IMPROVING PROCESSOR DESIGN BY EXPLOITING PERFORMANCE VARIANCE

A Dissertation

by

ZHE WANG

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,	Daniel A. Jiménez
Committee Members,	Paul V. Gratz
	Eun Jung Kim
	Valerie E. Taylor
Head of Department,	Nancy M. Amato

August 2014

Major Subject: Computer Science

ABSTRACT

Programs exhibit significant performance variance in their access to microarchitectural structures. There are three types of performance variance. First, semantically equivalent programs running on the same system can yield different performance due to characteristics of microarchitectural structures. Second, program phase behavior varies significantly. Third, different types of operations on microarchitectural structure can lead to different performance.

In this dissertation, we explore the performance variance and propose techniques to improve the processor design.

We explore performance variance caused by microarchitectural structures and propose *program interferometry*, a technique that perturbs benchmark executables to yield a wide variety of performance points without changing program semantics or other important execution characteristics such as the number of retired instructions. By observing the behavior of the benchmarks over a range of branch prediction accuracies, we can estimate the impact of a microarchitectural optimization optimization and not the rest of the microarchitecture.

We explore performance variance caused by phase changes and develop prediction-driven last-level cache (LLC) writeback techniques. We propose a rank idle time prediction driven LLC writeback technique and a last-write prediction driven LLC writeback technique. These techniques improve performance by reducing the write-induced interference.

We explore performance variance caused by different types of operations to Non-Volatile Memory (NVM) and propose LLC management policies to reduce write overhead of NVM. We propose an adaptive placement and migration policy for an STT-RAM-

based hybrid cache and writeback aware dynamic cache management for NVM-based main memory system. These techniques reduce write latency and write energy, thus leading to performance improvement and energy reduction.

Dedicated to my Parents and Grandparents.

ACKNOWLEDGEMENTS

I would like to thank many people who gave me assistance in my research and contributed to this dissertation.

First of all, I would like to thank my advisor Daniel A. Jiménez. My interests in computer architecture started with me taking the CS5513 Computer Architecture class which was taught by Daniel. After taking the class, I went to Daniel's office and told him I wanted to work with him in computer architecture research which turned out to be one of the best decisions I have ever made. During my graduate study, Daniel has been deeply involved, with patient mentoring and insightful guiding of my research. Daniel's enthusiasm in research influenced and inspired me to have fun in what I'm doing, as he used to say "if you can't have fun in this research project, you don't have to do it." Daniel has made it his responsibility to provide me with the research resources and built the research context which made it is possible for me to do the top-level research. I am and will always be grateful for all the help he gave me to accomplish my goals.

I would like to thank my committee members at UTSA and Texas A&M for their insightful feedback on my work. They are Paul V. Gratz, Daniel A. Jiménez, Eun Jung Kim, Valerie E. Taylor, Dakai Zhu, Hugh Maynard, Rajendra V. Boppana and Byeong Lee.

I would like to thank Yuan Xie who was my mentor during my intern at AMD research in Beijing. He helped me understand the Non-Volatile Memory and collaborated on two of our Non-Volatile Memory projects. I appreciate the generous help and invaluable advice Yuan provided me during the study.

I would like to thank Sooraj Puthoor and Bradford M. Beckmann who were my mentors during my intern at AMD research in Austin. While Brad directed my GPU study and

helped me understand the GPU architecture in high level, Sooraj helped me with learning the simulator and figuring out the implementation details of the experiments on a daily basis. Thank you for giving me a stimulating intern experience.

I also would like to thank Cong Xu, a graduate student at Pennsylvania State University. He collaborated with us on the APM project. He helped derive the STT-RAM parameters and helped me understand the NVSim simulator.

I am very thankful to many other graduate peers and colleges for their contribution to my research through discussions, suggestions on paper drafts and feedback on practice talk. They are: Jichi Guo, Yingying Tian, Elvira Teran, Samira Khan, kyungwook Chang, Somaieh Bahrami, Ting Cao, Yi Xu, Shan, Guangyu Sun, Ehsan Fatehi, Andrew Targhetta, Jinchun Kim and Luke McHale.

My special thanks to my friends: Meng Sun, Xin Ding and Juan Yang. Thank you for always being there to support me, share my happiness and help me get through countless frustrations.

Finally, I am deeply in debt to my family for being there for me. Grandpa Jianying Xia taught me independent thinking by personal example, and told me that it is one of the most important characteristics I should have. Grandma Xianzheng Li is the most hard working and considerate person I know. This dissertation could not have been written without the impacts of my grandparents to my life. I also want to thank my aunt Shaohua Xia for taking care of the family while I am away pursuing my PHD study. Finally, many thanks to my parents Guohua Xia and Mingxiang Wang for their caring love and sacrifices. Their unconditional love and support gave me the courage to complete this long journey.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Programs exhibit significant performance variance in their access to microarchitectural structures. There are three types of performance variance. First, there is performance variance caused by microarchitectural structures. For instance, semantically equivalent programs running on the same system with different code placements can yield different performance. This is caused by microarchitectural structures that use a hash of instruction and data addresses, where different code layout will result in a difference impact on performance. Second, there is performance variance caused by phase change. When a program goes through phases, the behavior of microarchitecture events can be different, such as cache miss ratio, branch misprediction ratio and memory access patterns, which lead to performance variance. Third, there is performance variance caused by different types of operations. Read and write operations have different access latency and power consumption in NVM-based memory. In this dissertation, we exploit performance variance to improve processor design.

1.1 Performance Variance Caused by Microarchitectural Structures

Mytkowicz *et al.* introduce the technique of object file reordering for showing that different link orders of object files, as well as other seemingly random and harmless details of an experimental setup, can yield significantly different performance [54]. Since several microarchitectural structures use a hash of instruction and data addresses. Such as caches, translation lookaside table and branch predictor. Sometimes addresses will accidentally collide in some microarchitectural structure. A particular code and data placement will result in a particular number of accidental collisions with a particular impact on performance. A different layout will result in a difference impact on performance, thus yields performance variance.

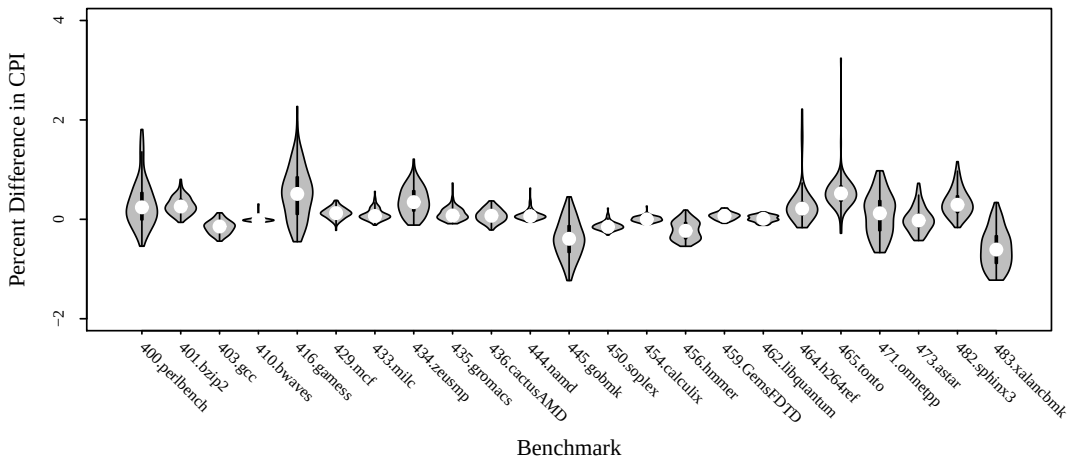


Figure 1.1: Violin plots for SPEC CPU 2006 percentage performance variation with code reordering.

Figure 1.1 shows the percent difference from average performance as measured by cycles-per-instruction (CPI) caused by 100 random but plausible code reorderings for the SPEC CPU2006 benchmarks. The graph is a violin plot, showing the probability density at each CPI value, i.e., the thickness at each CPI value is proportional to the number of CPIs observed in that neighbourhood. Clearly, some benchmarks are greatly affected by differences in instruction addresses while some are less sensitive.

By exploring the performance variance caused by code reordering, we develop a technique to build a performance model for program and microarchitecture by using real systems. The technique is called *Program Interferometry*. It is based on perturbing placement of code and data. By measuring the resulting adverse microarchitectural events using different code and data replacements, we can build a performance model for the program and microarchitecture. Compared with cycle-accurate simulators which are inaccurate with respect to real systems because many of the details of real systems are difficult or impossible to model or even to know about [10], the performance model can explore new microarchitectural ideas in the absence of clear information about what future microarchitectures

will look like.

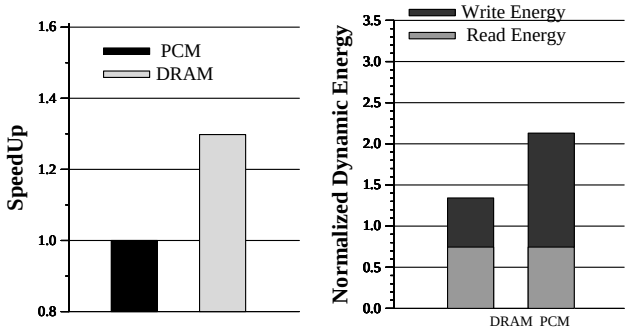
1.2 Performance Variance Caused by Phase Change

Programs can go through phases where the phase behavior varies significantly. When the program runs into different phases, the behavior of microarchitecural events are different, such as cache miss ratio, branch misprediction ratio and memory access patterns. We explore the memory access variance caused by phase change to improve the memory efficiency.

Memory access latency is a major performance bottleneck. A LLC miss can stall the pipeline and require hundreds of cycles of delay. Memory write requests compete with read requests for the available memory resources, delay the service of the following read requests. This write-induced interference can significantly degrade the system performance.

The memory access pattern exhibits significant variance. Memory read requests tend to come in bursts. The DRAM can busy service the memory requests for a while then idle for a while. Additionally, in modern DDRx-based systems, multiple memory controllers and multiple ranks are used to service memory requests in parallel. Due to workload characteristics and load imbalance, some ranks often have idle cycles while the application is running.

By exploring the memory access variance, we develop the prediction driven last-level cache writeback (LLC) technique. We propose a rank idle time prediction driven LLC writeback technique. This technique sends write request to DRAM during the long rank idle period, thus minimizing the delay it caused to the following read requests. We also propose a last-write prediction driven LLC writeback technique. It improves the writeback efficiency by increasing the write scheduling space. Our techniques significantly reduce the write-induced interference.



(a) Speedup (b) Dynamic Energy

Figure 1.2: The performance and dynamic energy impact of write on various systems

1.3 Performance Variance Caused by Operation Types

Read and write operations in memory have different access latency and power consumptions, especially for NVM-based memory. The latency and energy of write operations for NVM are significantly higher than for read operations. The long write latency can degrade performance by causing large write-induced interference to subsequent read requests. The high write energy can increase power consumption.

Figure 1.2 shows average performance and dynamic energy impacts of write requests on various systems for memory intensive SPEC CPU2006 benchmark. We assume that the read and write memory requests for DRAM-based main memory have similar access latency and dynamic power consumption. For PCM-based main memory, the write latency and energy consumption are assumed to be $10X$ of that for the read requests. The scheduling policy we used for evaluation is *read prioritizes write* [85]. From Figure 1.2(a), we can see that the speedup of DRAM-based main memory is 30% compared to PCM-based main memory. Figure 1.2(b) shows the write energy dominates the PCM energy consumption, and it consumes 65% of total dynamic energy consumption, although write requests only account for 25.5% of all the memory accesses.

By exploring the performance variance caused by asymmetric read and write opera-

tions, we propose LLC management policy to reduce the large write overhead of NVM. We propose adaptive placement and migration policy for an STT-RAM-based hybrid cache.It can achieve high performance by making use of the large capacity of STT-RAM and maintain low write overhead using SRAM. We also propose writeback-aware dynamic cache management for NVM-based main memory system. The technique improves system performance and energy efficiency by reducing the number of writeback requests to NVM-based main memory.

1.4 Thesis Statement

Programs exhibit significant performance variance in their access to microarchitectural structures. To the extent that this variance is predictable, it can be exploited to improve processor design.

1.5 Contributions

The dissertation will make the following original contributions:

- We explore the performance variance caused by microarchitectural structures and propose program interferometry technique [83]. This technique elicits microarchitectural events such as branch mispredictions and cache misses to enable the development of a performance model for a given program. We use program interferometry to develop a branch prediction performance model for SPEC CPU 2006 benchmarks running on the Intel Xeon E5440. Based on regression models developed with branch interferometry, we make specific predictions about the performance of the benchmarks at different branch prediction accuracies. Using a branch prediction simulator and our regression models, we estimate the performance of the benchmarks on a hypothetical Intel Core optimized with different branch predictors. We simulate only the branch predictor and do not need to simulate the rest of the microarchitecture.

- We explore the memory access variance caused by phase change and propose a rank idle time prediction driven LLC writeback technique [86] that makes use of the rank idle cycles to isolate the service of memory read and write requests as much as possible. This technique uses a low-overhead *rank idle time predictor* to predict long periods of idle time in memory ranks. Scheduled write requests are written back to the memory guided by the predictor to reduce the write-induced interference.

- We propose a decoupled last-write prediction driven LLC writeback technique [85]. This technique makes last-write blocks in the LLC available to the memory controller for scheduling. It effectively expands the write scheduling space and balances memory bandwidth by re-distributing memory write requests, thus reducing write-induced interference. The technique is completely decoupled from the LLC replacement policy.

- We explore the asymmetric read and write operation problem of NVM and propose adaptive block placement and migration policy for an STT-RAM-based hybrid LLC [84]. In the technique,LLC write accesses are categorized into three classes: prefetch-write, demand-write, and core-write. Our proposed technique places a block into either STT-RAM lines or SRAM lines by adapting to the access pattern of each class. An access pattern predictor is proposed to direct block placement and migration, which can benefit from the high density and low leakage power of STT-RAM lines as well as the low write overhead of SRAM lines.

- We propose a writeback-aware dynamic cache management technique to help mitigate the write overhead in NVM-based memory [87]. The technique predicts blocks that are frequently written back from the LLC. The LLC sets are dynamically partitioned into a frequent writeback list and a non-frequent writeback list. It keeps a best size of each list in the LLC.

2. BACKGROUND AND RELATED WORK

This dissertation explores performance variance to develop the performance model, reduce the write-induced interference in main memory and mitigate write overhead of NVM. To provide context of our research, we now give background and review some of the recent work related to our research.

2.1 Exploring Performance Variance to Develop the Performance Model

This section gives the background and recent work related to developing the performance model by exploring performance variance.

2.1.1 Eliciting Performance Variance

Mytkowicz *et al.* introduce the technique of object file reordering for showing that different link orders of object files, as well as other seemingly random and harmless details of an experimental setup, can yield significantly different performance [54]. That work indicts the architecture and programming languages community for falling victim to measurement bias, i.e., allowing oneself to believe that some observed improvement in program behavior is due to one's own technique rather than a happy coincidence of experimental factors. Our work was partly inspired by Mytkowicz *et al.*. We choose to see the phenomenon they exposed as an interesting opportunity to develop a tool to examine microarchitectural behavior.

Rubin *et al.* propose a framework to explore the space of data layouts using profile feedback to find layouts that yield good performance [71]. They point out that the general problem of optimal data layout is NP-hard and poorly approximable. The space of data layouts is similar to the space of code reorderings, and the impact of data layouts on the data cache is similar to the impact of code placement on the branch predictor and

instruction cache.

2.1.2 Impact of Code Placement on Performance

The impact of code placement on performance has not gone unnoticed in the academic literature. Many code-improving transformations have been proposed based on code placement. Hatfield and Gerald [18], Ferrari [15], McFarling [47], Pettis and Hanson [59], and Gloy and Smith [16] present techniques to rearrange procedures to improve locality using profiling. Mytkowicz *et al.* exploit the kind of performance variance described in this paper to optimize programs [38]. Calder and Grunwald present *branch alignment*, an algorithm that seeks to minimize the number of taken branches by reordering code such that the hot path through a procedure is laid out in a straight line [3]. Young *et al.* present a near-optimal version of branch alignment [92]. Jiménez proposes a technique to use code placement to explicitly avoid branch mispredictions due to conflicts in the predictor tables [27]. Knights *et al.* propose exploiting fortuitous object code orderings to improve performance [38].

From the microarchitecture side, a trace cache is a specialized instruction cache that exploits instruction locality by organizing instructions in the order they are executed, rather than in their static program order[70]. With a trace cache, branch prediction and instruction fetch can be made somewhat immune to the effect of code placement when there is a high hit rate in the trace cache. The Intel Netburst microarchitecture in the Pentium 4 processor line featured a micro-op trace cache [23].

Our technique is not an optimization, but a tool for peering inside the microarchitecture using code placement. If thoughtful code placement optimizations like those mentioned above were widely adopted, our results would show less variance in execution behavior and less confidence in the regression lines. Nevertheless, most production code is not optimized with code placement in mind; thus, our results are widely applicable to real

systems.

2.1.3 Estimating Simulation Results with Regression

Lee and Brooks [40] propose using regression modelling to estimate processor performance and power under a given microarchitectural configuration after sampling a small portion of the microarchitectural design space through simulation. Performance and power are accurately predicted with an error of about 4% on average. Joseph *et al.* propose non-linear [31] regression techniques such as neural networks for estimating CPI given a set of microarchitectural parameters. The technique predicts CPI with an error of 2.8% on average. Both of these proposal are intended to reduce the number of points in a processor design space that must be simulated to find parameters that give good performance.

