. a "noop" scheduler. The main difference between the two is that the deadline scheduler uses an elevator algorithm for scheduling the requests in a manner optimized for a spinning disk (i.e. the requests are sorted, and executed in the ascending/descending order so as to minimize disk seek times etc.). The noop scheduler executes the requests in the exact order as they arrive, thus allowing for a faster operation on NVM devices, with minimal scheduling overhead. In our experiments, uting the deadline scheduler impedes the performance by 3% on average, compared to the noop scheduler. In the rest of the work, the noop scheduler is used. Next, we note that by completely disabling the OS readahead ("OS no readahead" bars in the Figure) it is possible to

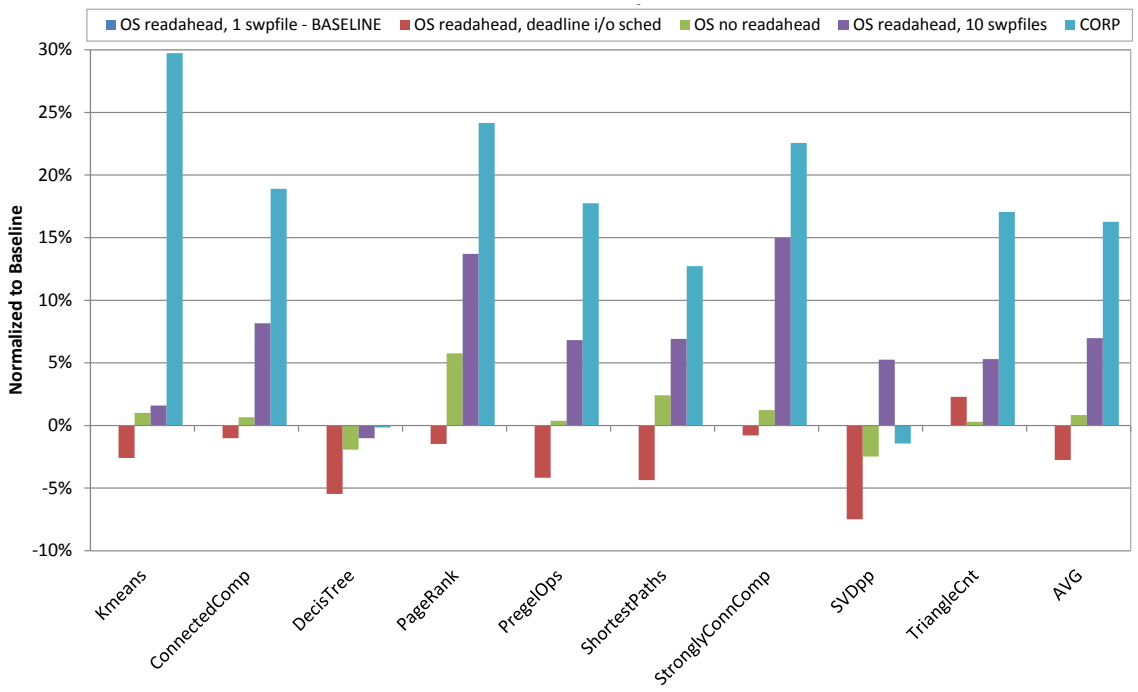


Figure 5.7: SparkBench suite running time improvement with OS readahead, and the offloaded readahead technique.

gain up to 6% application performance (depending on the application memory access patterns, the readahead requests increase the latency for the demand requests and tie the storage device bandwidth), and increase the performance by 1% on average.

Note that due to the way the OS is managing multiple swap devices using a round-robin mechanism, the adjacent pages in the LRU list get swapped out into consecutive swap files. Consequently, with ten swap files, on page faults we have a 10× boost in bandwidth, which allows the OS schemes to perform better with more swap files. However, the figure shows that modifications to the OS readahead mechanism are needed to extract this performance potential out of the fast devices. In the figure, the conventional OS readahead mechanism with 10 swap files is only able to increase the performance by 7% on average, compared to the baseline with

1 swap file. On the other hand, the proposed CORP approach achieves a higher performance gain of 16% on average. We note also that increasing the number of swap devices is in principle only limited by the NVM parallelism available.

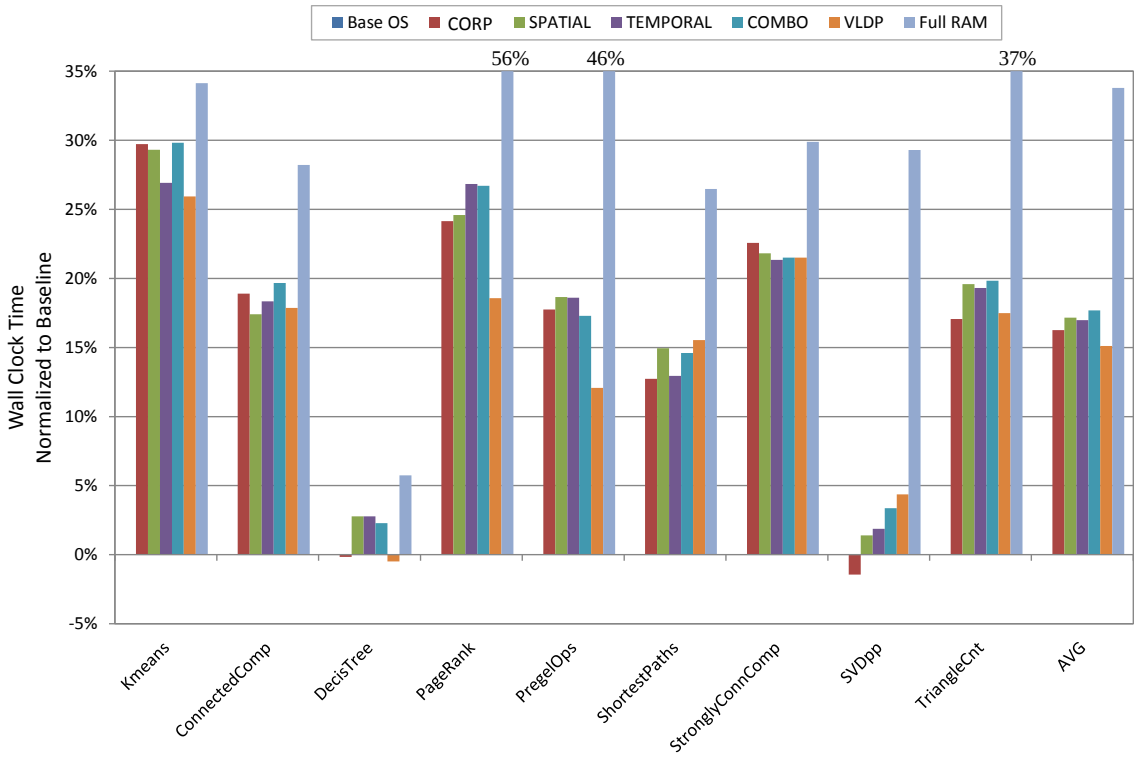


Figure 5.8: SparkBench suite running time improvement with various prediction schemes.

Figure 5.8 presents a comparison of the SparkBench applications with various proposed prefetching schemes vs. the baseline OS readahead. The bars represent the wall clock running time, normalized to the baseline wall clock time. The "AVG" bars are geometric mean averages of the respective schemes across the applications. We also include the "Full RAM" bar which depicts the maximum practical performance improvement if the full 96GB of Main Memory were available to each application.

Note that for KMeans, ConnectedComponent, and StronglyConnectedComponent, the prefetch algorithms come within 70-85% of the maximum achievable speedup. On average, our schemes are able to gain 50% of the lost performance.

In the figure, we compare the following schemes: SPATIAL means the spatial scheme, TEMPORAL denotes the temporal scheme. COMBO stands for the combination of the spatial and temporal schemes, with the intersecting predictions having the highest priority and the rest of the respective predictions being capped according to the observed accuracies of the underlying schemes. VLDP represents the delta-stream predictor. The above mentioned schemes employ the offloading mechanism with a separate CPU core queueing the prefetch requests into the secondary queue, out of the way of the demand faults. As described earlier, the secondary queue is utilized by the swap device only when the main queue is empty. Conversely, CORP is offloaded to a separate CPU core, but is injecting the prefetch requests into the main device queue. I.e. instead of blocking the application while scheduling the prefetches, just the faulted page is scheduled in the original fault, while the prefetched pages are scheduled in the background, after the faulted page.

Overall, the running time is improved by up to 30% (KMeans) and no worse than 2% (DecisionTree), and 18% on average. We note that our combination approach does not degrade the performance, unlike CORP. For DecisionTree and SVDPlusPlus the CORP approach incurs a 1-3% performance hit. This is due to somewhat low utilization of readahead pages, paired with bursty behavior of page faults, when injecting extra requests into the device queue leads to the increased page fault latencies for legitimate requests, and in turn lower application performance. Note that StronglyConnectedComponent application achieves the best performance with CORP, owing to the higher prefetch utilization and the fact that the predictions are executed with a lower delay compared to the secondary queue approaches.

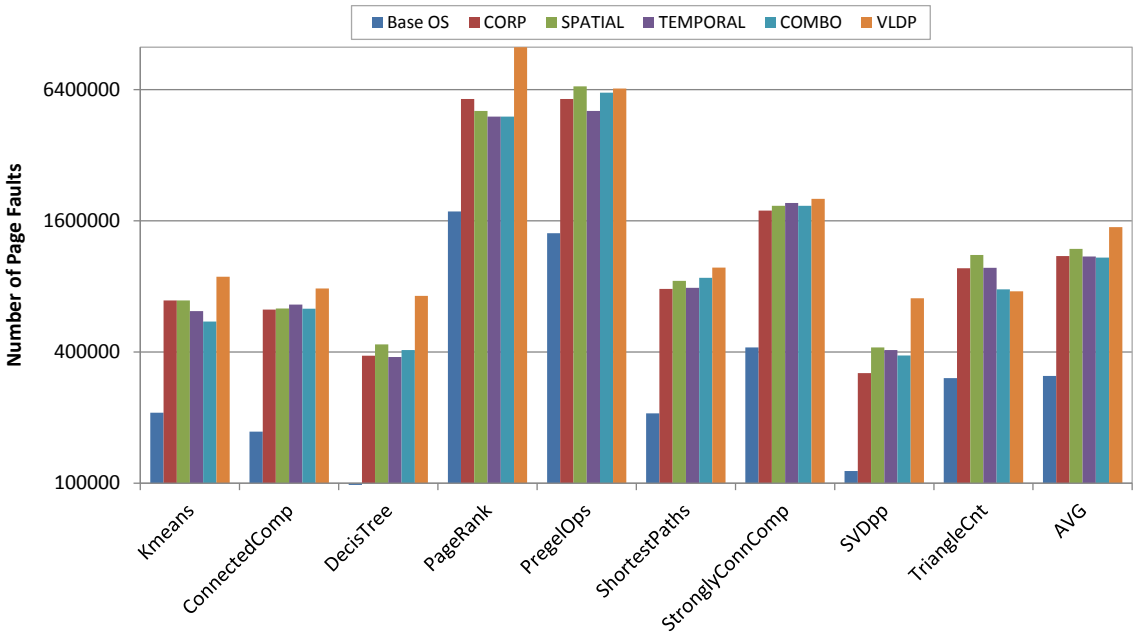


Figure 5.9: SparkBench suite number of page faults with various prediction schemes.

We observe that the spatial and temporal predictors are approximately equal in performance improvement, while behaving differently with various applications (cf. KMeans, PregelOps, and TriangleCnt). On the other hand, CORP, while enjoying an instant request scheduling to the main queue, falls short of the other schemes due to the request latency effect described above.

5.5.2.1 Page Fault Reduction vs. Performance Improvement

Figure 5.9 presents the number of Page Faults in the applications, on a logarithmic scale. Note that in the figure, the lower the bars are, the better, i.e. the base OS readahead reduces the most page faults of the schemes evaluated (the minimal number of page faults, close to zero, is observed when there is enough DRAM in the system to fully contain the applications' working sets). Note, however, how poorly the baseline performs compared to the other schemes (Figure 5.8). This effect is due

to the increased latency of each swap request caused by the filling of the main swap queue. It is very important to take into account this effect on page fault latency, rather than only observing the raw numbers of page faults, of one is to accurately assess the performance of various schemes. Since we conduct our experiments on a real system, the wall clock time accurately reflects the total performance.