Our technique differs in that we are modelling the behavior of a real system rather than a simulation design space. Simulators can be inaccurate with respect to real systems [10, 11]. On the other hand, real hardware is a perfectly valid model of itself. Through careful measurement, the performance impact of changing a single microarchitectural feature such as branch prediction can be estimated accurately using the hardware itself to model to rest of the microarchitecture.

2.1.4 Estimating Behavior of Real Systems

Contreras and Martonosi use performance monitoring counters to develop a linear power model of the Intel XScale processor [8]. This approach can enable a technique capable of quickly estimating future power behavior and adapting to it at run-time. Our technique is similar in that it uses performance monitoring counters to develop a model of program behavior. However, we focus on modelling the behavior of one program at a time to get very precise information about the change in performance in response to a small change in the behavior of microarchitectural structures, i.e., our work concentrates on a much finer level of granularity, and we focus on performance instead of power.

2.2 Exploring Performance Variance to Reduce Write-Induced Interference

This section gives the background and recent work related to reducing write-induced interference by exploring performance variance.

2.2.1 DRAM Systems

The DDRx based memory system [25, 9] consists of one or more dual in-line memory modules (DIMMs) composed of multiple chips. Each chip is organized as multiple banks that can be operated in parallel. A memory rank is made up of a set of chips where chips in the same rank can be accessed simultaneously. In a DDRx memory module, each rank has a 64-bit data bus. Chips within a rank work in unison to return 64 bits per cycle. The memory channel is made up of one or multiple memory ranks. Ranks in the same channel share the same data bus. Modern multicore processors may have multiple channels.

A memory access includes both row access and column access [9]. An entire row of bits that contains the required data is brought into the row buffer during row access, then a column of this row buffer is selected according to the column address. Memory access requests may be row-buffer hit requests, row-buffer closed requests, or row-buffer conflict requests. A row-buffer hit request goes to a currently open row. Data can be accessed without activating the row buffer again. A row-buffer closed request goes to a row when there is no open row in the row buffer. The required row must be activated before the data in the row-buffer can be accessed. A row-buffer conflict request goes to a row other than the currently open row. Data in the currently open row must be written back first, then the required row must be activated before the data can be accessed. Thus, the access latency for row-buffer conflict/closed requests is significantly higher than for row-buffer hit requests.

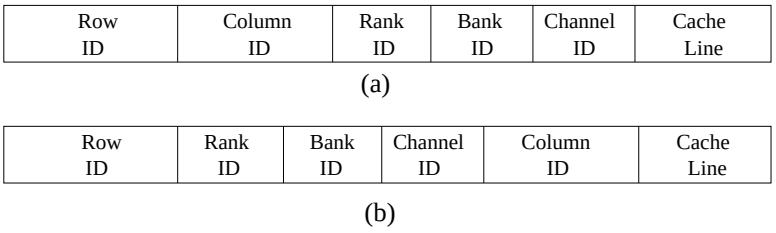


Figure 2.1: Address mapping scheme (a) cache line interleaving (b) page interleaving

2.2.2 Address Mapping Scheme

The memory address mapping scheme [93] [43] maps physical addresses to memory resources. Figure 2.1 shows the conventional cache line interleaving and page interleaving mapping schemes. In the cache line interleaving mapping scheme, consecutive cache lines are distributed to different rank/bank/channel combinations to maximize the parallelism of memory access, The page interleaving mapping scheme maps the lower order bits of the physical address into the column address to maximize the number of row buffer hits.

2.2.3 Memory Access Scheduling

Memory access scheduling [68] reorders memory references to improve memory performance. Much previous work [73, 78, 53, 37, 1, 52] focuses on improving memory efficiency by scheduling or relocating memory accesses to yield as many row hits as possible or servicing memory accesses in parallel. Shao *et al.* [73] propose a burst scheduling algorithm that schedules requests that hit in the same row buffer into a burst to increase row buffer hit rates and bus utilization. Sudan *et al.* [78] propose a page migration algorithm that collocates frequently accessed data in the same row buffer to increase row buffer hit rates in a multi-core system. Nesbit *et al.* [56] propose a fair queue scheduling algorithm for multi-core systems. The fair queue scheduling algorithm allocates to each thread a fraction of memory resources, thus reducing destructive interference and

improving fairness among threads. Mutlu *et al.* [53] propose a parallelism-aware batch scheduling technique for multi-core systems. Their technique first organizes memory requests into batches to ensure the fairness of service, then within each batch, requests are scheduled to maximize parallelism while at the same time minimizing the number of idle cores by using a shortest-job-first scheduling technique. Hur *et al.* [22] propose a scheduling algorithm that uses a state machine to make the next scheduling decision based on the past behavior. Ipek *et al.* [24] use a reinforcement-learning approach to learn the optimal memory scheduling policy according to past behavior.

2.2.4 LLC Writeback

Much previous work [78, 53, 37, 1, 52, 24] does not take into account the write interference problem. Eager writeback [42] is the first proposal that increases the visibility of the write buffer by using the LLC to reduce write-induced interference. Eager writeback writes back dirty cache blocks in the least-recently-used (LRU) position of the last-level cache sets whenever the bus is idle instead of waiting for the block to be evicted to reduce the memory traffic.

Stuecheli *et al.* [77] propose a virtual write queue (VWQ) technique. Their technique takes a fraction of the LRU positions in the LLC as the virtual write queue (also requiring LRU). Dirty cache blocks in the virtual write queue that target the same row buffer when mapping to the memory resource will be written back in a batch, therefore reducing write-induced interference. Chang *et al.* [55] propose a similar technique that writes back qualified dirty cache blocks in the LLC to improve the memory efficiency.

To reduce write-induced interference, both eager writeback and VWQ techniques issue write requests to DRAM when the rank is idle. Unfortunately, in their techniques, the memory controller does not have knowledge about how long the rank will remain idle. The write-induced penalty might be too long to be hidden by the short rank idle period.

Additionally, both eager writeback and VWQ techniques require that the LLC implement the costly LRU replacement policy.

2.2.5 Dead Block Prediction

Lai *et al.* [39] proposes last touch predictor that predicts the last touch cache blocks for core caches. The last touch predictor uses program counter (PC) traces to detect the last touch and invalidate the shared cache blocks to reduce cache coherence overhead. Several dead block predictors are proposed in previous work [35, 7, 44, 32]. The trace-based dead block predictor [39] can detect when a cache block is accessed for the last time based on the a given sequence of memory-access PCs. This predictor is used to prefetch data into dead blocks in the L1 data cache. Hu *et al.* [21] propose a time based dead block predictor that learns the number of cycles a block is live and predicts it dead if it is not accessed for twice that number of cycles. This predictor is used to prefetch into the L1 cache and filter a victim cache. Recent work proposes [32] sampling dead block predictor for LLC that predict the dead blocks in the LLC and replace them for useful cache blocks.

2.3 Exploring Performance Variance to Reduce Write Overhead of Non-Volatile Memory

This section gives the background and recent work related to reducing write overhead of NVM by exploring performance variance.

2.3.1 Emerging Non-Volatile Memory

In recent years, significant efforts and resources have been put on the researches and developments of emerging memory technologies that combine attractive features such as scalability, fast read/write, negligible leakage, and non-volatility. Multiple promising candidates, such as Phase-Change RAM (PCM), Spin-Torque Transfer RAM (STT-RAM), and Resistive RAM (RRAM), have gained substantial attentions and are being actively

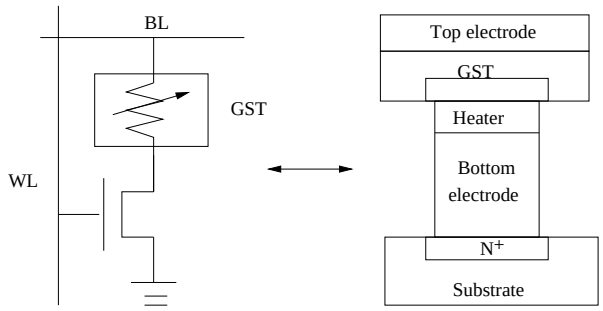


Figure 2.2: An illustration of Phase-change RAM (PCM) cell. The GST has two phases: the amorphous phase with high resistance and the crystalline phase with low resistance.

pursued by industry [66].

2.3.1.1 Phase Change Memory

Among various emerging memory technologies, Phase-Change RAM (PCM) is one of the most promising candidates for main memory because semiconductor companies have made dramatic R&D progress in recent years. For example, Samsung demonstrated an 8Gbit PCM memory chip recently [6], with CMOS-compatible embedded PCM (Hitachi and STMicro) [17, 58] have been demonstrated, paving the way for integrating these NVMs into traditional memory hierarchies. In addition, emerging 3D integration technologies [79] enable cost-effective integration of these NVMs with CMOS logic circuits. Compared with DRAM, the PCM [89] has high density, comparable read access time and reasonable write endurance which made it a promising alternatives to existing main memories. Thus, many innovative memory architectures using PCM as main memory have emerged in the last several years [61][89][64, 62, 94, 41].

In a PCM memory cell, the storage node is based on a chalcogenide alloy (typically GeSBTe (GST) material), as shown in Figure 2.2. The resistance differences between an amorphous (high resistance) and crystalline (low resistance) phase of chalcogenide-based material indicate the stored value as “1” and “0”, respectively. Writing a bit to the PCM

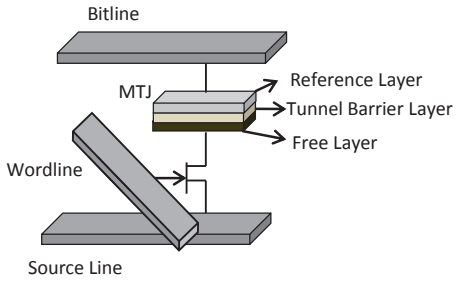


Figure 2.3: An illustration of STT-RAM cell

cell is done through *set* and *reset* operations: for set operations, the phase-change material is crystallized by applying an electrical pulse that heats a significant portion of the cell above its crystallization temperature. In reset operations, a larger electrical current is applied and then abruptly cut off to melt and then quench the material, leaving it an amorphous state. Compared to charge-based SRAM/DRAM, PCM intrinsically takes longer and consumes more energy to overwrite the existing data which could result in performance degradation and high energy consumption.

2.3.1.2 STT-RAM Technology

STT-RAM is the second generation of MRAM. As shown in Figure 2.3, it uses a Magnetic Tunnel Junction (MTJ) as an information carrier. Each MTJ consists of two ferromagnetic layers: a reference layer and a free layer. A tunnel barrier layer is sandwiched between the two ferromagnetic layers. The reference layer has a fixed direction while the free layer can change its direction by passing write current. The relative direction of the reference and free layers are used to represent a memory bit. If the layers have the same direction, the MTJ resistance is low which indicates state 0; otherwise, the MTJ resistance is high which indicates state 1.

A read operation is performed by turning on the access transistor and applying a small voltage difference between the bitline (BL) and source line (SL) to sense the MTJ resis-

tance. A write operation is performed by establishing a high voltage difference between BL and SL with a positive voltage difference for writing 1 and a negative voltage difference for writing 0.

2.3.2 Related Work on Mitigating PCM Write Overhead

Many researchers propose techniques to mitigate PCM write latency and energy overhead. For example, Lee et al. [41] propose to use narrow PCM buffers to mitigate high-energy PCM writes. Write cancellation and Write pausing [61] has been proposed to prioritize read requests over write requests by adaptively cancel or pause the service of write requests when read requests are waiting for service. Qureshi et al. [60] exploit asymmetry in write times for SET and PRESET operation of PCM devices and propose to initiate a PreSET request for a memory line as soon as data written into the LLC, thereby incurring low write-induced interference.

Hybrid main memory architecture has been proposed to leverage the benefits of both DRAM and PCM technologies. Qureshi *et al.* [64] propose a main memory system consisting of PCM storage coupled with a DRAM write buffer, so that it has the latency benefits of DRAM and the capacity benefits of PCM. Yoon *et al.* [91] propose to improve the hybrid performance by caching the frequent row buffer miss requests in DRAM. Ramos et al. [65] propose a page ranking and migration policy for the hybrid PCM and DRAM based main memory.

Write endurance poses another severe challenge in PCM memory design. The cells suffering from more frequent write operations will fail far sooner than the rest. A read-before-write operation [30] can help identify such redundant bits and cancel those redundant write operations to save energy and reduce impact on performance. A range of *wear-leveling techniques* [62, 94, 41] for PCM have been examined to increase the life time of PCM-based main memory architectures.

Most of these proposed techniques mitigate the write overhead of PCM by doing optimizations at the main memory level. They either use new memory architectures or add a new operation to PCM. However, write requests sent from the LLC remain unchanged. Zhou *et al.* [94] take the first step to exploit the LLC partitioning and replacement policy by considering the negative impact of writeback requests. They propose to partition the shared LLC among multi-core by taking into account the writeback penalty. Fedorov *et al.* [14] propose to divide the LRU stack into ”High-hit” and ”Low-hit” partitions. On a cache replacement request, the technique gives higher priority to evict clean block in ”Low-hit” part than dirty block in ”Low-hit” part. However, both of the techniques require a cache replacement policy with distinct recency levels, such as Least-Recently-Used (LRU) replacement policy. For some cheap replacement policy, such as Not-Recently-Used (NRU) and Random, these techniques can not be applied to them.

2.3.3 Related Work on Mitigating Write Overhead of STT-RAM

Many prior papers [29, 81, 46] focus on mitigating write overhead of an STT-RAM cache. Jog *et al.* [29] propose to improve the write speed of STT-RAM-based LLC by relaxing its data retention time. However, that technique requires large capacity buffers for a line level refreshing mechanism to retain reliability. Mao *et al.* [46] propose prioritization policies for reducing the waiting time of critical requests in the STT-RAM-based LLC. However, the technique increases the power consumption of LLC. Recently, researchers propose hybrid SRAM and STT-RAM techniques [79, 5, 26, 4, 80] for improving LLC efficiency. Sun *et al.* [79] take the first step introducing the hybrid cache structure. That technique uses a counter-based approach for predicting write-intensive blocks. Write-intensive blocks are placed in SRAM ways for reducing write overhead to STT-RAM portion. However, that technique is optimized only for core-write operations. It cannot reduce the prefetch-write and demand-write operations to STT-RAM. Jadidi *et al.* [26] propose

reducing write variance of STT-RAM lines by migrating frequently written cache blocks to other STT-RAM lines or SRAM lines. However, frequently migrating data between cache lines incurs significant performance and energy overhead. Chen *et al.* [5] propose a combined static and dynamic scheme to optimize the block placement for hybrid cache. The downside of the technique is it requires the compiler to provide static hints for initializing the block placement.

3. EXPLORING PERFORMANCE VARIANCE TO DEVELOP PERFORMANCE MODEL*

Modern microprocessors have many microarchitectural features. Quantifying the performance impact of one feature such as dynamic branch prediction can be difficult. On one hand, a timing simulator can predict the difference in performance given two different implementations of the technique, but simulators can be quite inaccurate. On the other hand, real systems are very accurate representations of themselves, but often cannot be modified to study the impact of a new technique.

We develop a performance model for branch prediction using real systems [83][*]. The technique perturbs benchmark executables to yield a wide variety of performance points without changing program semantics or other important execution characteristics such as the number of retired instructions. By observing the behavior of the benchmarks over a range of branch prediction accuracies, we can estimate the impact of a new branch predictor by simulating only the predictor and not the rest of the microarchitecture. We call this technique *Program Interferometry* based on its similarity to astronomical optical interferometry.

Figure 3.1 demonstrates the potential of program interferometry. Each of the 100 points represents an executable with a different code reordering of the SPEC CPU 2006 benchmarks `400.perlbench` and `471.omnetpp` running on `ref` inputs. Performance monitoring counters enable collecting the cycles-per-instruction (CPI) and branch mispredictions per 1000 instructions (MPKI) of each run. The plot shows actual measurements as well as a least-squares regression line estimating the linear relationship between MPKI and CPI. They also show 95% confidence intervals and 95% prediction intervals.

[*] ©2011 IEEE. Reprinted, with permission, from Zhe Wang; Daniel A. Jiménez, "Program Interferometry," Workload Characterization (IISWC), 2011 IEEE International Symposium, Nov. 2011

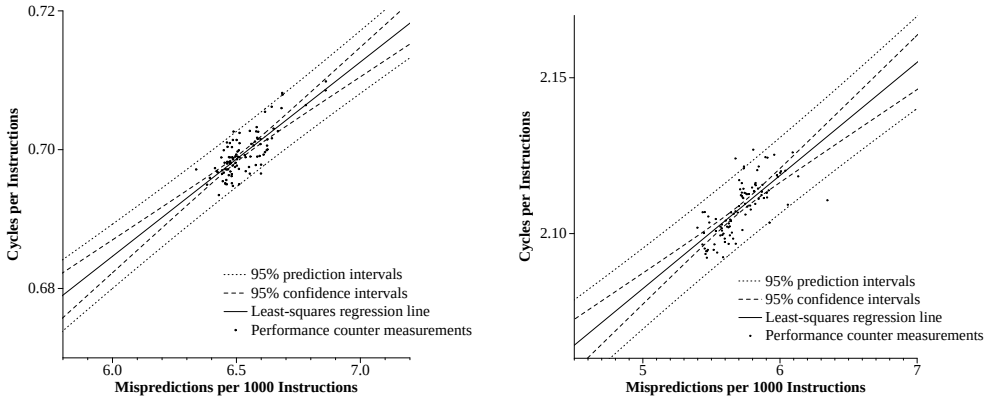


Figure 3.1: Performance changes with branch prediction accuracy for 400.perlbench and 471.omnetpp.