5.5.3 Analysis

In this section we analyze the schemes. Recall that in order to improve the application performance under page swapping, we must lower the page fault latency. One way of achieving this is utilizing a faster device (clearly, if the swap device were as fast as DRAM, the performance under swapping would equal to that of the system with enough RAM). The other way is page prefetching. If the prediction is correct and the prefetched page is utilized, the effective latency of the respective page fault is reduced to zero. However, by issuing too many prefetches, or prefetches with low accuracy, we would unnecessarily keep the swap device and the main memory busy, which would lead to higher page fault latencies. It is thus the balance between the accuracy and the number of prefetch requests issued that leads to the optimal application performance.

Figures 5.10 and 5.11 present the prefetched page utilizations for the constituents of the Combination Prefetch scheme, and the final utilizations, respectively. Recall that the scheme gives the first priority to intersecting predictions, and then merges the rest of the predictions according to the observed utilizations of the respective schemes. We note from Figure 5.10 that the Spatial scheme is superior to the Temporal, except in TriangleCnt application. The final utilization numbers are generally good, except for SVDpp and TriangleCnt, at about 30%. Intersecting predictions show very high accuracy, which reinforces the correctness of our design approach of

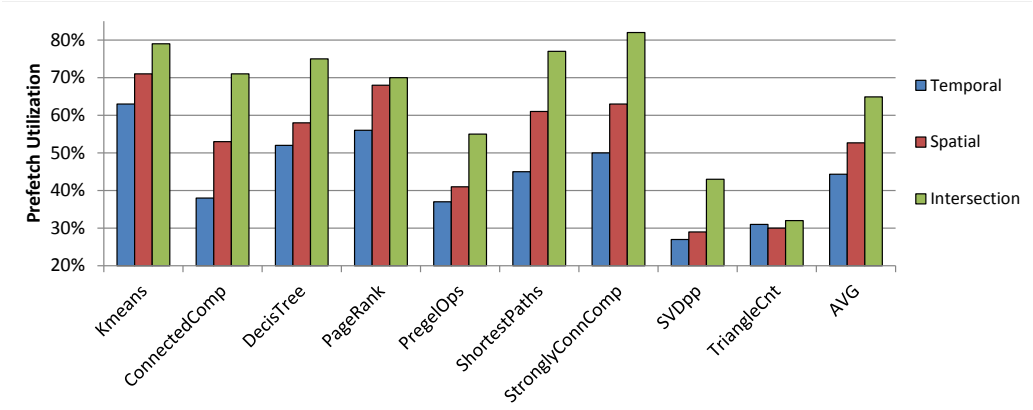


Figure 5.10: Prefetch utilizations for the combination prefetch algorithm, sparkbench suite.

prioritizing them in the combination scheme. However, our experiments show that the number of prefetches from intersecting predictions is 15% of the total prefetches, on average, thus their high accuracy does not contribute significantly to the overall prefetched page utilization and performance improvement.

Since the OS readahead and the Temporal scheme are based on the same principle of operation, we can see that the OS readahead accuracy is low. By injecting the prefetches directly into the main swap queue it is able to prefetch more total pages and thus more useful pages than the schemes with higher accuracy (Figure 5.9). However, it suffers from the increased page fault latency, and overall the performance is lower (Figure 5.8).

Figure 5.12 presents the breakdown of the lifetime of predictions made by the Combination algorithm. The predicted pages are first checked against the Page Table and the Swap Cache and the ones present in those are not considered further ("Filtered before queue" in the figure). The rest of the predictions are placed into the secondary queue where a fraction of them are wiped away due to pressure from

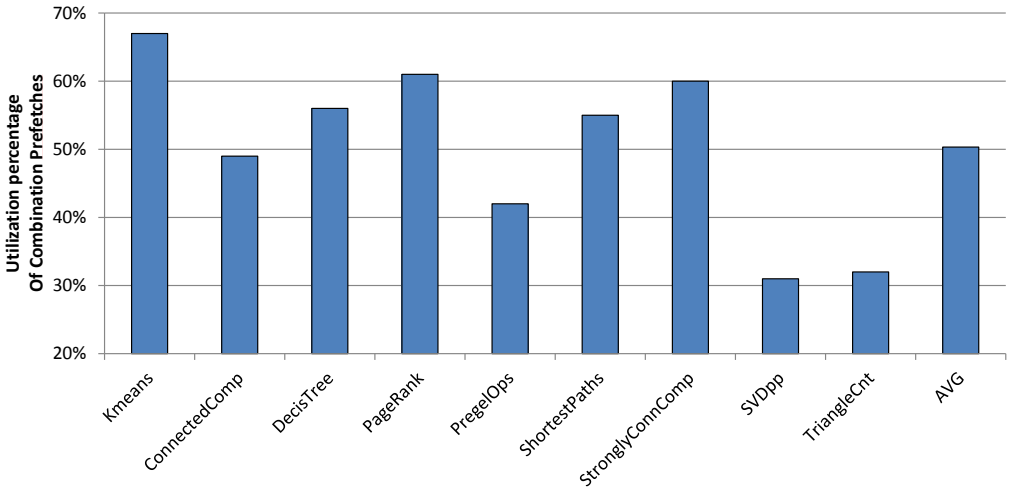


Figure 5.11: Final utilizations for the combination prefetch algorithm.

incoming predictions on one side, and insufficient bandwidth for prefetches on the other ("Wiped from queue"). Before the remaining predictions from the secondary queue are considered for prefetch, they are checked against the Swap Cache once again to ensure no duplicates ("In Memory"). The prefetched pages are placed in the swap cache. Figure 5.13 presents this breakdown as fractions of the total number of predictions made. We observed approximately 50% of the predictions make it to the secondary queue. Further, only about 20% of the total predictions are considered for prefetch. We note that NVM devices with more parallelism and/or lower latencies will help provide more bandwidth for prefetches and allow for increased performance.