As an example of the usefulness of program interferometry to branch predictor design, linear regression allows us to make the following predictions for `400.perlbench` with 95% probability:

1. A perfect branch predictor would yield a CPI of 0.517 ± 0.029, an improvement of $26.0\% \pm 4.2\%$.

2. Halving the average MPKI from 6.50 to 3.25 would improve CPI by $13.0\% \pm 2.2\%$ from 0.70 to 0.61 ± 0.022.

3. A 10% improvement in CPI due to branch prediction improvement would require a 38% reduction in mispredictions.

3.1 Motivation

Astronomers used the earliest telescopes to view the universe from a single point of view. Their observations were dim and blurry, limited by the tiny amount of light that their small telescopes could collect and the effects of atmospheric turbulence. However, in recent years, astronomers have used a technique called optical interferometry to combine

the observations of many telescopes from many different points of view to obtain images with a much higher resolution [2].

Similarly, by sampling and observing many points in a space of program performance, we can get a much better understanding of program behavior. *Program Interferometry* is based on perturbing placement of code and data. Many executable versions of a program are produced by pseudo-randomly re-ordering procedures and objects files. Similarly, the memory allocator places objects pseudo-randomly on the heap. A given random placement of code and data can be repeated by using the same key for the pseudo-random number generator so that runs are reproducible. Each code and data placement is semantically equivalent, but because the instruction addresses are different, different conflicts will arise among microarchitectural structures such as the branch predictor and instruction cache [54]. The situation is isomorphic to one in which we keep the code and data placement constant, but change the hash functions for microarchitectural structures. Thus, we may measure the performance impact of changing these structures.

An alternative would be to use cycle-accurate simulators with best-guess estimates of future microarchitectural structures. However, it is not clear to researchers what future microarchitectures will be like. The return of Intel from the more complex Netburst to the simpler P6-inspired Intel Core 2 is an example of this uncertainty. The trend in 2001 was toward deeper and deeper pipelines, so contemporaneous branch prediction papers simulating pipeline depths of up to 40 were way off the mark. Also, simulators are notoriously inaccurate with respect to real systems because many of the details of real systems are difficult or impossible to model or even to know about [10]. Earnest efforts at simulation are subject to bugs that can invalidate research conclusions made with them [11]. Thus, demonstrating that a new branch predictor (or other optimization) can improve an existing microarchitecture is another way to have confidence in that optimization's contribution to unknown future microarchitectures.

3.2 Description

In this section we describe the technique of program interferometry. The basic idea is to execute code under many different reorderings, causing a wide variance in performance due to different accidental collisions in microarchitectural structures. By measuring the resulting adverse microarchitectural events, we can build a performance model for the program and microarchitecture.

3.2.1 Instruction Addresses in Microarchitectural Structures

Program interferometry exploits the fact that several microarchitectural structures use a hash of instruction and data addresses. For example:

1. A 128-set instruction cache with 64 byte blocks would likely use bits 6 through 12 of the instruction address as the set index.

2. A branch direction predictor might index a table of counters using a combination of branch history and branch address bits.

3. A branch target buffer (BTB) or indirect branch predictor would use lower-order bits of the branch address to index a table of branch targets.

Sometimes addresses will accidentally collide in some microarchitectural structure. For example, conflict misses in the instruction cache occur when the number of blocks mapping to a particular set exceeds the associativity of the cache. Although this phenomenon has been studied in academic research, most compilers do not optimize to protect against these kinds of conflicts.

Compiler writers are aware of uses of instruction addresses and write compilers to exploit these uses. For instance, a common heuristic is to align the target of a branch on a boundary divisible by the number of bytes in a fetch block to allow the fetch beginning at that target to read the maximum number of instruction bytes in one cycle.

3.2.2 A Wide Range in Performance

These accidental conflicts result in adverse microarchitectural events such as branch mispredictions, cache misses, BTB misses, etc. A particular code and data placement will result in a particular number of accidental collisions with a particular impact on performance. A different layout will result in a difference impact on performance. By exploring a wide range of layouts, we can force a wide range of adverse performance events to take place and explore a wide range of performances.

3.2.3 Causing Collisions

To generate many random but plausible code layouts, we extend the technique of Mytkowicz *et al.* [54], i.e., object-file reordering. We compile each benchmark once, lowering it to assembly language files. Then we produce executables with hundreds of different code reorderings. We then reorder procedures within assembly files, assemble the files, and then link with different randomly-generated order of the object files. The linker lays code out in the order in which it is encountered on the command line, so each random procedure and object-file ordering results in a different code layout.

We execute each resulting executable five times, collecting performance monitoring counter information such as number of instructions committed, number of branch mispredictions, number of clock cycles, etc. We take the performance monitoring counter statistics that gave the median performance. Details of our infrastructure are given in Section 3.3

3.2.4 Making Predictions

Once the performance monitoring counter information has been collected, we can begin using statistical tools to build a performance model. We use least-squares linear regression to estimate the relationship between various microarchitectural events and perfor-

mance outcomes. For instance, for the plots in the Introduction, we found a regression line of CPI $= 0.02799 *$ MPKI $+ 0.51667$. That is, we use the MPKI to predict the CPI. For a range of MPKI values, we also 95% computed confidence intervals and prediction intervals. A 95% confidence interval has a 95% chance of containing the true regression line, i.e., of all the data collected, the line that best illustrates the linear relationship between CPI and MPKI has a 95% chance of being in that confidence interval [48]. The larger 95% prediction interval has a 95% chance of containing all of the observations (i.e. CPIs) that would be encountered in a given domain (i.e. set of MPKIs).

3.2.5 When Things Go Wrong

Some benchmarks do not give a wide range in performance under code reordering, or the range in performance cannot be explained by events related to the instruction address. For each type of prediction we would like to make for a given benchmark, we first determine whether there is significant correlation between the dependent variable and independent variables. We use Student's t-test with the null hypothesis "there is no correlation," i.e., if we cannot reject the null hypothesis, then we cannot say whether there is any correlation between the events observed [48]. For the 23 SPEC CPU2006 benchmarks that compiled in our infrastructure, estimating CPI with MPKI, the null hypothesis was rejected at $p = 0.05$ or less for 20 benchmarks. In other words, for the great majority of the benchmarks, we determined that there was at least a 95% chance that program interferometry found significant correlation between CPI and MPKI. For the other benchmarks, there was not enough range of MPKI to predict CPI.

3.3 Experimental Methodology

This section describes the experimental methodology used for the interferometry technique.

3.3.1 Compiler

We use the Camino compiler infrastructure [20]. This system is a post-processor for the Gnu Compiler Collection (GCC) version 4.2.4. C, C++, and FORTRAN programs are compiled into assembly language, the assembly language is instrumented by Camino, and the result is assembled and linked into an executable. Camino features a number of profiling passes and optimizations, but for this study we implement and use only the profiling and instrumentation pass described below. All of the executables produced for this study target the x86_64 instruction set.

3.3.2 Benchmarks

We use the SPEC CPU 2006 benchmarks for this study. Of the 29 benchmarks, 23 compile and run without errors with our compiler infrastructure. These benchmarks are listed in the x-axes of several graphs in later sections.

3.3.3 Generating Random Code Reorderings

Each benchmark is compiled once from C/C++/FORTRAN into assembly. The Camino infrastructure is then used to reorder procedures within files and then assemble the files into object code files. The resulting object files are randomly reordered and linked to make an executable. Camino accepts a seed to a pseudo-random number generator to generates pseudo-random but reproducible orderings of procedures and object files.

3.3.4 System

We perform our study using four Dell systems with identical configurations running the 64-bit version of Ubuntu Linux 8.04 Server and a custom compiled kernel with performance monitoring counter support. Each system contains two quad-core Intel Xeon E5440 processors. The Intel processor 5400 Series are based on 45nm Enhanced Intel Core Microarchitecture. Each processor has 16GB of SDRAM and 12MB second level

cache. Each core in the Intel Xeon E5440 processor has 32KB instruction cache and a 32KB data cache. The branch predictor of the Intel Xeon E5440 is not documented, but through reverse-engineering experiments we have determined that it is likely to contain a hybrid of a GAs-style branch predictor and a bimodal branch predictor [90, 75, 13].

3.3.5 Running with Performance Monitoring Counters

We measure a number of performance monitoring counters using the `perfex` command found in the PAPI performance monitoring package [50]. The Intel Xeon processor allows up to two user-defined microarchitectural events to be counted simultaneously. We are interested in more than two events, so we make multiple runs of each benchmark to collect all of the desired counters. We group the counters into three sets of two. For each set we run each benchmark five times and take the measurements given by the run with the median number of cycles. Only the microarchitectural events that occur while user code is running are counted, thus the impact of system events is minimized. We collect the following statistics: 1) Retired branches mispredicted, 2) Retired x86 instructions excluding exceptions and interrupts, 3) L1 instruction cache misses, 4) L2 cache misses, and 5) Elapsed clock cycles.

From these counters, we can derive other statistics such as cycles-per-instruction (CPI), branch mispredictions per 1000 instructions (MPKI), various cache miss rates, etc.

Although each system is configured identically and each core has the same microarchitecture, we use the Linux `taskset` command to make sure that each benchmark always runs on the same core to eliminate the effect of possible slight differences among the cores. Each run is performed on an otherwise quiescent system with as many system services stopped as possible without compromising the ability to access remote files and log in remotely. Stack address randomization, a security feature that resists stack-smashing attacks, is disabled to minimize performance variance not due to code placement.

3.3.6 Simulation

We develop several branch predictor simulators. We implement these as a tool in Pin [45]. We then run pin on the same executables that we run natively. Our Pin tool instruments each branch with a callback to code that simulates a set of branch predictors. The tool counts the number of branches executed and the number of branches mispredicted for each predictor simulated.

3.3.7 Timing Concerns

Many of the SPEC CPU 2006 benchmarks run for over 30 minutes on the first `ref` input. For this study, we have executed each of the 23 benchmarks at least 100 times on a set of 4 computers. To facilitate this study, we instrument the benchmarks such that under native execution they run for up to approximately two minutes each. To do this, we implement a two-pass profiling and instrumentation pass in the Camino compiler. The first pass inserts instrumentation that collects information about each procedure. The benchmark is allowed to run for two minutes. Then the collected information is analyzed to find a procedure with a low dynamic count that is also executed near the end of the two-minute run. The second pass of the compiler instruments only that procedure such that when it is executed the same number of times as before, the program is ended. The first instrumentation has low overhead, thus the resulting executable runs for approximately two minutes. The second instrumentation affects a low-frequency procedure and takes two x86 instructions, thus it has negligible overhead. All of the executables in this study are compiled from this second instrumentation, or are from benchmarks that naturally run for less than two minutes. Because we are counting procedures and not elapsed time, each run of a benchmark executes the same number of user instructions.

3.4 Estimating Performance by Counting Microarchitectural Events

This section shows the potential of program interferometry to predict performance. We develop and evaluate regression models for a number of benchmarks using several characteristics of program behavior such as branch prediction and cache misses.

3.4.1 Assigning Blame

Code reordering can elicit a wide range of CPIs for our benchmarks. Here, we determine how much blame to place on certain microarchitectural structures for the performance variance. We focus on what we believe to be the microarchitectural events most likely to be affected by code placement: 1) Branch mispredictions. Conditional branch predictors use the address of an instruction to index one or more tables. Branches may conflict with one another in these tables leading to *aliasing* [49] causing branch prediction accuracy to suffer. 2) L1 instruction cache misses. The Intel Xeon Core has a 32KB 8-way set associative instruction cache. If nine or more frequently used blocks map to the same set, there will be frequent cache misses. 3) L2 cache misses.

We also use multi-linear regression to develop a combined model that takes into account all three of these events in the hope that a combined model will be more accurate than using one of the observations by itself.

Using r^2, the coefficient of determination, we can determine what portion of performance is due to a particular microarchitectural event. Figure 3.2 shows the cumulative r^2 for each of the three events, as well as r^2 for the combined regression model. On average, 27% of the CPI difference between different code reorderings can be explained by branch misprediction.Some benchmarks are more sensitive; for instance, 84.2% of the CPI variance of `462.libquantum` is due to branch mispredictions.

The average bar for the combined model does not reach exactly the same height as that of the sum of the three measurements. This is because the three measurements are not

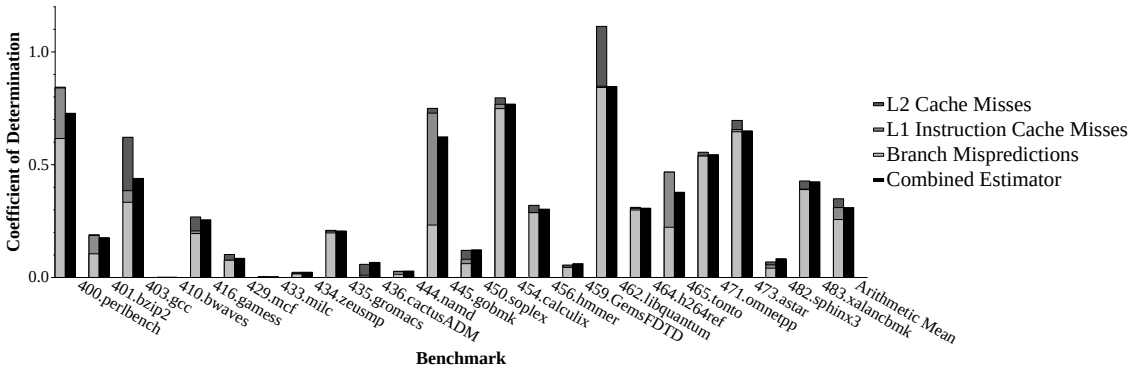


Figure 3.2: Coefficient of determination showing how much of each type of event accounts for overall performance.

altogether independent of one another; for instance, in some cases, a branch misprediction might cause an L1 cache event, sometimes causing cache pollution and other times causing prefetching. It must be emphasized that the correlation we report between microarchitectural events and performance is with respect to code ordering. Other changes to the execution environment would show other correlations.

3.4.2 Establishing Statistical Significance

Clearly many benchmarks' performance show correlation with microarchitectural events. However, we must ask whether the correlation is statistically significant. We use Student's t-test to determine statistical significance. For each of the three measurements as well as the combined model we attempt to reject the null hypothesis that there is no correlation. The value $p \leq 0.05$ for the t-test is traditionally accepted as proof of statistical significance. For the combined model we use the F-test $p \leq 0.05$ instead of the t-test, as the t-test is appropriate for single-variable linear regression models.

3.4.3 Number of Samples

For some benchmarks, the effect of code reordering on performance is harder to detect than for others. To establish statistical significance for as many benchmarks as possible,

we sample a number of code reorderings in multiples of 100 until the benchmark is able to reject the null hypothesis, or until by inspection we determine that the benchmark is unlikely to reject the null hypothesis with a much larger number of samples. Most benchmarks reject the null hypothesis within the first 100 samples. Some take 200 samples, and a few require 300 samples. We do not discard any data when building or testing our regression models: we use the data from each reordering.

	Event			
Benchmark Name	**Branch MPKI**	**L1 I-Cache Misses**	**L2 Cache Misses**	**Combined Estimator**
400.perlbench	yes	yes	-	yes
401.bzip2	yes	yes	-	yes
403.gcc	yes	yes	yes	yes
410.bwaves	-	-	-	-
416.gamess	yes	-	yes	yes
429.mcf	yes	-	-	yes
433.milc	-	-	-	-
434.zeusmp	yes	-	-	-
435.gromacs	yes	-	-	yes
436.cactusADM	-	-	yes	yes
444.namd	yes	-	-	yes
445.gobmk	yes	yes	-	yes
450.soplex	yes	yes	yes	yes
454.calculix	yes	-	-	yes
456.hmmer	yes	-	-	yes
459.GemsFDTD	yes	-	-	-
462.libquantum	yes	-	yes	yes
464.h264ref	yes	-	-	yes
465.tonto	yes	yes	-	yes
471.omnetpp	yes	-	-	yes
473.astar	yes	-	yes	yes
482.sphinx3	yes	-	-	yes
483.xalancbmk	yes	-	-	yes

Table 3.1: "Yes" means that the null hypothesis of "no correlation" is rejected with $p \leq 0.05$, i.e., with 95% probability, the given measurement is correlated with CPI.

Table 3.1 shows "yes" for each combination of measurement and benchmark where the null hypothesis can be rejected with at most $p = 0.05$, i.e., with 95% probability there is correlation between CPI and the measurement for that benchmark.

3.4.4 Blame the Branch Predictor

Of the 23 benchmarks, 20 show significant correlation between CPI and branch prediction. No other measurement consistently shows statistically significant correlation with CPI. The combined estimator does not increase the number of benchmarks showing significant correlation, and indeed two benchmarks that show significant linear correlation with MPKI through the t-test fail to reject the null hypothesis for the F-test with the combined model and multiple linear regression. Thus, in this paper we focus our attention on branch prediction.

3.4.5 A Linear Performance Model

We use least-squares linear regression to derive branch prediction performance models for the Average Model and each of the benchmarks that passed the hypothesis testing phase. For each benchmark, we find the best fit of the observed data to a regression line $y = mx + b$ where y is CPI and x is MPKI. The slope (m) gives the cost for performance of one additional MPKI and the y-intercept (b) gives the predicted average CPI for perfect branch prediction, i.e. 0 MPKI.

We also derive 95% confidence intervals and 95% prediction intervals for the regression lines. Figure 3.1 in the Introduction shows the regression line and intervals for `400.perlbench` and `471.omnetpp`. The confidence interval has a 95% chance of containing the true regression line for the data observed. The much wider prediction interval has a 95% chance of containing future observations. Thus, we can be 95% sure that the CPI of `471.omnetpp` with perfect branch prediction would be between 1.86 and 1.94. Table 3.2 shows the slopes and y-intercepts found by linear regression for each

benchmark. It also shows the high and low prediction intervals for perfect prediction.

Benchmark	Slope	y-intercept	Low	High
`400.perlbench`	0.028	0.517	0.488	0.546
`401.bzip2`	0.017	0.596	0.485	0.708
`403.gcc`	0.028	1.839	1.796	1.882
`416.gamess`	0.041	0.548	0.519	0.577
`429.mcf`	0.019	4.675	4.531	4.819
`434.zeusmp`	0.373	0.863	0.813	0.913
`435.gromacs`	0.020	0.811	0.795	0.827
`444.namd`	0.033	0.620	0.551	0.689
`445.gobmk`	0.019	0.643	0.515	0.771
`450.soplex`	0.016	1.822	1.741	1.904
`454.calculix`	0.023	0.461	0.460	0.463
`456.hmmer`	0.041	0.203	0.032	0.375
`459.GemsFDTD`	0.516	1.229	1.189	1.269
`462.libquantum`	0.022	1.432	1.431	1.433
`464.h264ref`	0.032	0.466	0.451	0.481
`465.tonto`	0.027	0.632	0.617	0.647
`471.omnetpp`	0.036	1.901	1.860	1.941
`473.astar`	0.022	2.373	2.289	2.456
`482.sphinx3`	0.036	0.916	0.798	1.034
`483.xalancbmk`	0.029	1.914	1.881	1.947

Table 3.2: Least-squares regression model relating branch prediction to performance. Shows high and low prediction intervals for perfect prediction i.e. 0 MPKI.

3.5 Estimating Branch Prediction Performance

This section presents results of simulation experiments using program interferometry to predict the performance impact of changes to the branch predictor. We use the performance model derived with program interferometry to predict the performance given by several predictors.

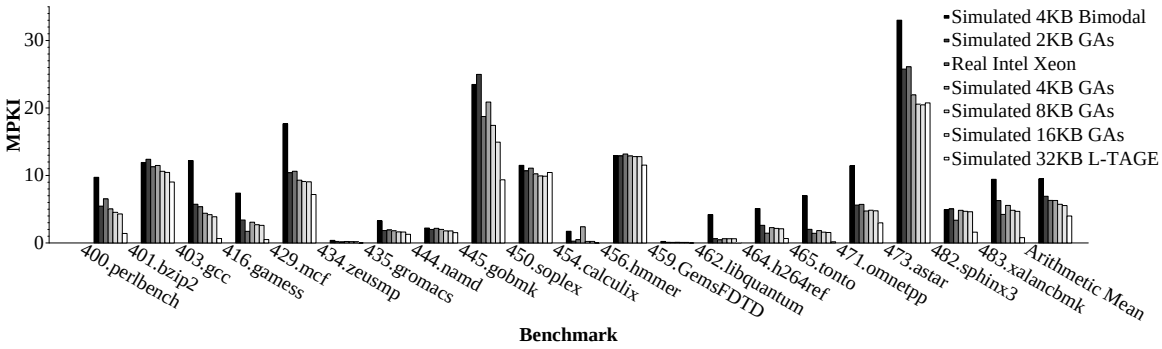


Figure 3.3: MPKI of real and simulated branch predictors.

3.5.1 Branch Prediction Simulation

The Pin tool instruments each branch with a callback to code that simulates a set of branch predictors. The tool counts the number of branches executed and the number of branches mispredicted for each predictor simulated.

3.5.2 Impact of Mispredictions on Performance

We explore only those benchmarks that were demonstrated in the previous section to be suitable for program interferometry (i.e. those with "yes" in Table 3.1). Figure 3.3 shows the average MPKI for various branch predictors simulated with Pin as well as the average MPKI from the real Intel Xeon branch predictor. These data are averaged over 100 different pseudo-randomly generated code reorderings. For each benchmark, these are the same first 100 reorderings used for the performance monitoring counter measurements. Pin runs only once for each reordering; since we control the initial conditions of the simulator and Pin is not affected by system-level events, there is no variance in the simulation result. We simulate GAs branch predictors [90] ranging in size from 2KB to 16KB to explore the effect of decreasing or increasing the hardware budget for the branch predictor. The average MPKI over all benchmarks and code reorderings for the real branch predictor is 6.306, compared with 5.729 for a simulated 8KB GAs predictor. A 16KB simulated

GAs branch predictor yields 5.542 MPKI.

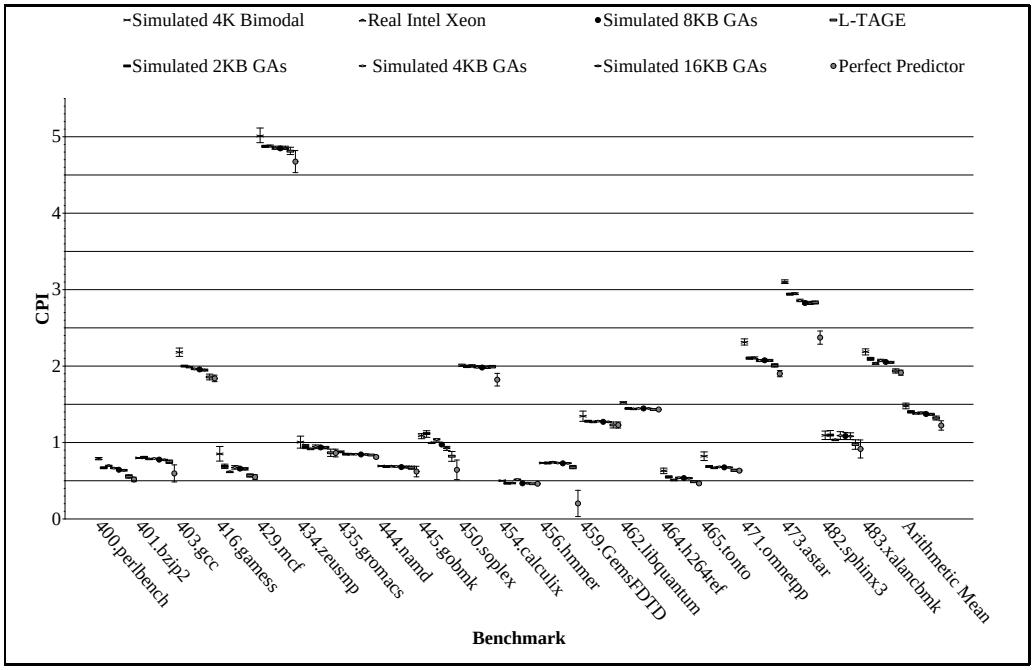


Figure 3.4: Predicted CPI of real and simulated branch predictors.

Figure 3.4 shows the predicted CPI for the various branch predictors as well as a perfect (0 MPKI) predictor using the performance model derived in the previous section. Each point in the graph shows a marker superimposed on error bars giving the 95% prediction interval for the benchmark's regression model. For the real branch predictor, the error bars indicate the tighter confidence interval since the data are observations and not predictions. Most of the benchmarks have reasonable prediction intervals even for the perfect predictor.

3.5.2.1 Perfect Branch Prediction

The real branch predictor yields an average CPI of 1.387 ± 0.012. The estimated CPI for perfect prediction is 1.223 ± 0.061. Thus, the performance improvement going from the current predictor to perfect prediction would be between 7% and 16%, with an average

of 11.8%.

3.5.2.2 Academic Predictor

The L-TAGE branch predictor is currently the most accurate branch predictor in the academic literature [72]. We simulate this predictor using Pin and estimate the CPI yielded using our regression models. On average, L-TAGE yields 3.995 MPKI, compared with 6.306 MPKI for the real Intel predictor, an improvement of 37%. Our regression model estimates that this predictor would yield an average 1.320 ± 0.03 CPI, an improvement of between 2.4% to 6.8%, with an average of 4.8%. Several different sized GAs predictors are also shown. GAs predictors are simple global predictors uses in current microprocessors. The accuracy of GAs improves as its size grows.

3.5.2.3 Practical Concerns

We do not suggest that Intel should or should not replace their predictor with some other predictor. There are other concerns such as access latency to the prediction table that would guide such a decision. Our tool allows exploring the performance impact of hypothetical predictors before the decision is taken to spend design effort to accommodate them in a microarchitecture. For instance, it is possible that Intel could spare an extra 24KB for the L-TAGE branch predictor, but that the access latency and design complexity for such a structure might exceed the time allowed for branch prediction resulting in an unacceptable pipeline bubble. The design effort to include latency mitigating techniques [28] might not be worth the improvement in performance or delay in time to market. Nevertheless, our tool allows a quick way of evaluating many potential branch predictors for a given microarchitecture.

4. EXPLORING PERFORMANCE VARIANCE TO REDUCE WRITE-INDUCED INTERFERENCE*

Memory access latency is a major performance bottleneck. A LLC miss can stall the pipeline and require hundreds of cycles of delay. Memory write requests compete with read requests for the available memory resources, increasing the average service time of read requests. When a write request is in service, subsequent read requests to the same rank must wait the completion of the write as well as the bus turnaround time. This write-induced interference has a significant impact on system performance [55] [86][*]

Figure 4.1 shows read latency normalized to conventional writeback on a quad-core processor for perfect writeback [85][*]. Perfect writeback assumes memory write access does not cause any interference to read access, which is the optimal case. we can see the read latency for perfect writeback is 74.6% of conventional writeback. Thus, 25.4% of the read latency suffered by conventional writeback is caused by write-induced interference. Therefore, the write-induced interference significantly degrade the system performance.

There are two aspects to reducing write-induced interference. First, we must consider when to schedule the write requests [77]. System performance is sensitive to memory read latency, so write requests should be scheduled to have minimal interference with read requests. Second, we must consider how to schedule the write requests. Write requests should be scheduled in a way that they can be serviced by DRAM efficiently.

In a conventional writeback policy, dirty cache blocks are sent to the write buffer when

[*]©2012 Association for Computing Machinery, Inc. Reprinted by permission, from Zhe Wang, Samira M. Khan, and Daniel A. Jiménez. 2012. Rank idle time prediction driven last-level cache write-back. In Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC '12). ACM, New York, NY, USA, 21-29. DOI=10.1145/2247684.2247690 http://doi.acm.org/10.1145/2247684.2247690.

[*]©2012 IEEE. Reprinted, with permission, from Zhe Wang; Samira, M. Khan; Daniel, A. Jiménez, "Improving writeback efficiency with decoupled last-write prediction," Computer Architecture (ISCA), 2012 39th Annual International Symposium, June 2012

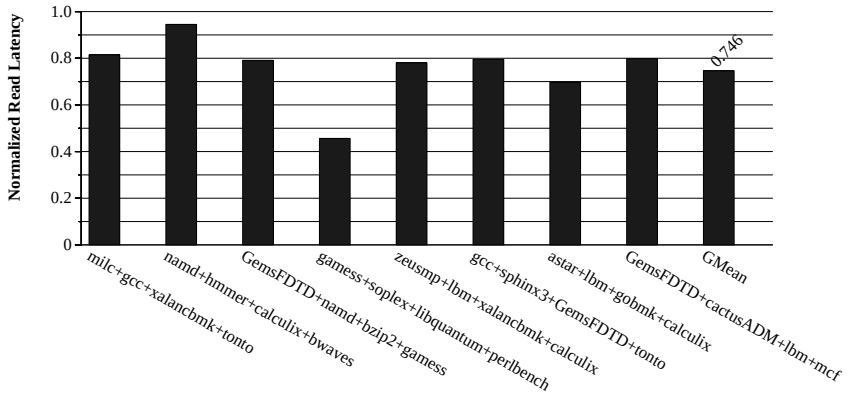


Figure 4.1: Read latency using conventional writeback and perfect writeback techniques in quard-core processor

they are evicted from the LLC. Write requests in the write buffer are scheduled for service according to the buffer management policy. However, the write buffer only has a small number of entries due to design complexity and power efficiency, limiting the ability to schedule high locality write requests as well as the possibility to flexible adjust read/write priority.

LLC writeback techniques [42, 77] have been proposed to expand write resources using near least recently used (LRU) position of the LLC. Eager writeback [42] sends dirty cache blocks in the LRU position to DRAM for service when the rank is idle, thus re-distributing write requests. The virtual write queue (VWQ) [77] issues scheduled writebacks from near the LRU position in the LLC to improve writeback efficiency. To reducing write-induced interference, both eager writeback and VWQ techniques issue write requests to memory when no read requests target the same rank. Unfortunately, these techniques have no knowledge about when the next read request will come. If a read request comes soon after a write request is issued, the write will still impose large penalty on the read. Additionally, the previous LLC writeback techniques depend on the recency levels of LLC replacement policy. Thus, these techniques can not work with LLC replacement policies

with no distinct recency levels, such as not recently used (NRU) and random replacement policy.

The memory access pattern exhibits significant variance. Memory read requests tend to come in bursts. The DRAM can busy service the memory requests for a while then idle for a while. Additionally, in modern DDRx-based systems, multiple memory controllers and multiple ranks are used to service memory requests in parallel. Due to workload characteristics and load imbalance, some ranks often have idle cycles while the application is running.

By exploring the memory access variance, we develop the prediction driven last-level cache writeback (LLC) technique. We propose a rank idle time prediction driven LLC writeback technique. This technique send the scheduled writebacks into the DRAM during the long rank idle period, thus minimizing the delay it caused to the following read requests. We also propose a last-write prediction driven LLC writeback technique. The technique improves the writeback efficiency by increasing the write scheduling space. It is completely decoupled with LLC replacement policy , thus it can work with any LLC replacement policy. Our techniques significantly reduce the write-induced interference.

4.1 Rank Idle Time Prediction Driven Last-Level Cache Writeback

4.1.1 Description

The rank idle time prediction driven LLC writeback technique fills DRAM idle rank cycles with scheduled writeback requests. The technique predicts when there will be long stretches of idle rank cycles and issues scheduled writeback requests in those stretches of times such that significant interference with subsequent read requests in the same rank will not occur. This technique contrasts with eager writeback, which has no knowledge about how long the bus idle cycles might last and can issue writeback requests in short idle cycles that still cause large writeback penalties to subsequent read requests.

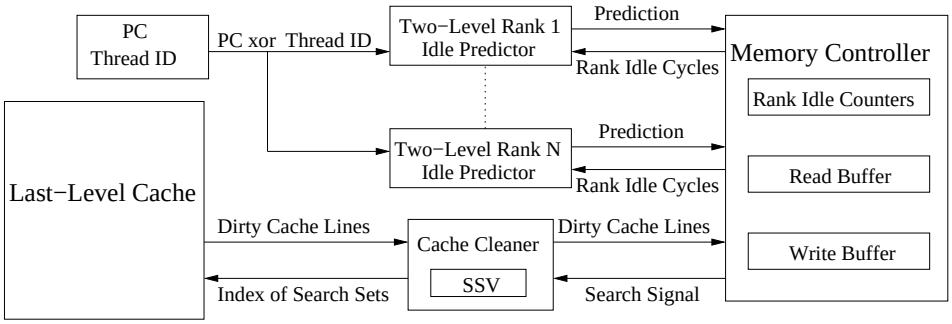


Figure 4.2: System structure

Figure 4.2 illustrates the structure of our technique. A two-level predictor is used to predict long stretches of idle rank cycles for a given rank. The two-level predictor is composed of two predictors making predictions at different times to predict whether there will be significant idle rank time for a particular rank. Each rank has one two-level predictor. Thus, the number of two-level rank idle predictors for a DRAM system is equal to the number of ranks this system has. A sequence of scheduled dirty cache blocks that are generated by the Cache Cleaner [77] are written back during a predicted long idle period.

4.1.2 Address Mapping

The baseline address mapping scheme we use in our system is the page interleaving scheme. The cache line interleaving mapping scheme maps consecutive physical addresses to different channels and ranks. This mapping scheme will cause read requests to go to different ranks frequently and produce fragmented short idle cycles which might be too short to compensate for large write-induced interference.

Compared with the cache line interleaving mapping scheme, the baseline mapping scheme tends to collect small chunks of idle rank cycles into large runs. Thus, the long idle cycles can be used to write back dirty cache blocks. The service of write requests and

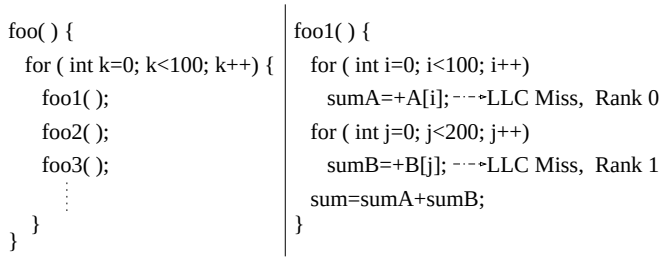


Figure 4.3: Example of memory access

read requests are isolated from one another during this long idle time, increasing the bus utilization and reducing the write-induced interference. Our technique prefers to map the rank ID and channel ID bits higher than the row ID bits.

4.1.3 Two-Level Rank Idle Time Predictor

The two-level rank idle predictor is used to predict long idle rank periods. The technique works well with applications that have long stretches of idle rank cycles, especially for DRAM system with multiple ranks. For DRAM system with multiple ranks, memory access can conflict in some ranks and leave other ranks idle.

The rank idle time predictor is a program counter (PC) based predictor inspired by the PC-based sampler dead block predictor (SDBP) [32]. The SDBP uses PC information to accurately predict whether an LLC block is "dead," i.e. whether it will be accessed again before being evicted. The design of the rank idle time predictor is based on the observation that if there is a long idle rank period after an instruction related to a LLC miss when there are no read requests in that rank, there is a high probability that the same behavior will be observed the next time this instruction causes a miss in the LLC with no read requests in that rank.