5.5.3.1 Future NVMe Devices

We experimented with faster NVM devices by setting the emulated latency to 10 μs, which is about 1.6× the latency of a page read from DRAM (assuming that fetching a 64-byte cache block from DRAM takes on the order of 100 ns, the full 4 KB page will take approximately 6.5 μs to fetch). The results are presented in

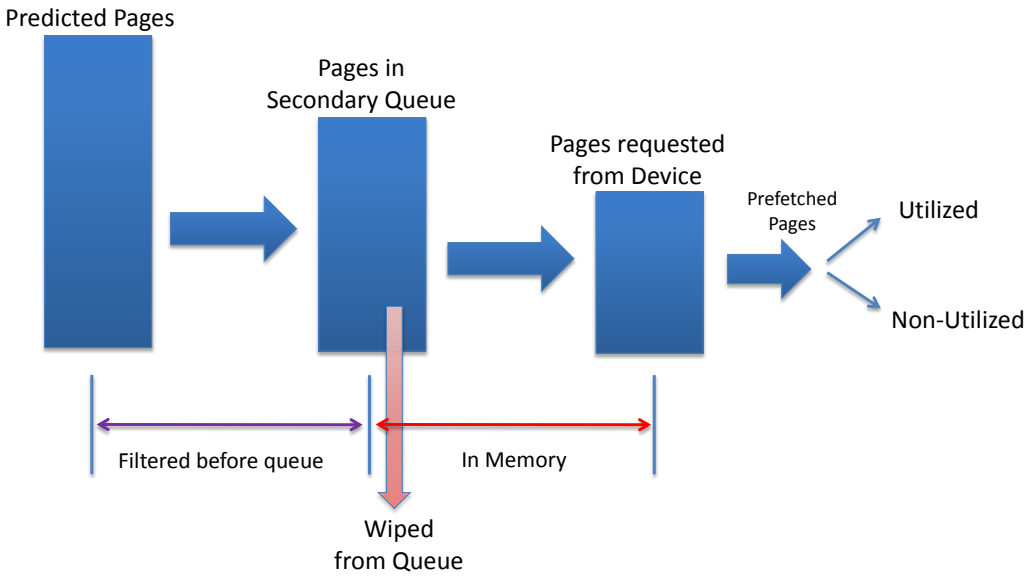


Figure 5.12: Prediction lifetime in the proposed scheme.

Figure 5.14. First, we note that the maximum practical performance improvement (Base OS readahead vs. Full DRAM) is only 21%, on average, with the faster devices (cf. Figure 5.8 at 34%). Theoretically, if the NVM device were as fast as DRAM, there should be no difference in performance when using either type of memory. However, the OS swapping approach is slower due to the bookkeeping overheads. Second, with the lower NVM device latencies, the distances between page faults are shrinking since the applications are able to run faster. For reference, we observed a 16% geometric mean speedup under the Base OS Readahead of the SparkBench applications with 10-μs devices vs. the 50-μs ones. Further, there is less opportunity for prefetching and the overheads of the more complex algorithms may start to negatively affect the performance. We note that the SPATIAL prefetcher achieves 7.7% performance improvement on average, while the Combination scheme boosts the performance by 7%. CORP scheme does as well as the TEMPORAL prefetcher,

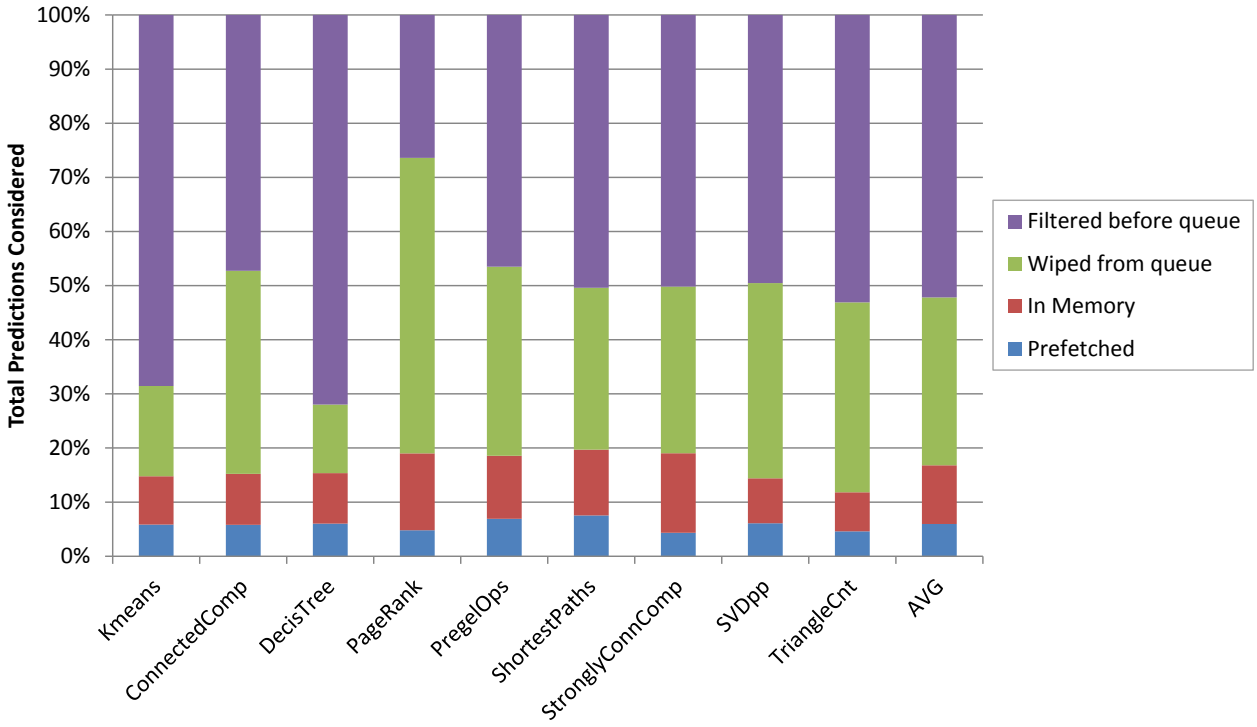


Figure 5.13: Prediction distribution for the combination prefetch algorithm.

however, both schemes fall about 1.5% short of the best prefetchers on average. Overall, our proposed approach adapts to the underlying NVM device characteristics by leveraging the high-accuracy predictions, and is able to achieve about 35% of the potential maximum performance improvement.

5.6 Related Work

Hybrid memory architectures have been investigated in the past [14, 103]. To our knowledge, our work is the first to consider the swap prefetching approach for DRAM + NVM memory. Similar to our two-level approach, schemes using DRAM as a cache for slower NVM memory were proposed [55, 61, 75]. These approaches require additional hardware modifications and are thus less flexible with respect to

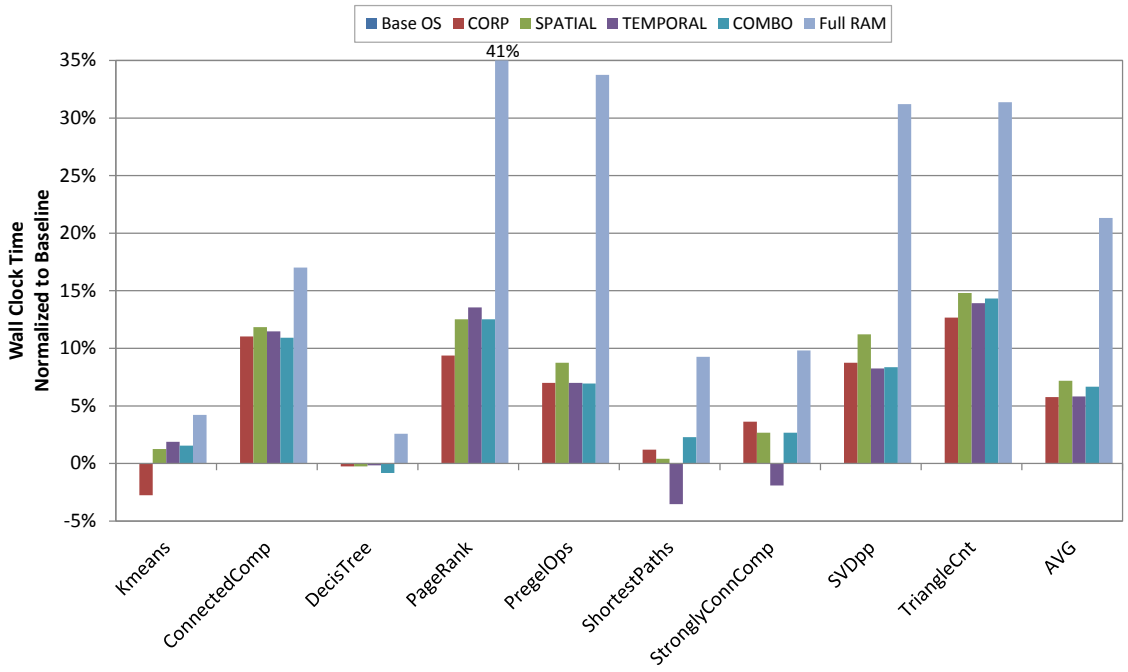


Figure 5.14: SparkBench suite running time improvement with various prediction schemes, 10 μs NVM device latency.

the NVM technology utilized.

Apart from managing DRAM + NVM memory as a two-tier architecture, it is possible to have the two memory types on the same level. Adaptive page placement based on data usage patterns was proposed by Ramos et al. [80]. They looked at PCM as complementary technology, but their approach can be extended to other fast memory technologies as well.

A great amount of research has been devoted to prefetching for CPU caches [9, 33, 93]. Smomgyi et al. proposed spatial memory streaming prefetcher [92]. They noticed that access patterns to similar application data structures are repetitive and it is thus possible to apply the information about past accesses to predict the future ones. Their approach, however, does not scale well to the page level. First, the

"spatial region generations" need to span dozens of OS pages, thus the time it takes to capture such generation grows by several orders of magnitude as compared to tracking the cache blocks. Second, the vast majority of such generations end up full, i.e. the scheme reduces to a simple next-page predictor, this observation inspired our Spatial scheme.

These approaches are a level lower than our proposed approach, are mostly implemented in hardware, and can be utilized independent of our schemes for further speedup. Our approach, being fully software-based, allows for more sophisticated prediction algorithms with higher accuracy. The drawback of page-level prediction is the low amount of feedback available at the OS level, making it harder to determine and prefetch the application page access patterns.

Ferdman et al. show that, while CPU cache prefetching is useful for high-performance scientific applications, modern scale-out workloads are too large for the conventional caches and thus higher-level techniques are needed to boost their performance [17]. They also show that the available memory bandwidth exceeds the requirements of such workloads, which is one of the motivators for us to pursue memory page prefetching.

ScaleMP vSMP Foundation for FLash eXpansion (FLX) [85] is aiming to extend the DRAM with Flash devices. Their approach is based on the shared RAM algorithms, where the remote RAM is replaced with local Flash. However, the system is proprietary and closed-source. Additionally, it is implemented inside a hypervisor, which might negatively affect the system performance. Our approach is running as an integral part of the OS with the maximum possible performance.

Some effort has been devoted to page prefetching in the Linux kernel in the past [51]. The schemes assume a DRAM-only system, need ample free main memory and take care to only operate during idle periods in system activity. They fetch

pages back from the swap device and thus boost the performance for *background applications* where the OS proactively swaps out the working sets even when no memory pressure exists in the system. As such, these approaches are orthogonal to our schemes because we aim at helping the actively-running applications in a DRAM + NVM system with restricted amounts of DRAM.

5.7 Summary

In order to lower the cost of Main Memory in systems with huge memory demand, it is becoming practical to substitute less-expensive NVM for some of the DRAM in the system, for a much lower system cost. The main challenge with NVM, however, is its relatively high access latency. We presented OSVPP, a software-only, OS swap-based page prefetching approach to managing hybrid DRAM and NVM systems. We show that it is possible to gain about 55% of the lost performance due to swapping into the NVM and thus enable the utilization of such hybrid systems for memory-hungry applications, lowering the memory cost while keeping the performance comparable to the DRAM-only system.

6. CONCLUSION

In this dissertation, we addressed some of the challenges related to the adoption of NVM on both the architecture and the software levels. We looked into the possibilities of using NVM for complete DRAM replacement as well for DRAM extension.

In Chapter 2 we addressed the issues related to utilization of NVM for replacing DRAM as Main Memory. We showed that it is vital to control the amount of writebacks from the Last-Level Cache (LLC) to Main Memory. We proposed ARI, an adaptive LLC management policy that modifies replacement *as well as* insertion policies in the cache, in response to the changing program behavior, so as to optimize the writebacks to NVM main memory and at the same time improve the cache miss-rate. Our evaluation indicates that the proposed scheme provides a 49% NVM lifetime improvement through the reduction in the cache writeback rates, while sustaining high application performance by reducing the miss rates at the same time.