Figure 4.3 shows an example of memory access. Function foo calls function $foo1$ iteratively. In function $foo1$, we assume the physical address of $A[0] - A[100]$ map to

rank zero in DRAM and access to $A[i]$ always LLC misses. Similarly, assuming physical address of $B[0] - B[200]$ map to rank one in DRAM and access to $B[i]$ always LLC misses. This is a practical assumption, because the consecutive physical address have high probability of mapping to the same rank. After a miss in the LLC at the instruction that loads $A[99]$, the data flow will go into accessing data $B[i]$ which map to rank one. Therefore, there will be a long idle period in rank zero. After several iterations in foo, the predictor learns the access pattern related to the instruction that loads $A[99]$. The next time, when there are no read requests in rank zero and there is a miss in the LLC at the same instruction loading data $A[99]$, the predictor will predict that rank zero will be idle for a long period.

4.1.3.1 Making Prediction

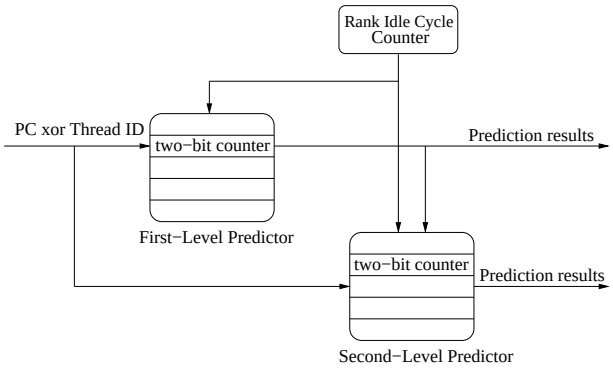


Figure 4.4: A two-level rank idle time predictor

Each rank has a two-level predictor. The structure is shown in Figure 4.4. Two levels are used so that if the first predictor mispredicts a long idle period, the second predictor has another chance to predict this long idle period. The two predictors have the same structure, make their predictions at different times, and update at the same time. The prediction

state consists of a table of two-bit saturating counters, much like a branch predictor. The predictor table is indexed by the address (PC) of the instruction and the thread number. The PC is that of the last instruction before the rank becomes idle. The predictor makes a prediction according to the high bit of the selected counter: long idle time if the bit is one, short idle time if the bit is zero. The rank idle cycle counter is used to count the number of idle cycles. This number is used to choose the predictor to make a prediction and update the predictor.

4.1.3.2 Prediction Driven Writeback Mechanism

Figure 4.5 shows the prediction driven writeback mechanism. As soon as a rank becomes idle, the first-level predictor makes a prediction about whether there will be read requests coming to that rank in the next m cycles. A sequence of s scheduled dirty cache blocks will be written back to DRAM during the predicted m idle cycles. In the DRAM system with eight-bank per rank, we choose $s = 8$ to maximize the bank-level parallelism when servicing the write requests. The parameter m is related to s; we want to make sure m can cover most of the service time of s scheduled dirty cache blocks.

Figure 4.6 shows the time to make a prediction during the idle rank cycles. Assuming the rank is idle from time t_1, the rank idle cycle counter starts to count the rank idle cycles and the first predictor makes a prediction at time t_1. If the prediction result from the first level predictor $P1$ is false (i.e., no long idle time predicted) and there are no read requests coming after n idle cycles, the second level predictor $P2$ is used to make a prediction. If the prediction result is true, s scheduled dirty cache blocks will be send to DRAM for service.

If both of the prediction results are false, but the idle rank time is longer than a threshold k, s scheduled dirty cache blocks are written back. This optimization comes from the observation that if there are rank idle cycles longer than k, there is a high probability that

```
function PredDrivenSched
begin
    if rank_idle_cycles == 1 then
        prediction = first_predictor_predict
    end
    else if rank_idle_cycles == n
        prediction=second_predictor_predict
    end
    else if rank_idle_cycles == k
        prediction = true
    end
    if prediction == true then
        call check_writeback
    end
end

function check_writeback
    if rank_idle_cycles == 0 then
        return
    end
    else if (prediction == true&&write_issued == s)
        call schedule_writeback
    end
    call add_event(check_writeback, write_issued == s)
end
```

Figure 4.5: Rank idle time prediction driven writeback scheduling algorithm

the idle cycles are also longer than $k + m$.

If either of the predictor results is true or the idle rank period is longer than the threshold k, the system will monitor the service of the write requests. If all of the previous s write requests have been finished service. and there are still no read requests coming in, another group of s scheduled dirty cache blocks will be sent to the DRAM system for service.

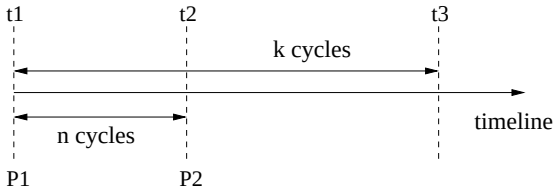


Figure 4.6: Prediction timeline

4.1.3.3 Predictor Update

The predictor will be updated when a read request comes and the rank is idle. If the idle rank cycles counted by the rank idle cycle counter are larger than m, the two-bit counter in the first-level predictor indexed by the the last PC and thread ID encountered before the rank was idle will be incremented; otherwise it will be decremented. If there are more than $m+n$ idle rank cycles, the corresponding two-bit counter in the second-level predictor will be incremented, otherwise it will be decremented.

Why does the rank idle time predictor work in multi-core systems The memory access patterns of most applications have spatial locality. Our technique is applied to the address mapping scheme that maps the rank and channel bits higher than the column bits, so the application tends to access a certain rank for a while before switching to another rank. In the modern DDRx memory systems, multiple controllers and multiple ranks are used to service the memory requests in parallel, thus in a lengthy stretch, only a small number of applications access a certain rank. Therefore, the memory access pattern for a certain rank is repeatable and predictable. Additionally, the rank idle predictor only makes the prediction when the rank starts to become idle, i.e., when all of the programs leave a rank idle. From our observation, the memory read accesses tend to come in bursts. The same program behavior that leads to one burst tends to lead to other bursts, as well as those bursts ending.

Channel	Rank	Bank	SSV				Next Ptr
0	0	0	0	1	1	1	Ptr
0	0	1	1	1	1	1	Ptr
0	0	2	1	1	1	0	Ptr
0	0	3	1	0	0	0	Ptr
0	0	4	1	1	1	1	Ptr
0	0	5	0	1	1	0	Ptr
0	0	6	1	1	1	0	Ptr
0	0	7	0	1	0	1	Ptr

Figure 4.7: SSV structure

4.1.4 LLC Writeback Policy

The LLC writeback policy searches for dirty cache blocks near the LRU position in the LLC and sends a sequence of scheduled dirty cache blocks to the write buffer. Scheduled writebacks are used because scheduled write requests map to memory resources in a way that can be serviced more efficiently.

In our implementation, a cache block is considered "near the LRU position" if it resides in the bottom eighth of the LRU recency stack [77]. We incorporate the rank idle time predictor into the LLC parallelism-aware writeback policy.

The LLC parallelism-aware writeback policy searches the dirty cache blocks in the LLC that target to the same rank but different banks. Compared with LLC writeback policy of VWQ, which exhaustively searches the row-hitting cache blocks in the related cache sets in the direction of Cache Cleaner [77], our scheme does not need to search a large amount of cache sets and perform tag matching, thus consuming less power and searching time.

The Cache Cleaner [77] uses a Search Set Vector (SSV) to help searching dirty cache blocks in the LLC that could be serviced more efficiently when mapping to the DRAM resources. Figure 4.7 shows a simple example of the SSV table with a 32-sets LLC and a single rank, eight-bank per rank DRAM system. Each bank has an entry in the SSV table

while each cache set that maps to this bank has a bit in the vector that is saved in this bank entry. When a dirty cache block is moved close to the LRU position, the bit in the SSV corresponding to this particular set will be set to one signifying that this set has a dirty cache block near the LRU position. Thus, when searching the dirty cache blocks in the LLC that target to different banks of the same bank, the cache set that has a bit set in its SSV entry will be issued, thus reducing the search time.

In the parallelism-aware scheduling scheme, when the predictor predicts a long idle period, a group of dirty cache blocks composed of the writeback requests to this idle rank but different banks are sent out to the DRAM system. In Figure 4.7, dirty cache blocks in the cache sets correspond to the bits in a vertical pattern are in the first group. If the rank is still idle after all the write requests in the group have been finished service, another group of dirty cache blocks in the horizontal pattern will be sent out to the DRAM system. Most modern DDRx systems use an eight-bank per rank memory configuration. Therefore, when more than one group of scheduled write requests are issued during the idle rank period, the rank resource access latency can be overlapped by bus burst cycles, reducing the average write request service time.

If the number of write requests in the write buffer is larger than a threshold, and there are no predicted long idle periods, the write requests will be sent to the DRAM for service whenever the rank is idle or the write buffer is full.

4.1.5 Storage Overhead

For each rank, we use a two-level rank idle predictor. Both levels are the same size. Each predictor has 8K entries and each entry has a two-bit counter. Thus, the total storage for the two-level rank idle predictor is 4K bytes. For an eight-core, 16M and 16-way LLC, the storage for SSV table is 2K bytes. Therefore, the total storage for the memory system with two memory controllers, two-rank per channel and four-rank per channel are 18K

bytes, 34K bytes, respectively. Both of them are less than 0.3% of the capacity of the 16M LLC.

4.2 Last-Write Prediction Driven Last-Level Cache Writeback

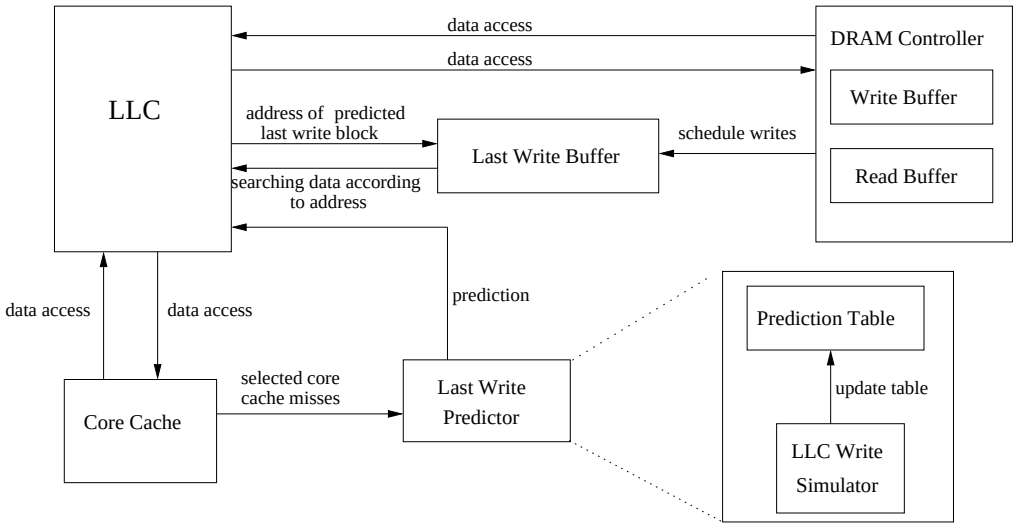


Figure 4.8: System structure

We propose a last-write prediction driven (LWPD) LLC writeback policy. Figure 4.8 shows the structure of our technique. A last-write predictor (LWP) is proposed to predict last-write blocks once they access the LLC. A last-write buffer is used to track predicted last-write blocks. Write requests in the last-write buffer as well as the write buffer are available to memory controller for scheduling. The LWPD writeback policy has the following advantages: 1) re-distributing the memory requests and balancing the memory bandwidth 2) expanding the scheduling space of memory controller, maintaining row-buffer hits and bank-level parallelism locality, and 3) completely decoupling from cache replacement policy allowing it to be applied to any LLC replacement policy.

4.2.1 Last-Write Predictor

The last-write predictor is used to predict last-write blocks in the LLC. It is composed of a lightweight LLC write simulator and a prediction table. Once a dirty block is evicted from the core cache and accesses the LLC, the last-write predictor consults the prediction table to make a prediction. The instruction PC related to the dirty block is hashed to index the prediction table to get the prediction result. A LLC write simulator is used to update the prediction table according to the simulated write behavior of the LLC.

The last-write predictor is a PC-based predictor. It is based on the observation that if an instruction PC leads to the last write access to one block, then there is a high probability that the next time this instruction is reached it will also lead to a last-write block. For a writeback cache, once a dirty block is evicted from the core cache, it has no PC information with it. Thus, a PC field is associated with each core block. Once a write accesses the core cache, the PC related to this write will be stored with the block.

4.2.1.1 Prediction Table

The prediction table uses skewed organization [32, 49] to reduce the impact of conflicts in the table. It consists of three tables, each indexed by a different hash of 16-bits partial PC. Each entry in the table has a two-bit saturating counter. Once a dirty block is evicted from the core cache and accesses the LLC, the LWP predicts whether or not this dirty block is a last-write block. The prediction decision is based on the sum of the counter values for all three tables that indexed by different hashes of the PC related to this dirty block: if the sum is greater than a threshold, then it is a last-write block. The prediction table is updated by the LLC write simulator.

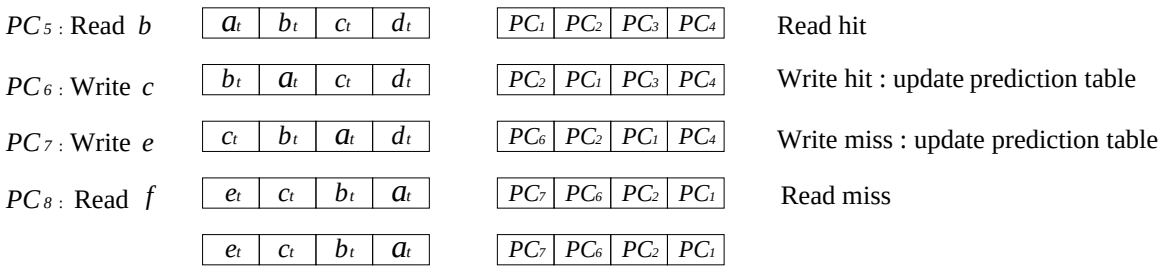


Figure 4.9: Behavior of the LLC write simulator

4.2.1.2 LLC Write Simulator

The LLC write simulator simulates the write behavior of the LLC and updates the prediction table. To reduce overhead, only a few sets of the LLC are represented. LLC sets are sampled; there is one simulated set for every 16 cache sets. Only partial tags are represented since simulator correctness is not required; in practice, we find 16 bits of tag leads to $>$99% accuracy with respect to full tags. Of course, no data are represented. The LLC write simulator only simulates the write behavior of the LLC, i.e. missing reads from memory are not placed in the simulator. The write accesses of the LLC account for about $1/3$ of total number of accesses on average in the memory intensive SPEC CPU 2006 benchmarks. Thus, the write simulator can use a smaller associativity compared with the LLC. The associativity of the LLC simulator is 6 while the associativity of the LLC is 16. Each entry in the simulator set has a partial tag field, a partial write PC field, a valid bit and an LRU recency field. When a write accesses a sampled LLC set, it also accesses the simulator simultaneously. The corresponding sampled set is searched for an entry with a matching tag; if there is a miss in the simulator, an entry is allocated using an LRU victim entry. LRU is used in the simulator, but since the associativity and number of sampled sets are low, the implementation of LRU is far more feasible than in the LLC [32]. The simulator also updates the prediction table. When a read accesses the simulator, if it is a

hit, the LRU recency will be updated. If it is a miss, the simulator will do nothing. Read access to the simulator updates the recency information for synchronizing the behavior of the simulator with the LLC, while the write access also needs to update the predictor.

Figure 4.9 illustrates the set behavior of the write simulator. Assuming a four-entry set, the box on the left side shows the LRU stack of the partial tag field. The box on the right side shows the partial write PC corresponding to the same entry with the partial tag on the left side. The PC for write access on the left in Figure 4.9 is the partial PC related to the evicted dirty block from the core cache.

At beginning, partial tags a_t, b_t, c_t, d_t of blocks a, b, c, d and their related PCs are reside in the set entries. First, request "read b" accesses the simulator, it is a read hit, so it updates the LRU recency of block b to the MRU position. Since it is a read access, the prediction table is not updated. Then, request "write c" accesses the simulator. It is a write hit meaning that PC_3 leads to a dirty block that could rewritten again before it is evicted. Thus, we update the entry in prediction table that indexed by PC_3 using 'not last-write', and update the LRU recency of block c to MRU position. Then request "write e" accesses the set. It is a miss, so we replace d with e since PC_4 leads to a last-write block d that did not access again before it is evicted. Thus, we update the entry in prediction table that indexed by PC_4 using 'last-write'. Finally, request "read f" accesses the set. It is a read miss, so the simulator does nothing.

The write simulator itself uses LRU replacement policy, but it can also accurately simulate the last-write behavior for LLC with other replacement policies. Write accesses to the write simulator and LLC are the same, thus they have same behavior. Though the replacement policy in LLC and write simulator may differ, a dirty block in the write simulator with LRU replacement policy that will not be accessed again before it is evicted also has a high probability that it will not be accessed again in LLC. Thus, the last-write predictor is independent of the LLC replacement policy.

4.2.2 Writeback Mechanism

4.2.2.1 Last-Write Buffer

In our technique, two buffers are used to hold write requests: the write buffer and the last-write buffer. The evicted dirty blocks are placed in the write buffer. The last-write buffer is used to track the predicted last-write blocks in the LLC. When the predictor predicts a last-write block, the physical address of the predicted last-write block will be placed into the last-write buffer. The write requests in the write buffer and the last-write buffer are available for scheduling. Since each entry in the last-write buffer only contains a 64-bit physical address, the data for the write requests are still in the LLC. Thus, memory read requests do not need to search the last-write buffer for address matching. This allows the last-write buffer to have many more entries than the write buffer. In our experiment, we use a 256-entry/channel (256-entry/c) per-rank last-write buffer, i.e. the last-write buffer is organized by rank and the total number of write buffer entries for a channel is 256 entries.