In Chapter 3 we revisited the conventional decade-old assumptions in the Operating System about the typical cache sizes and application working sets, which have led to suboptimal performance with increased amount of unnecessary cache misses and writebacks which are critical with NVM. These OS effects on the caches have not been considered in the cache management literature previously. We analyzed the degree of the OS influence on caching performance and presented a software-only, adaptive approach to mitigating such influence with minimal overheads. This approach decreases up to a 100% of the cold cache misses, lowers the overall miss-rates by 15-18% on average across a diverse set of benchmarks, boosting the application performance by up to 15%.

In Chapter 5 we explored the potential of NVMe Flash as main memory. We

proposed the use of PCIe-attached Flash NVM [91] for transparent DRAM extension as a software-only approach, using the OS swapping subsystem as a base. We showed that the OS swapping support of NVMe devices is inefficient since it is based on the old assumptions about the high-latency, sequential-access spinning disks. We presented the solutions to the issues with OS swapping, and built a framework for managing the DRAM+NVMe hybrid system by adopting a predictive page fetching algorithm, providing an impression of large memory capacity with the effective latency of DRAM. Our results indicate that up to 55% of the performance of the full DRAM capacity can be achieved utilizing only a fraction of DRAM extended with NVMe, with the proposed software approach.

As an unconventional approach to utilizing emerging memory technologies, in Chapter 4 we presented a Ternary Content-Addressable Memory (TCAM) design with Flash transistors. We showed an area improvement of 7.9× and a power improvement of 1.64× compared to conventional approaches. Such design could be utilized in Virtual Memory accelerator applications, allowing faster operation when employing NVM for data storage. Instead of increasing the TLB sizes, which would induce additional delay on the critical path of CPU reads, for instance, TCAM could be leveraged for storage of the Page Table and hence fast Virtual-to-Physical address translations, obviating the need of the costly Page Table walks on TLB misses.

REFERENCES

[1] H. An, K. Kim, S. Jung, H. Yang, K. Kim, and Y. Song. The threshold voltage fluctuation of one memory cell for the scaling-down NOR flash. In *2nd IEEE International Conference on Network Infrastructure and Digital Content*, Sept 2010.

[2] Gilles Audemard and Laurent Simon. Glucose 2.1: Aggressive, but reactive, clause database management, dynamic restarts (system description). In *Pragmatics of SAT 2012 (POS'12)*, jun 2012. dans le cadre de SAT'2012.

[3] Avraham Ben-aroya and Sivan Toledo. Competitive analysis of flash-memory algorithms. Technical report, Tel Aviv University, 2006.

[4] C. Bienia, S. Kumar, and K. Li. Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 47–56, Sept 2008.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[7] Simona Boboila and Peter Desnoyers. Write endurance in flash drives: Measurements and analysis. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 9–9, Berkeley, CA, USA, 2010. USENIX Association.

[8] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[9] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[10] E Choi and S Park. Device considerations for high density and highly reliable 3D NAND flash cell in near future. In *IEEE International Electron Devices Meeting (IEDM)*, pages 9.4.1 – 9.4.4, San Francisco, CA, Dec 2012.

[11] Fernando J. Corbató. A paging experiment with the multics system. Technical report, DTIC Document, 1968.

[12] Peter J. Denning. Virtual memory. *ACM Comput. Surv.*, 2(3):153–189, September 1970.

[13] J DeVos, L Haspeslagha, M Demand, K Devriendt, D Wellekens, S Beckx, and J Houdt. A scalable stacked gate NOR/NAND flash technology compatible with high-k and metal gates for sub 45nm generations. In *IEEE International Conference on Integrated Circuit Design and Technology (ICICDT)*, pages 1–4, 2006.

[14] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. Pdram: A hybrid pram and dram main memory system. In *Proceedings of the 46th Annual Design*

Automation Conference, DAC '09, pages 664–469, New York, NY, USA, 2009. ACM.

[15] K. Eshraghian, Kyoung-Rok Cho, O. Kavehei, Soon-Ku Kang, D. Abbott, and Sung-Mo Steve Kang. Memristor mos content addressable memory (mcam): Hybrid architecture for future high performance search engines. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(8):1407–1417, Aug 2011.

[16] V.V. Fedorov, M. Abusultan, and S.P. Khatri. An area-efficient ternary cam design using floating gate transistors. In *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, pages 55–60, Oct 2014.

[17] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 37–48, New York, NY, USA, 2012. ACM.

[18] Alexandre P. Ferreira, Miao Zhou, Santiago Bock, Bruce Childers, Rami Melhem, and Daniel Mossé. Increasing pcm main memory lifetime. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 914–919, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[19] R. F. Freitas and W. W. Wilcke. Storage-class memory: The next storage system technology. *IBM J. Res. Dev.*, 52(4):439–447, July 2008.

[20] B Gamache, Z Pfeffer, and S Khatri. A fast ternary CAM design for IP networking applications. In *IEEE International Conference on Computer Communications and Networks (ICCCN)*, pages 434–439, 2003.

[21] Hongliang Gao and Chris Wilkerson. A dueling segmented lru replacement algorithm with adaptive bypassing. In *JWAC 2010-1st JILP Worshop on Computer Architecture Competitions: Cache Replacement Championship*, 2010.

[22] Jayesh Gaur, Mainak Chaudhuri, and Sreenivas Subramoney. Bypass and insertion algorithms for exclusive last-level caches. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 81–92, New York, NY, USA, 2011. ACM.

[23] M. Ghosh and H. H. S. Lee. Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3d die-stacked drams. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 134–145, Dec 2007.

[24] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th annual international symposium on Computer architecture*, ISCA '83, pages 124–131, New York, NY, USA, 1983. ACM.

[25] Q Guo, X Guo, Y Bai, and E Ipek. A resistive TCAM accelerator for data-intensive computing. In *MICRO'11*, pages 339–350, 2011.

[26] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[27] Intel haswell. `http://ark.intel.com/products/81061/Intel-Xeon-Processor-E5-2699-v3-45M-Cache-2_30-GHz`. 2014.