4.2.2.2 Priority Mechanism

An infinite write scheduling space would be able to always prioritize reads over writes, thus eliminating all write-induced interference. Given a finite scheduling space, it is better to prioritize writes over reads such that writes cause less interference to subsequent reads. In our technique, the service of write requests prioritizes read requests whenever either of the following conditions is satisfied: 1) The rank is idle and the write buffer has more active entries than a threshold m, or the last-write buffer has more active entries than a threshold n. 2) The write buffer or the last-write buffer is full. Condition 1 is to fill rank idle cycles with writes, reducing the contention between reads and writes. In condition 2, to ensure the progress of the application, scheduled writes in write buffer must be sent to DRAM for service when the write buffer is full to avoid pipeline stalls. Once the last-write buffer is full, the predicted last-write blocks must also be scheduled and sent to DRAM. Thus

entries in the last-write buffer can be used to hold the next predicted last-write requests.

Given the same group of scheduled write requests, writing them back through condition 1 imposes a less penalty to subsequent reads than in condition 2. Tracking last-write blocks using the last-write buffer allows more opportunities to redistribute the write requests into idle rank cycles. The threshold conditions for the write buffer and the last-write buffer ensure that a large number of scheduling candidates are available to the DRAM controller so they can be scheduled such that they can be efficiently serviced by the DRAM.

4.2.2.3 Scheduling Mechanism

When writes are prioritized over reads, the memory controller will schedule a sequence of a maximum number of s write requests to DRAM for service. The memory controller first schedules the row-buffer hit requests for the write with oldest time stamp. If all the row-buffer hit requests for this write have been scheduled, but the number of scheduled requests is still less than s, then the requests to the adjacent banks but same rank will be scheduled. The row-buffer hit and bank-level parallelism requests in the write buffer have high priority to be scheduled over the requests in the last-write buffer. Choosing the number of scheduled writebacks each time issued s is a trade off. If we issue fewer, we cause a high bus turnaround penalty and low row-buffer hit rate. If we issue more, the subsequent read requests can be delayed for a long time due to the service for writes. We choose s empirically.

Once the write request in the last-write buffer is ready to issue, it will first search the LLC for that dirty block according to the physical address in last-write buffer. If it is found in the LLC, the dirty block will be pulled from the LLC and send to DRAM for service. Then the corresponding dirty bit for that block will be cleaned. If the block is not found, then it has been evicted from the LLC, so this entry in the last-write buffer will be freed.

LLC misses tend to occur in bursts. Dirty blocks in or near the LRU position can be

evicted in a cluster. These writeback data compete for the memory bandwidth with the data being fetched into LLC, thus degrading system performance. In our technique, the predicted last-write blocks are exposed to DRAM controller once they access the LLC. Exposing last-write blocks to the memory controller at the early stage balances the memory bandwidth, allowing the service of write requests at a time that causes less interference with read requests.

Write requests in a small scheduling space tend to have low spatial and temporal locality. Servicing write requests with low locality imposes a large penalty on subsequent read requests. In our technique, the last-write buffer effectively expands the write scheduling space. The predicted last-write blocks increase the available scheduling candidates. Thus, our technique increases the possibility of scheduling row-buffer hit and bank-level parallelism write requests. Servicing a sequence of write requests with high locality not only improve write service efficiency for DRAM, but also reduces the write-imposed penalty to the subsequent reads.

4.2.3 Storage Overhead

In our technique, each core cache keep a 16 bits partial PC related to each block. For an eight-core 64 KB data cache, it consumes 16KB of storage. In the LLC write simulator, each entry keeps a 16 bits partial PC, 16 bits partial tag, 1 valid bit, 3 bits LRU position. The simulator has 1024 sets and 6 way associativity for a 16M capacity LLC, consuming 27.75KB. The three prediction tables for the skewed dead block predictor are each 4,096 two-bit counters, so they consume 3KB of storage. The dead write buffer has 512 entries, each entry has a 64 bits partial physical address stored in it, it consumes 4K Bytes. Thus, the total storage is 16KB+27.75KB+3KB+4KB=50.75KB, which is less than 0.5% of the 16M LLC capacity.

Execution core	4.8GHZ, eight-core CMP, out of order, 256 entry buffer, 48 entry load queue 44 entry store queue, 4 width issue/decode, 15 stages, 256 physical registers
Caches	L1 I-cache: 64KB/2 way, private, 64 bytes block size, LRU, 2-cycle L1 D-cache: 64KB/2 way, private, 64 bytes block size, LRU, 2-cycle L2 Cache: 16MB/16 way, shared, 64 bytes block size, LRU, 14-cycle
Main Memory	2 memory controllers, 8 banks per rank, 8K bytes row buffer per-bank DDR3-1600 11-11-11

Table 4.1: System configuration

4.3 Experimental Methodology

This section outlines the experimental methodology used in this study.

4.3.1 System

Name	**Symbol**	**Timings**	**Name**	**Symbol**	**Timings**
Precharge	^tRP	11	Burst Length	BL	4
Row access strobe	^tRAS	28	Row to column command delay	^tRCD	11
Read column address strobe	^tCL	11	Write column address strobe	^tCWL	8
Row activate to row activate delay	^tRRD	6	Row cycle	^tRC	39
Column address strobe to column address strobe	^tCCD	4	Read to precharge	^tRTP	6
Write recovery time	^tWR	12	Write to read delay time	^tWTR	6
Four activation window	^tFAW	24	Rank to rank switching time	^tRTRS	1

Table 4.2: DDR3-1600 DRAM timing

We use the MARSSx86 [57], a cycle-accurate simulator for X86-64 architecture. The experiment models an out-of-order eight-core processor with 16M shared LLC. The system configuration is shown in Table 4.1. The DRAMsim2 [69] is incorporated into MARSSx86 to simulate a detailed cycle-accurate DRAM system. We configure DRAMsim2 to model a DDR3-1600 DRAM system with two channels. Table 4.2 shows the detailed timing constraint for the DDR3-1600 DRAM modeled in our system.

4.3.2 Benchmarks

We use the SPEC CPU2006 [19] benchmarks for this study. Of the 29 SPEC CPU2006 benchmarks, 24 could be compiled and run without errors on MARSSx86. Table 4.3 shows

Name	Benchmarks
mix1	hmmer sphinx3 libquantum GemsFDTD gobmk perlbench lbm astar
mix2	perlbench gobmk namd lbm gamess GemsFDTD xalancbmk cactusADM
mix3	omnetpp hmmer cactusADM xalancbmk GemsFDTD gcc soplex astar
mix4	gromacs astar h264ref lbm omnetpp gcc libquantum calculix
mix5	gobmk tonto zeusmp milc bzip2 mcf hmmer astar
mix6	omnetpp libquantum hmmer sphinx3 bwaves milc xalancbmk calculix

Table 4.3: Multi-core workload mixes

six mixes of these 24 SPEC CPU2006 benchmarks randomly chosen eight at a time. We use these mixes for eight-core simulation. Each benchmark runs simultaneously with the others. For each mix, we made a checkpoint by running the one of the memory intensive benchmarks to a typical phase identified by SimPoint [74]. Then we run the experiment for 2 billion instructions total for all eight cores starting from the checkpoint. Each benchmark is run with the first *ref* input provided by the *runspec* command.

The memory scheduling technique we use for evaluation is First Ready-First Come First-Served (FR_FCFS) [68, 67]. The other memory read scheduling techniques could also work with our write scheduling optimization, we choose FR_FCFS for simplicity.

4.4 Experimental Results for Rank Idle Time Prediction Driven LLC Writeback Technique

In this section, we give the experimental results of rank idle time prediction driven LLC writeback studies.

4.4.1 Techniques

We evaluate six techniques for this study. Table 4.4 gives these techniques and a legend for their name. For traditional writeback, we simulated the following write buffer management policies: 1) writes in the write buffer are sent to the DRAM for service when the corresponding rank is idle or the write buffer is full, 2) writes in the write buffer are

Name	Technique
CI-CWB	Conventional writeback with cache line interleaving mapping scheme
PI-CWB	Conventional writeback with page interleaving mapping scheme
PA-WB	Parallelism-aware writeback
Eager-WB	Eager writeback
VWQ	Virtual Write Queue
RITPD-WB	Rank Idle Time Prediction Driven LLC Writeback in Section 4.1

Table 4.4: Legend for various writeback techniques.

sent to the DRAM only when the write buffer is full, 3) writes in the write buffer are sent to DRAM when the corresponding bank is idle or the write buffer is full. Our evaluation shows the policy 1) yields the best performance. To ensure fairness we choose to use the policy 1) for conventional writeback evaluation. Both two-rank per channel and four-rank per channel configurations are evaluated. The size of write buffer is 32-entry in our experiment.

4.4.2 Performance Analysis

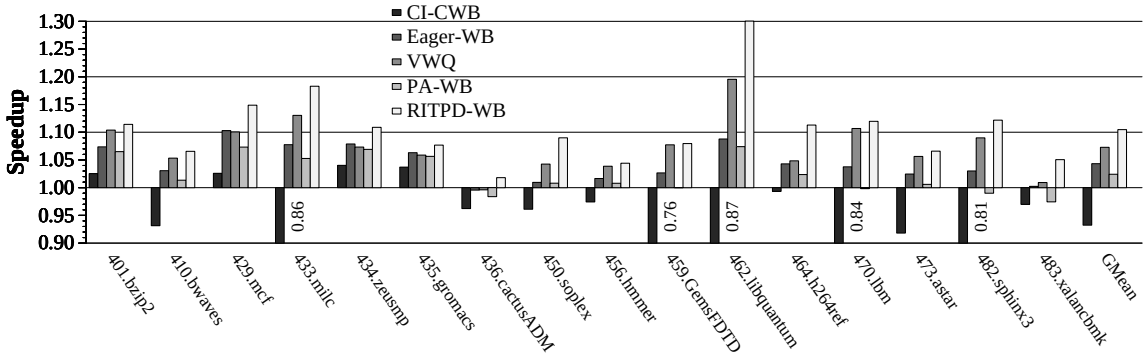


Figure 4.10: Performance evaluated on eight-core two-rank system

The baseline technique in our evaluation is PI-CWB. Figure 4.10 shows the IPC speedups

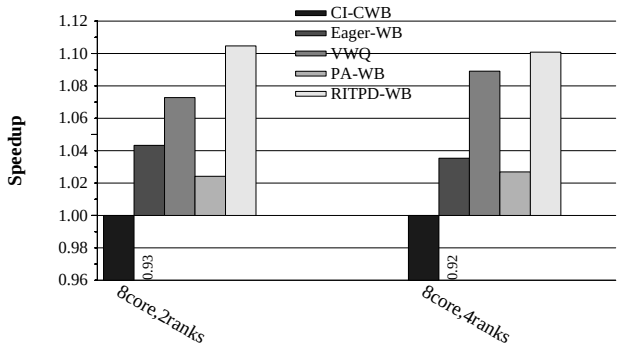


Figure 4.11: Average performance evaluated on two-rank and four-rank systems

normalized to baseline in a simulated eight-core processor with a two-rank DRAM system; that is, each channel has two ranks. For each benchmark, we show the speedup of the first run in the random combination. Benchmarks showing in Figure 4.10 are those the performance of perfect writeback could be improved more than 10% over the baseline. Perfect writeback means all write-induced interference is eliminated. If perfect writeback gives a significant improvement over the baseline for a particular benchmark, that means the performance of this benchmark has a potential to be improved when using writeback optimization. In this experiment, for 16 of 24 benchmarks, the performance of perfect writeback could be improved more than 10% over the baseline. Thus, most of the benchmarks can benefit from writeback optimization in a multi-core system.

In Figure 4.10, conventional writeback with page interleaving mapping scheme yields much better performance than conventional writeback with cache line interleaving mapping scheme. Therefore, we implement page interleaving mapping scheme in all the other techniques. From Figure 4.10, we can see RITPD-WB technique outperforms all the other techniques tested across all the benchmarks. Benchmark *libquantum* has a performance improvement as large as 30.0% when using the RITPD-WB technique due to its high memory access spatial locality. That is, the memory read requests access a particular rank

consecutively for a long stretch. So if write requests access the busy rank that services the read requests, there will be significant interference with the read requests. Therefore, *libquantum* benefits significantly by using the prediction driven technique to service memory write requests when the rank is idle.

In Figure 4.10, eager writeback improves performance by a geometric mean speedup of 4.3%. The performance improvement for the VWQ is 7.3%. Notice that the VWQ technique we implemented is in an optimal assumption that all the row-hitting write requests can be transferred back-to-back [77]; that is all the row-hitting dirty cache blocks in the near LRU position in LLC can be searched and provided during transferring the previous data from write buffer to DRAM. However, it is possible that the optimal assumption is not always satisfied in real systems, because searching a large number of cache blocks for tag matching is time consuming. The row-hitting ratio for write requests will be decreased when the optimal assumption is not satisfied, thus the system performance will be degraded compared with the optimal VWQ. Additionally, searching a large number of cache blocks for tag matching consumes significant LLC power. PAWB yields an average speedup of 2.4%. The RITPD-WB technique yields better performance over all other techniques. It improves performance by at least 10% of eight benchmarks and delivers an geometric mean speedup of 10.5%.

Figure 4.11 shows the average IPC improvement for two-rank and four-rank memory system configurations. For the four-rank configuration, eager writeback yields 3.5% speedup. The VWQ and PAWB techniques improve performance by 8.9% and 2.7% respectively, The RITPD-WB technique also delivers the best performance among all the tested techniques. It yields a 10.1% speedup.

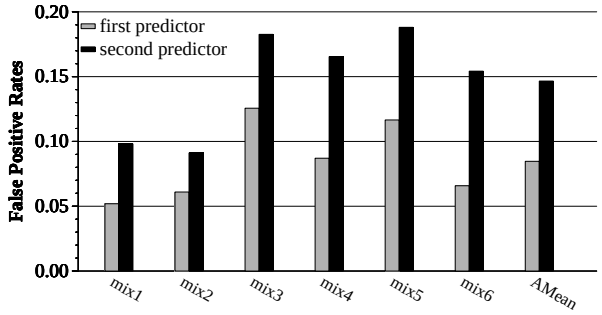


Figure 4.12: False positive rates for two-level predictor evaluated on eight-core two-rank system

4.4.3 Prediction Analysis

4.4.3.1 Predictor Accuracy

Mispredictions comes in two varieties: false positives and false negatives. False positives are more harmful because they wrongly allow the short rank idle periods to service the LLC writebacks. Those short idle periods can not cover the majority of the service time of writebacks, thus still causing significant write-induced interference. The false positive rate is calculated by the number of mispredicted positive predictions divided by the total number of predictions. Figure 4.12 shows the false positive rates of the two-level predictor for a two-rank system. False positive rates for the first-level and second-level predictors are 8.5% and 14.7% on average, respectively. These low false positive rates allow our predictor to effectively predict the large rank idle period while minimizing the damage caused by mispredictions.

4.4.3.2 Choosing Parameter

The threshold m is the minimum number of idle cycles the predictor predicts it will occur. We want m to cover most of the service time of the s (s=8 in our experiment) scheduled writebacks. In the DDR 1600 11-11-11 memory system, servicing a write re-

quest requires $\approx 29ns$, and the write-to-precharge latency is $\approx 14ns$. The write-to-read delay is $\approx 8ns$. So if the idle rank cycles $\geq 29+14+8 = 51ns$, most of the write-induced interference to the subsequent read will be eliminated. With a 4.8GHZ clock frequency, $51ns$ is 245 cycles, so we set $m = 300$ cycles for two-rank configuration. With the number of ranks in the same channel increasing, when a particular rank is idle, the data bus might be busy with transferring the data requested by other ranks. It might take a while for the bus to transfer the write request data for that idle rank. We set $m = 400$ cycles in the four-rank configuration.

The first predictor makes a prediction immediately after the rank becomes idle. The second predictor will make a prediction after the rank has been idle for n cycles. Parameter k is the threshold that if the number of idle cycles more than k, the scheduled writebacks will also be send to DRAM. In our experiment, we found $n = 200$ cycles and $k = 600$ cycles yield the best result.

4.4.3.3 Eliminated Write Interference

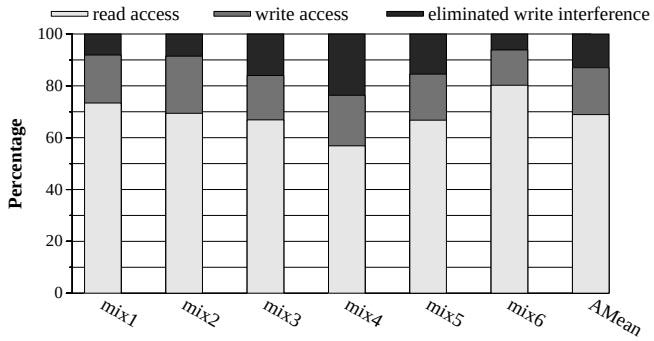


Figure 4.13: The percentage of write access, read access and completely eliminated write interference

Figure 4.13 shows the percentage of read accesses, write accesses and the completely

eliminated write interference using the rank idle time predictor. Eliminated write interference means write accesses that could be serviced during the predicted rank idle time. Write accesses account for 31.1% memory accesses on average. By using the prediction driven technique, 41.8% write accesses can be serviced during the predicted rank idle time. Our technique significantly reduces the write-induced interference.