[28] International Technology Roadmap for Semiconductors (ITRS) Working Group. International Technology Roadmap for Semiconductors (ITRS), 2009 Edition, 2009.

[29] Yasuo Ishii, Mary Inaba, Kei Hiraki, et al. Cache replacement policy using map-based adaptive insertion. In *JWAC 2010-1st JILP Worskhop on Computer Architecture Competitions: Cache Replacement Championship*, 2010.

[30] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 151–162, Washington, DC, USA, 2010. IEEE Computer Society.

[31] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM.

[32] Andhi Janapsatya, Aleksandar Ignjatović, Jorgen Peddersen, and Sri Parameswaran. Dueling clock: Adaptive cache replacement policy based on the clock algorithm. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 920–925, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[33] N. D. E. Jerger, E. L. Hill, and M. H. Lipasti. Friendly fire: understanding the effects of multiprocessor prefetches. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 177–188, March 2006.

[34] Xiaowei Jiang, Asit Mishra, Li Zhao, Ravishankar Iyer, Zhen Fang, Sadagopan Srinivasan, Srihari Makineni, Paul Brett, and Chita R. Das. Access: Smart scheduling for asymmetric cache cmps. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 527–538, 2011.

[35] Norman P. Jouppi. Cache write policies and performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 191–201, New York, NY, USA, 1993. ACM.

[36] Dawoon Jung, Yoon-Hee Chae, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A group-based wear-leveling algorithm for large-capacity flash memory storage systems. In *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '07, pages 160–164. ACM, 2007.

[37] Samira Khan and Daniel A. Jiménez. Insertion policy selection using decision tree analysis. In *Computer Design (ICCD), 2010 IEEE International Conference on*, pages 106–111, 2010.

[38] Samira Khan, Yingying Tian, and Daniel A. Jiménez. Sampling dead block prediction for last-level caches. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 175–186, 2010.

[39] Samira Manabi Khan, Yingying Tian, and Daniel A. Jimenez. Sampling dead block prediction for last-level caches. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 175–186, Washington, DC, USA, 2010. IEEE Computer Society.

[40] M. Kobayashi, T. Murase, and A. Kuriyama. A Longest Prefix Match Search Engine for Multi-Gigabit IP Processing. In *Proc. IEEE ICC*, volume 3, 2000.

[41] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM.

[42] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-change technology and the future of main memory. *Micro, IEEE*, 30(1):143–143, 2010.

[43] Chang Joo Lee, Onur Mutlu, Eiman Ebrahimi, Veynu Narasiman, and Yale N Patt. Dram-aware last-level cache replacement. Technical Report TR-HPS-2010-007, High Performance Systems Group, Department of Electrical and Computer Engineering, The University of Texas at Austin, 2010.

[44] Chang Joo Lee, Onur Mutlu, Veynu Narasiman, Eiman Ebrahimi, and Yale N Patt. Dram-aware last-level cache writeback: Reducing write-caused interference in memory systems. Technical Report TR-HPS-2010-002, High Performance Systems Group, Department of Electrical and Computer Engineering, The University of Texas at Austin, 2010.

[45] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '99, pages 134–143, New York, NY, USA, 1999. ACM.

[46] Hsien-Hsin S. Lee, Gary S. Tyson, and Matthew K. Farrens. Eager writeback - a technique for improving bandwidth utilization. In *Proceedings of the 33rd*

annual ACM/IEEE international symposium on Microarchitecture, MICRO 33, pages 11–21, New York, NY, USA, 2000. ACM.

[47] K-W. Lee, K-S. Kim, S.-W. Shin, S.-S. Lee adn J.-C. Om, G.-H. Bae, and J.-H. Lee. Modeling of Vth shift in NAND flash-memory cell device considering crosstalk and short-channel effects. *IEEE Transactions on Electron Devices*, 55(4):1020–1026, Apr 2008.

[48] Han-Lin Li, Chia-Lin Yang, and Hung-Wei Tseng. Energy-aware flash memory management in virtual memory system. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(8):952–964, Aug 2008.

[49] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 53:1–53:8, New York, NY, USA, 2015. ACM.

[50] M Lin, J Luo, and Y Ma. A low-power monolithically stacked 3D-TCAM. In *IEEE International Symposium on Circuits And Systems*. IEEE, 2008.

[51] Linux Swap Prefetching. Con Kolivas, `https://lwn.net/Articles/153353/`.

[52] Fang Liu, Fei Guo, Yan Solihin, Seongbeom Kim, and Abdulaziz Eker. Characterizing and modeling the behavior of context switch misses. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 91–101, New York, NY, USA, 2008. ACM.

[53] Fang Liu and Yan Solihin. Understanding the behavior and implications of context switch misses. *ACM Trans. Archit. Code Optim.*, 7(4):21:1–21:28, December 2010.

[54] Gabriel H Loh. Extending the effectiveness of 3d-stacked dram caches with an adaptive multi-queue policy. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 201–212. IEEE, 2009.

[55] Gabriel H. Loh and Mark D. Hill. Supporting very large dram caches with compound-access scheduling and missmap. *Micro, IEEE*, 32(3):70–78, 2012.

[56] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[57] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[58] Anthony J. McAuley and Paul Francis. Fast Routing Table Lookup Using CAMs. *Proc.IEEE INFOCOM*, pages 1382–1391, March-April 1993.

[59] Jagan Singh Meena, Simon Min Sze, Umesh Chand, and Tseung-Yuen Tseng. Overview of emerging nonvolatile memory technologies. *Nanoscale research letters*, 9(1):1–33, 2014.

[60] Inc Meta-Software. HSPICE user's manual. Campbell, CA.

[61] Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan. Enabling efficient and scalable hybrid memories using fine-granularity dram cache management. *IEEE Comput. Archit. Lett.*, 11(2):61–64, July 2012.

[62] Pierre Michaud. The 3p and 4p cache replacement policies. In *JWAC 2010-1st JILP Worshop on Computer Architecture Competitions: Cache Replacement Championship*, 2010.

[63] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 261–270, Washington, DC, USA, 2009. IEEE Computer Society.

[64] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association.

[65] K Pagiamtzis and A Sheikholeslami. Content-addressable memory (CAM) circuits and architectures: a tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, March 2006.