4.4.4 Memory Efficiency Analysis

4.4.4.1 Read Latency

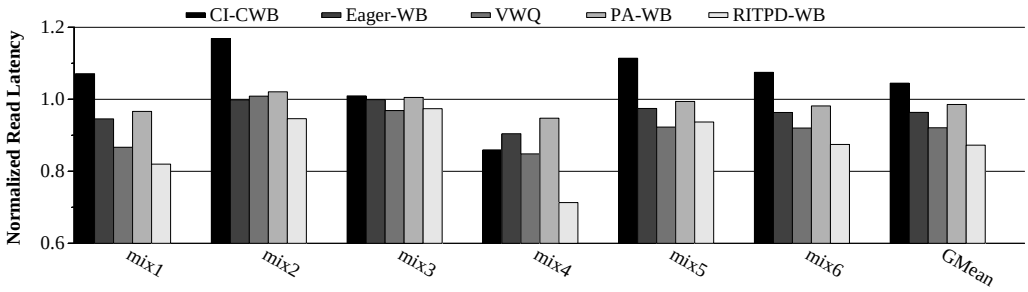


Figure 4.14: Read latency evaluation on eight-core two-rank system

Figure 4.14 shows the read latency normalized to eager writeback for the two-rank configuration. The RITPD-WB technique reduces the write-induced interference to read accesses, thus reducing the average read latency. From Figure 4.14, we can see the RITPD-WB technique reduces the read latency significantly across all the workloads. The VWQ technique even increases the read latency for mix2; in order to schedule more memory row-hitting write requests, the dirty cache blocks that reside in the bottom fourth [77] of the LRU recency stack are considered eligible for early writebacks in the VWQ technique. Although they use the cleaned bit technique to eliminate the extra writebacks, this technique can not eliminate the extra writebacks caused by early writing back the dirty cache

blocks for the first time. Compared with RITPD-WB technique, the VWQ technique has a larger rewrite ratio for mix2. These extra writebacks interfere with the read accesses, thus hurting the performance and increase the average read latency for mix2.

In our experiments, RITPD-WB reduces the read latency on average by 12.7% with two-rank configuration and 14.8% with four-rank configuration.

4.4.4.2 Bus Utilization

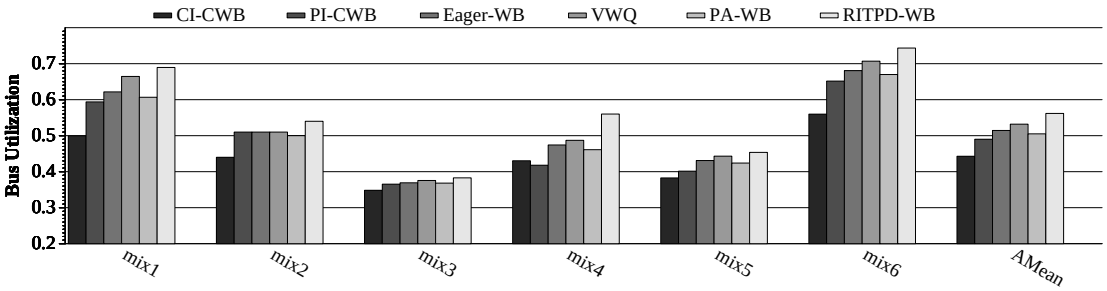


Figure 4.15: Bus Utilization evaluation on eight-core two-rank system

Bus utilization is calculated as the number of cycles the bus transfers data for read accesses divided by the total number of execution cycles. Memory write accesses are not taken into account for calculating bus utilization because the techniques we used for evaluation cause extra memory write accesses due to the early writebacks. If the write accesses are taken into account to calculate the bus utilization, the extra writebacks are contribute to the bus utilization, but the bus cycles used to transfer the extra writebacks are wasted. So to ensure fairness, only the read accesses are used to evaluate the bus utilization.

Figure 4.15 shows the bus utilization for the two-rank configuration. The RITPD-WB technique reduces the write-induced interference to reads, thus increasing bus utilization. The RITPD-WB technique delivers bus utilization superior to all the other techniques

Name	Technique
32-entry/c per-channel WB	Conventional writeback with 32-entry/c per-channel write buffer, this is the baseline
256-entry/c per-bank WB	Conventional writeback with 256-entry/c per-bank write buffer
512-entry/c per-bank WB	Conventional writeback with 512-entry/c per-bank write buffer
Eager Writeback	Eager writeback
VWQ	Virtual writeback queue
LWPG Writeback	Last write predictor guided writeback with 32-entry/c per-channel write buffer in Section 4.2

Table 4.5: Legend for various cache optimization techniques.

across all the workloads. It improves bus utilization on average by 14.5% and 15.3% with two-rank and four-rank configurations over PI-CWB technique.

4.5 Experimental Results for Last-Write Prediction Driven LLC Writeback Technique

In this section, we give the experimental results of last-write prediction driven LLC writeback studies.

4.5.1 Techniques

We use five distinct writeback optimizations for evaluation. In the graphs that follow, these techniques are referred to with abbreviated names. Table 4.5 gives a legend for these names.

A large per-channel and per-rank write buffer is complex and power inefficient. Given the same number of write buffer entries for a channel, a write buffer organized by bank consumes less on-chip power because memory read requests only need to search the write entries that target the same bank of the read request. Thus, we evaluate the per-bank write buffer structure with large number of entries, such as 512-entry/c, that is the total number of write buffer entries for a channel is 512 entries. A large number of write buffer entries is space inefficient, thus 512-entry/c per-bank write buffer is the largest write buffer we

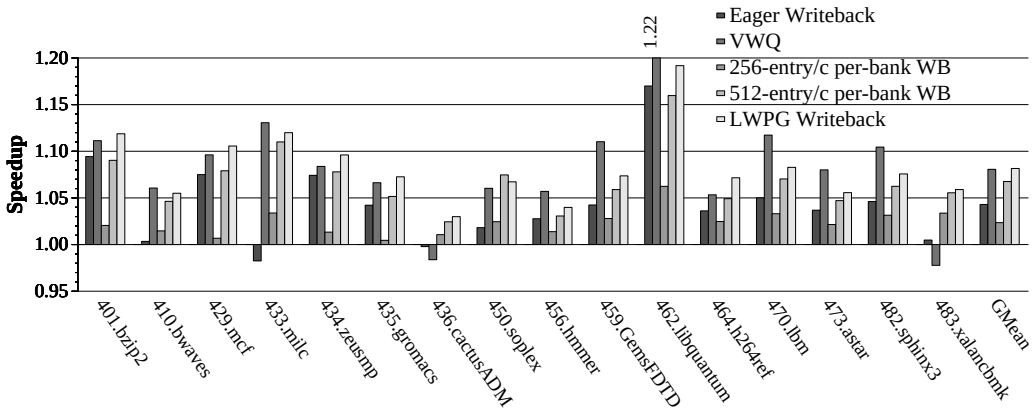


Figure 4.16: Results running on eight-core one-rank system with LRU LLC

evaluate. In the LWPG writeback technique, we use a 32-entry/c per channel write buffer and 256-entry/c per rank last-write buffer.

4.5.2 Performance Evaluation

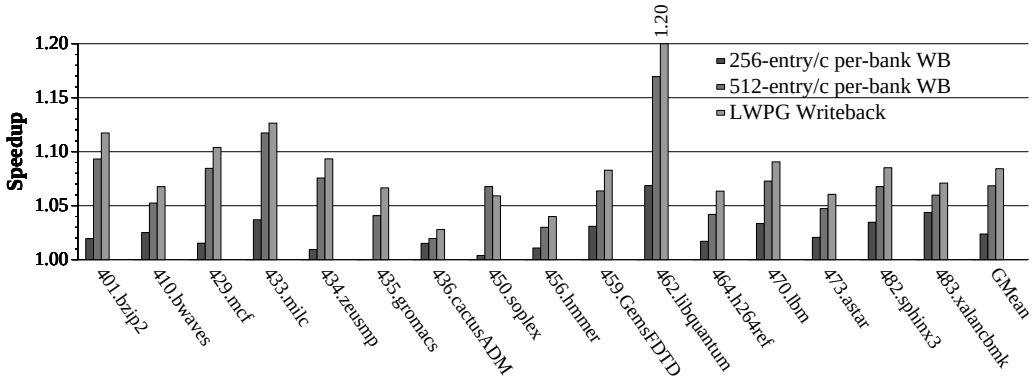


Figure 4.17: Results running on eight-core one-rank system with NRU LLC

We evaluate writeback optimizations with three LLC replacement policies: LRU, NRU and random.

Figure 4.16 shows the speedups of various writeback optimizations over the baseline

in a simulated eight-core processor with LRU LLC and a one-rank memory system; that is, each channel has one rank. For each benchmark we show the speedup of the first run in the random combination.

We choose benchmarks for which the performance of perfect writeback could be improved more than 10% over the baseline. Perfect writeback means all write-induced interference is eliminated. If perfect writeback gives a significant improvement over the baseline for a particular benchmark, that means the performance of this benchmark has a potential to be improved when using writeback optimization. In this experiment, for 16 of 24 benchmarks, the performance of perfect writeback could be improved more than 10% over the baseline. Thus, most of the benchmarks can benefit from writeback optimization in a multi-core system.

From Figure 4.16, we can see that LWPG writeback technique yields better performance than other techniques. The performance improvement over eager writeback is 4.3% on average over the baseline. The state-of-the-art VWQ technique achieves a 8.1% speedup on average. The LWPG writeback technique yields a average of 8.2% speedup. The traditional writeback with 256-entry/c and 512-entry/c per-bank write buffer yields 2.4% and 6.8% speedup respectively. Though the 512-entry/c per-bank write buffer has more buffer entries than the LWPG technique, its performance is not as good as the LWPG technique since the per-bank write buffer structure causes conflict misses for write requests that target to the same bank.

Figure 4.17 shows the IPC speedups with NRU LLC. The NRU recency stack has two levels. The recency information for NRU can not be used to accurately detect the last-write cache blocks. Thus, the eager writeback and VWQ techniques can not be applied to it. The traditional writeback with 256-entry/c and 512-entry/c per-bank write buffer achieve geometric mean of 2.3% and 6.7% speedups respectively. The LWPG writeback technique yields 8.4% geometric mean speedup.

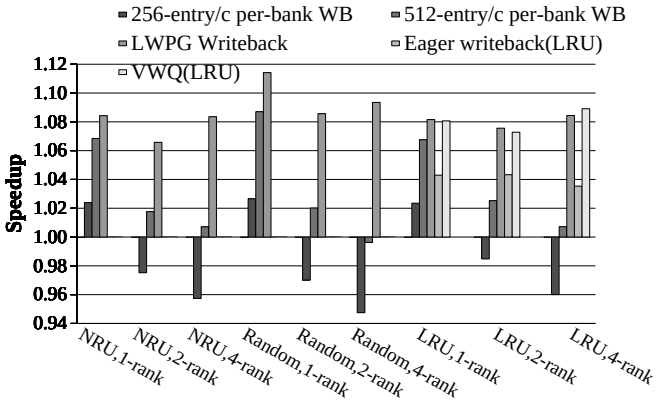
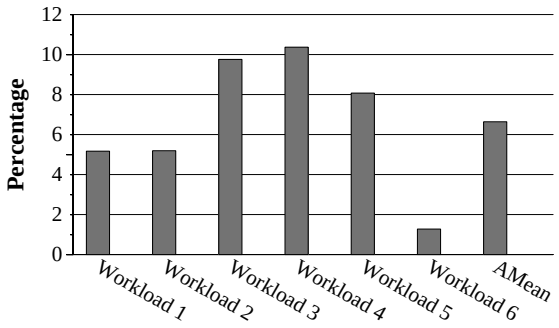


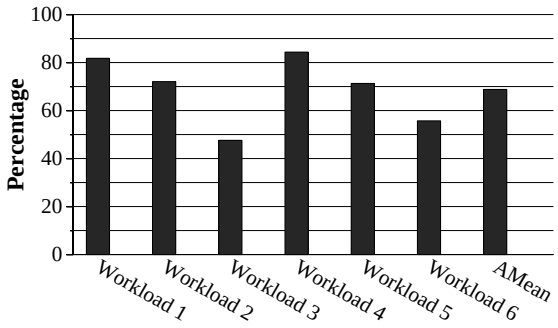
Figure 4.18: Performance evaluated for various configurations

Figure 4.18 shows the average IPC improvement for one-rank, two-rank and four-rank memory system configurations with LRU, NRU and random replacement policies. The LWPG writeback technique improves performance by 6.5%-11.4% with various DRAM configurations and LLC replacement policies. The system with random LLC replacement policy yields the best performance improvement since the random replacement policy randomly chooses a cache block to be evicted when a new block is placed. Thus, writes in a small write buffer have low temporal and spatial locality. The LWPG writeback technique that expands the scheduling space, providing more scheduling candidates. For the traditional writeback with 256-entry/c and 512-entry/c per-bank techniques, the speedups decrease as the number of ranks per-channel increases because increasing the number of ranks per channel decreases the number of write buffer entries for each bank, thus causing more conflict misses for write requests.

In our technique, once the rank is idle and the write buffer has more than m active entries, or the last-write buffer has more than n active entries for this idle rank, a sequence of scheduled write requests will be sent to DRAM for service. Choosing the parameters m and n is a trade-off between the ability to balance memory bandwidth and expanding the scheduling candidates. Choosing large values for m and n increases the possibility of



(a) False positive rate



(b) Fraction of correctly predicted last-write blocks

Figure 4.19: False positive rate and fraction of correctly predicted last-write blocks for last-write predictor with one-rank and NRU LLC configuration

high locality write requests, but decrease ability to balancing the memory bandwidth. In our experiment, $m = 12, 8, 4$ and $n = 96, 64, 32$ for 1/2/4 rank configurations respectively yields best performance. The maximum number of scheduled requests s each time issued by DRAM controller is also trade-off. A large value of s allows high row-buffer hit rate and low bus turnaround penalty, but can stall pipeline for a long time. In our experiment, we found $s = 12, 16, 16$ for 1/2/4 rank configurations respectively achieves best performance.

4.5.3 Prediction Evaluation

We evaluate the last-write predictor using false positive rate. The false positive rate is calculated by the number of mispredicted positive predictions divided by the total number of predictions. False positives allow the dirty cache blocks to be written again before they are evicted from the LLC to be written into the DRAM, thus causing extra memory writes. Figure 4.19 (a) shows the LWP yields a low false positive rate of 6.6% on average for NRU LLC with one-rank DRAM configuration.

We also evaluate the fraction of correctly predicted last-write blocks of LWP. The fraction of correctly predicted last-write blocks is calculated by the number of correctly predicted last-write blocks divided by the number of last-write blocks. A large fraction

means more opportunities for optimizations. Figure 4.19 (b) shows the fraction of correctly predicted last-write blocks is 68.8% on aveage for NRU LLC with one-rank DRAM configuration.

We also evaluate the LWP with all the 1/2/4 rank configurations and LRU, NRU and Random LLC. It yields false positive rate between 6.4%-7.1% and fraction of correctly predicted last-write blocks between 68.8%-76.0% on average with various configurations. This large fraction of correctly predicted last-write blocks and low false positive rates allows more opportunities for optimization without inducing significant extra writebacks.

4.5.4 Bus Utilization and Read Latency Evaluation

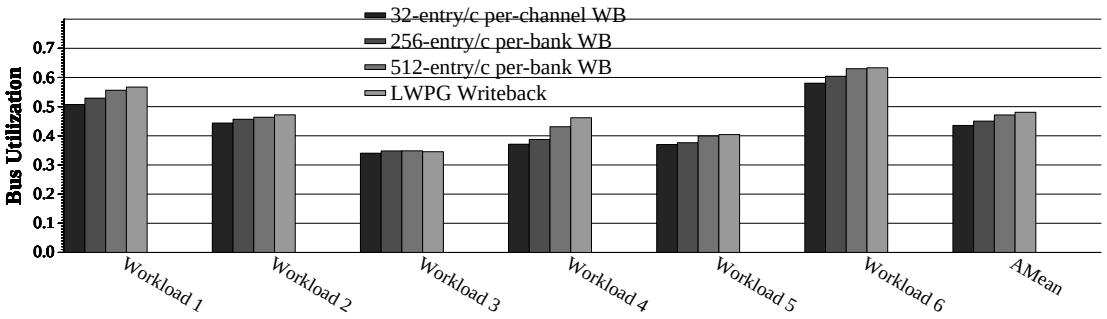


Figure 4.20: Bus utilization results running on eight-core one-rank system with NRU LLC

Bus utilization is calculated as the number of cycles the bus transfers data for read accesses divided by the total number of execution cycles. Memory write accesses are not taken into account for calculating bus utilization because the techniques we used for evaluation cause extra memory write accesses due to the early writebacks. If the write accesses are taken into account to calculate the bus utilization, the extra writebacks contribute to the bus utilization, but the bus cycles used to transfer the extra writebacks are wasted. So to ensure fairness, only the read accesses are used to evaluate the bus utilization.