[66] B. Park, J Song, E Cho, S Hong, J Kim, Y Choi, Y Kim, S Lee, C Lee, D Kang, D Lee, B Kim, Y Choi, W Lee, J Choi, K Suh, and T Jung. 32nm 3-bit 32gb NAND flash memory with DPT (double patterning technology) process for mass production. In *IEEE Symposium on VLSI Technology*, pages 125–126, June 2010.

[67] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. Cflru: A replacement algorithm for flash memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for*

Embedded Systems, CASES '06, pages 234–241, New York, NY, USA, 2006. ACM.

[68] T. B. Pei and C. Zukowski. VLSI Implementation of Routing Tables: Tries and CAMs. *Proc. IEEE INFOCOM*, 2:515–524, 1991.

[69] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '03, pages 318–319, New York, NY, USA, 2003. ACM.

[70] PTM website. `http://ptm.asu.edu/`.

[71] Moinuddin K. Qureshi, Michele M. Franceschini, and Luis A. Lastras-Montano. Improving read performance of phase change memories via write cancellation and write pausing. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–11, 2010.

[72] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[73] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 14–23, New York, NY, USA, 2009. ACM.

[74] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 14–23, New York, NY, USA, 2009. ACM.

[75] Moinuddin K. Qureshi and Gabriel H. Loh. Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 235–246, Washington, DC, USA, 2012. IEEE Computer Society.

[76] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[77] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[78] Moinuddin K. Qureshi, David Thompson, and Yale N. Patt. The v-way cache: Demand based associativity via global replacement. In *In Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 544–555, 2004.

[79] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital Integrated Circuits*. Prentice Hall, 2nd edition, 2003.

[80] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 85–95, New York, NY, USA, 2011. ACM.

[81] Synopsys Raphael Interconnect Analysis Tool. `http://www.synopsys.com/Tools/TCAD/InterconnectSimulation/Pages/Raphael.aspx`.

[82] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Comput. Archit. Lett.*, 10(1):16–19, January 2011.

[83] University of Oregon Route Views Project. `http://www.routeviews.org`.

[84] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 57–68, New York, NY, USA, 2011. ACM.

[85] ScaleMP vSMP Foundation FLX. `http://www.scalemp.com/media-hub/events/ismc/`.

[86] Nak Hee Seong, Dong Hyuk Woo, and Hsien-Hsin S. Lee. Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 383–394, New York, NY, USA, 2010. ACM.

[87] Devavrat Shah and Pankaj Gupta. Fast Updating Algorithms for TCAMs. *IEEE Micro*, 21:36–47, Jan/Feb 2001.

[88] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H. Pugsley, and Zeshan Chishti. Efficiently prefetching complex

address patterns. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 141–152, New York, NY, USA, 2015. ACM.

[89] H Shim, S Lee, B Kim, N Lee, D Kim, H Kim, B Ahn, Y Hwang, H Lee, J Kim, Y Lee, H Lee, J Lee, S Chang, J Yang, S Paark, S Aritome, S Lee, K Ahn, G Bae, and Y Yang. Highly reliable 26nm 64Gb MLC E2NAND (embedded-ECC and enhanced-efficiency) flash memory with MSP (memory signal processing) controller. In *IEEE Symposium on VLSI Technology*, pages 216–217, June 2011.

[90] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. Ibm power7 multicore server processor. *IBM Journal of Research and Development*, 55(3):1:1–1:29, 2011.

[91] Sivashankar and S. Ramasamy. Design and implementation of non-volatile memory express. In *Recent Trends in Information Technology (ICRTIT), 2014 International Conference on*, pages 1–6, April 2014.

[92] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 252–263, Washington, DC, USA, 2006. IEEE Computer Society.

[93] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 63–74, Feb 2007.

[94] Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C. Hunter, and Lizy K. John. The virtual write queue: Coordinating dram and last-level cache policies. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 72–82, New York, NY, USA, 2010. ACM.

[95] Ranjith Subramanian, Yannis Smaragdakis, and Gabriel H. Loh. Adaptive caches: Effective shaping of cache behavior to workloads. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 385–396, Washington, DC, USA, 2006. IEEE Computer Society.

[96] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *J. Supercomput.*, 28(1):7–26, April 2004.

[97] K. Takeuchi. Novel co-design of nand flash memory and nand flash controller circuits for sub-30 nm low-power high-speed solid-state drives (ssd). *Solid-State Circuits, IEEE Journal of*, 44(4):1227–1234, April 2009.

[98] Dominique Thiebaut and Harold S. Stone. Footprints in the cache. *ACM Trans. Comput. Syst.*, 5(4):305–329, October 1987.

[99] J Wade and C Sodini. A ternary content addressable search engine. *IEEE Journal of Solid-State Circuits*, 24(4):1003–1013, Aug 1989.

[100] J Wakerly. *Digital Design Principles and Practices*. Prentice Hall, 1990.

[101] Zhe Wang, Samira M. Khan, and Daniel A. Jiménez. Improving writeback efficiency with decoupled last-write prediction. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pages 309–320, 2012.

[102] B. Wicht, T. Nirschl, and D. Schmitt-Landsiedel. A yield-optimized latch-type sram sense amplifier. In *Solid-State Circuits Conference, 2003. ESSCIRC '03.*

Proceedings of the 29th European, pages 409–412, Sept 2003.

[103] Xiaoxia Wu, Jian Li, Lixin Zhang, Evan Speight, Ram Rajamony, and Yuan Xie. Hybrid cache architecture with disparate memory technologies. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 34–45, New York, NY, USA, 2009. ACM.

[104] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 6. ACM, 2015.

[105] Wei Xu, Tong Zhang, and Yiran Chen. Design of spin-torque transfer magnetoresistive ram and cam/tcam with high sensing and search speed. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 18(1):66–74, Jan 2010.

[106] HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael A. Harding, and Onur Mutlu. Row buffer locality aware caching policies for hybrid memories. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 337–344, 2012.

[107] Ying Zheng, B. T. Davis, and M. Jordan. Performance evaluation of exclusive cache hierarchies. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '04, pages 89–96, Washington, DC, USA, 2004. IEEE Computer Society.

[108] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared re-

sources in multicore processors. *ACM Comput. Surv.*, 45(1):4:1–4:28, December 2012.