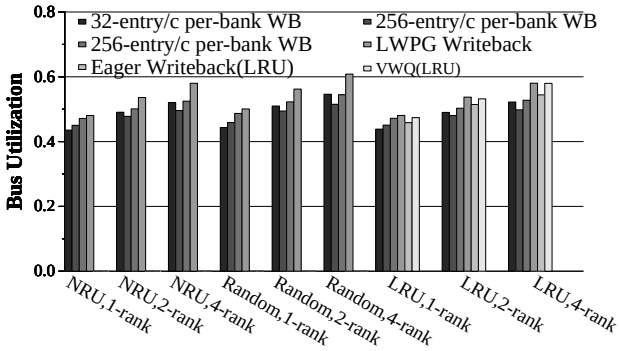


Figure 4.21: Performance evaluated for various configurations

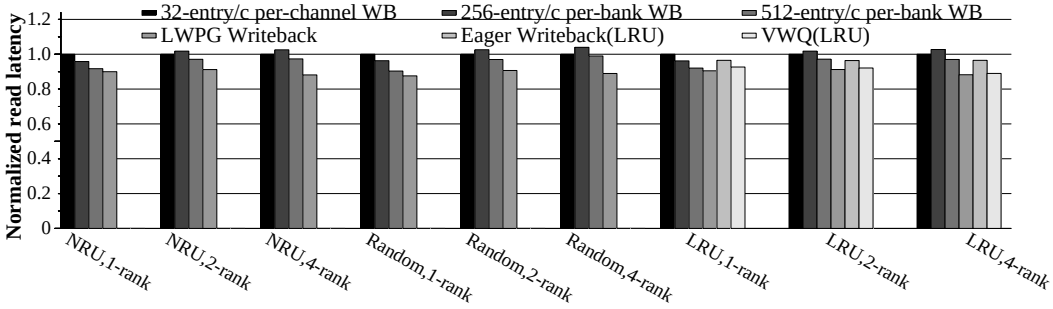


Figure 4.22: Read latency results for various configurations

Figure 4.20 shows the bus utilization results for system with NRU LLC. The LWPG writeback technique improves bus utilization across all workloads with an average of 11.6% compared with the baseline. Figure 4.21 shows the average bus utilization for multi-rank configurations for LRU, NRU and random replacement policies. Our technique consistently improves bus utilization by 9.2%- 13.6% for various DRAM configurations and LLC replacement policies.

Figure 4.22 shows the read latency for various configurations. The read latency is computed as the sum of the DRAM busy cycles for each core divided by the number of LLC misses. The LWPG writeback technique reduces the write-induced interference to read accesses, thus reducing the average read latency. The LWPG technique reduces the

read latency by 8.8%-12.4% on average across various configurations.

4.5.5 Row-buffer Hits Rate Evaluation for DRAM Writes

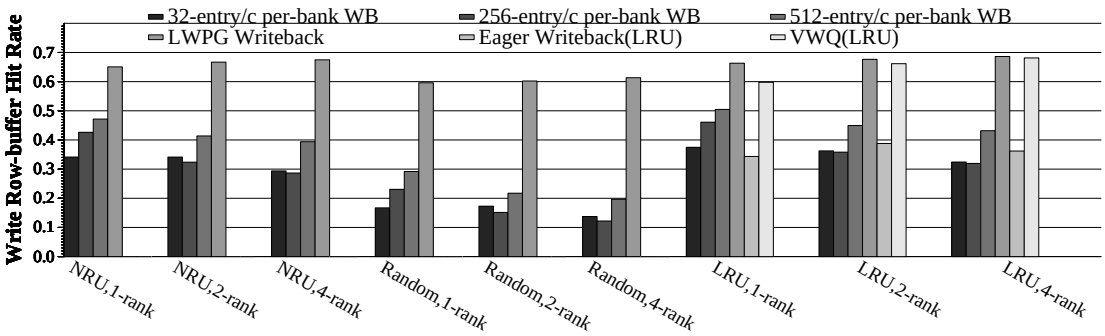


Figure 4.23: Writes row-buffer hit rate for various configurations

Figure 4.23 shows results for average write row-buffer hit rates with various configurations. Since caches filter the spatial locality of writes, the traditional writeback with a small write buffer yields low row-buffer hit rate. The traditional writeback with a randomly-replaced cache only yields 13.7%-17.3% row-buffer hit rate on average because the random replacement policy randomly chooses a cache block to be evicted once a new cache block comes in. Our technique significantly improves row-buffer hit rate for writes across various configurations to 59.6%-68.6% on average.

5. EXPLORING PERFORMANCE VARIANCE TO REDUCE WRITE OVERHEAD OF NON-VOLATILE MEMORY*

Technology scaling of SRAM and DRAM is increasingly constrained by fundamental technology limits. Emerging memory technologies, such as Spin Torque Transfer RAM (STT-RAM), Phase-Change RAM (PCM), and Resistive RAM (RRAM) are being explored as potential alternatives to existing memories in future computing systems. Compared to the traditional SRAM/DRAM technology, these emerging memories have the common advantages of high density, low standby power, better scalability, and non-volatility, and hence become very attractive as the alternatives for future memory hierarchy [84]*.

In order to use such emerging memories, several design issues must be solved. The most important is the performance and energy costs of writes. Since NVM has an inherently stable mechanism for data storage, it takes more time and energy to overwrite data [87]*.

From the previous chapter, we know that write-induced interference can significantly reduce performance. Large write overhead is a more severe problem in NVM-based memory. The long write latency can degrade performance by causing large write-induced interference to subsequent read requests. The high write energy can increase the power consumption.

By exploring the asymmetric read/write feature of an STT-RAM based LLC, we pro-

*©2014 IEEE. Reprinted, with permission, from Wang, Zhe; Jiménez, Daniel A.; Xu, Cong; Sun, Guangyu; Xie, Yuan, "Adaptive placement and migration policy for an STT-RAM-based hybrid cache," High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium,Feb. 2014

*©2013 Association for Computing Machinery, Inc. Reprinted by permission, from Zhe Wang, Shuchang Shan, Ting Cao, Junli Gu, Yi Xu, Shuai Mu, Yuan Xie, and Daniel A. Jiménez. WADE: Writeback-aware dynamic cache management for NVM-based main memory system. ACM Trans. Archit. Code Optim. 10, 4, Article 51 (December 2013), 21 pages. DOI=10.1145/2555289.2555307 http://doi.acm.org/10.1145/2555289.2555307

Memory Type	1M SRAM	2M SRAM	2M STT-RAM	4M STT-RAM
Area (mm^2)	0.825	1.650	0.518	1.035
Read Latency (ns)	1.751	2.017	2.681	2.759
Write Latency (ns)	1.530	1.663	10.954	10.993
Read Energy (nJ/access)	0.055	0.072	0.132	0.142
Write Energy (nJ/access)	0.039	0.056	0.608	0.618
leakage power (mW)	29.798	59.596	7.108	14.216

Table 5.1: Characteristics of SRAM and STT-RAM caches (22nm, temperature=350K)

pose an adaptive placement and migration policy for an STT-RAM-based hybrid cache. The technique places a block into either STT-RAM lines or SRAM lines by adapting to the access pattern of write requests. It can achieve high performance by making use of the large capacity of STT-RAM and maintain low write overhead using SRAM.

By exploring read/write disparity of PCM-based main memory, we propose writeback-aware dynamic cache management for NVM-based main memory system. The technique improves performance and energy efficiency by reducing the number of writeback requests to NVM-based main memory.

5.1 APM: Adaptive Placement and Migration Policy for an STT-RAM-Based Hybrid Cache

5.1.1 Comparison of STT-RAM and SRAM Cache

Compared to SRAM, STT-RAM caches have higher density and lower leakage power, but higher write latency and write power consumption. Table 5.1 lists the technology features of various STT-RAM cache bank sizes and SRAM cache bank sizes used in our evaluation. The technology parameters are generated by NVSim [12], a performance, energy, and area model based on CACTI [51]. The cell parameters we used in NVSim are based on the projection from Wang *et al.* [82]. We assume a 22nm $\times$ 45nm MTJ built with 22nm CMOS technology. The SRAM cell parameters are estimated using CACTI [51].

The density of STT-RAM is currently 3×-4× higher than SRAM. Another benefit for STT-RAM is its low leakage power. Leakage power can dominate the total power consumption for large SRAM-based LLCs [36]. Thus, the low leakage power consumption of STT-RAM makes it suitable for a large LLC. Disadvantages of STT-RAM are long write latency and high write energy.

5.1.1.1 Hybrid Cache Structure

The hybrid cache structure is composed of STT-RAM banks and SRAM banks. Each cache set consists of a large portion of STT-RAM cache lines and a small portion of SRAM cache lines distributed among multiple banks. The hybrid cache architecture relies on an intelligent block placement policy to bridge the performance and power gaps between STT-RAM and SRAM.

An intelligent block placement policy for hybrid cache design should be optimized for three requirements. First, the SRAM portion should service as many write requests as possible, thus minimizing the write overhead of STT-RAM portion. However, sending many write operations to SRAM without considering the access pattern can cause misses due to the small capacity of SRAM, leading to performance degradation. Thus, the second requirement is that reused cache blocks should be placed in the LLC to maintain performance by hiding the memory access latency. Finally, the block placement policy should be a low overhead and low complexity design without incurring frequent migration between cache lines.

5.1.2 Analysis of LLC Write Access Patterns

LLC block placement is often initiated by a LLC write access that can be categorized into three classes: *prefetch-write, core-write and demand-write*. Figure 5.1 shows the breakdown of each class of LLC write accesses for 17 memory intensive SPEC CPU2006 benchmarks. The study is performed using the MARSSx86 [57] simulator with single-

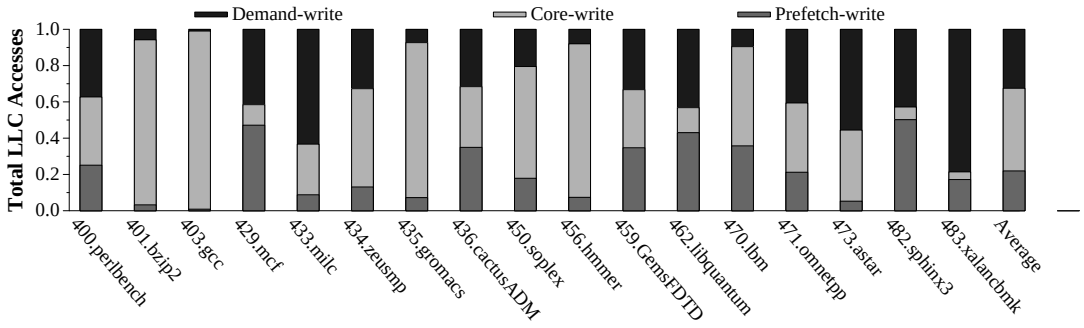


Figure 5.1: Distribution of LLC write accesses. Each type of write access accounts for a significant fraction of total write accesses

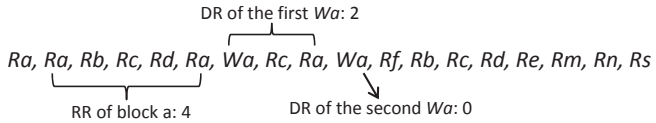


Figure 5.2: An example illustrating read range and depth range

core configuration and a 4MB LLC. We implement a middle-of-the-road stream prefetcher that models the prefetcher of the Intel Core i7. From Figure 5.1, we can see that each type of write access accounts for a significant fraction of total write accesses. Prefetch-writes account for 21.9% on average while core-writes and demand-writes take 45.6% and 32.5%, respectively. In this section, we analyze the access pattern of each write access type and suggest a block placement policy that adapts to the access pattern for each class.

We first define the terminology that will be used later in this section for pattern analysis. To be clear, when we write "block" we mean a block of data apart from its physical realization. When we write "line" we mean the physical frame, whether in STT-RAM or SRAM, where a block may reside.

- **Read-range**: The read-range is a property of a cache block that fills the LLC by a demand-write or prefetch-write request. It is the largest interval between consecutive

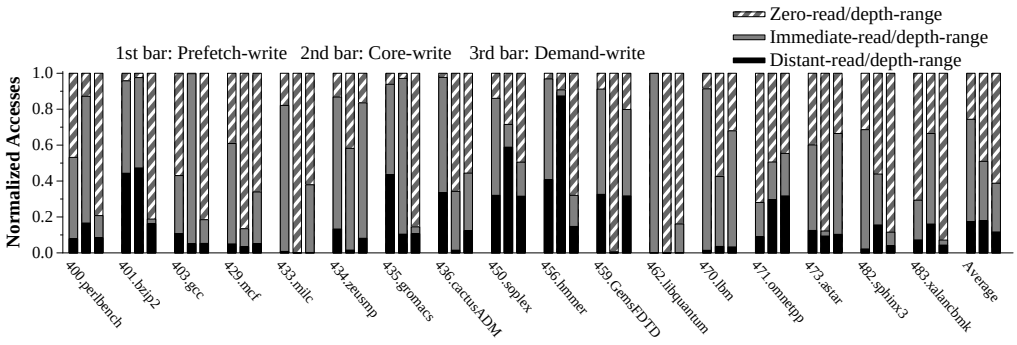


Figure 5.3: The distribution of access pattern for each type of LLC write access

reads of the block from the time it is filled into the LLC until the time it is evicted.

- **Depth-range**: The depth-range is a property of a core-write access. It is the largest interval between accesses to the block from the current core-write access until the next core-write access to the same block. The "depth" refers to how deep the block descends into the LRU recency stack before it is accessed again.

We use an example to illustrate the read range and depth range. Figure 5.2 shows the behavior of a block a from the time it fills into a 8-way set until it is evicted. In the example, Ra represents "read block a" while Wa represents "write block a". The largest re-read interval of block a during the time it resides in the cache is 4 which is the read interval between the second Ra and the third Ra. Thus, the read range (RR) of block a is 4. The depth range (DR) of the first Wa access is 2 which is the access interval between the first Wa access and the fourth Ra access. The depth range of the second Wa access is 0 meaning a is not re-written from the second Wa until it is evicted from the LLC.

We further classify the read/depth-range into three types: *zero-read/depth-range, immediate-read/depth-range* and *distant-read/depth-range*.

- Zero-read/depth-range: Data is filled into the LLC by a prefetch or demand request/core-write request, and it is never read/written again before it is evicted.

- Immediate-read/depth-range: The read/depth-range is smaller than a parameter m. We set $m = 2$ which is the same as the number of SRAM ways in our hybrid cache configuration as in Section 5.

- Distant-read/depth-range: The read/depth-range is larger than $m = 2$ and at most the associativity of the cache set which is 16 in our configuration.

5.1.2.1 Pattern Analysis of Prefetch-Write Block

Prefetching [76, 88] data into the cache before it is accessed can improve performance by hiding memory access latency. However, prefetching can also induce cache pollution by inaccurate prefetch requests.

We analyze the access pattern for the LLC prefetch-write blocks by using read-range. The first bar in Figure 5.3 shows each type of access pattern as a fraction of the total number of prefetch-write blocks. Zero-read-range prefetch-write blocks are inaccurately prefetched blocks accounting for 26% of all of prefetch blocks. Placing the zero-read-range prefetch-write blocks into the STT-RAM lines causes pollution and high write overhead. Thus, zero-read-range prefetch-write blocks should be placed in SRAM lines.

The immediate-read-range access pattern is related to cache bursts. After an initial burst of references, a cache block becomes dead, i.e. it is never used again prior to eviction. Of all prefetch-write blocks, 56.9% are immediate-read-range blocks. Immediate-read-range prefetch-write blocks should be placed in SRAM lines so they can be accessed by subsequent demand requests without incurring write operations to STT-RAM lines. Moreover, placing immediate-read-range prefetch-write blocks in SRAM allows them to be evicted when they are dead, reducing pollution.

Distant-read-range prefetch-write blocks should be placed in STT-RAM lines to make use of the large capacity to avoid cache misses. Distant-read-range prefetch-write blocks account for only 17.5% of all prefetch blocks.

Zero-read-range and immediate-read-range prefetch-write blocks account for 82.5% of all prefetch-write blocks. Thus, we suggest the initial placement of the prefetch-write blocks in SRAM. Once a block is evicted from the SRAM, if it is a distant-read-range block, i.e. it is still live, it should be migrated to STT-RAM lines. Otherwise, the block is dead and should be evicted from the LLC. Since only 17.5% of prefetch blocks are distant-access prefetch blocks, the migration will not cause significantly increased traffic.

5.1.2.2 Pattern Analysis of Core-Write Access

In our design, if a core-write access misses in the LLC, the data will be written back to the main memory directly. Thus, our core-write placement policy is only designed for core-write hit accesses.

We analyze the access pattern of the LLC core-write access by using depth-range. The second bar in Figure 5.3 shows the access pattern for core-write accesses. Zero-depth-range accesses account for 49.1% of all core-write accesses. Though the data written by the zero-depth-range core-write access will not be rewritten before it is evicted, it still has some chance to be read again. Thus, we suggest leaving zero-depth-range data in its original cache line for avoiding read misses and block migrations.

Immediate-depth-range accesses account for 32.9% of total core-write accesses. The immediate-depth-range accesses are the write-intensive accesses with write burst behavior. Thus, the immediate-depth-range access data is preferred to be placed in the SRAM line for low write overhead. The distant-depth-range access data should remain in its original cache line, thus minimizing migration overhead.

5.1.2.3 Pattern Analysis of Demand-Write Block

The access pattern of demand-write blocks is analyzed using read-range. Zero-read-range demand-write blocks, also known in the literature as “dead-on-arrival” blocks, are brought to the LLC by a demand request and never referenced again before being evicted.

It is unnecessary to place zero-read-range demand-write block into the LLC so the block should bypass the cache (assuming a non-inclusive cache). The third bar in Figure 5.3 shows dead-on-arrival blocks account for 61.2% of LLC demand-write blocks. Thus, bypassing dead-on-arrival blocks can significantly reduce write operations to LLC. Moreover, bypassing can improve cache efficiency by allowing the LLC to save space for other useful blocks in the cache.

The immediate-read-range and distant-read-range demand-write blocks account for 38.8% of the total demand-write blocks. We suggest placing them in the STT-RAM ways for making use of the large capacity of the STT-RAM portion and reducing pressure on the SRAM portion.

5.1.2.4 Pattern Analysis Conclusions

Each class of LLC write access can be applied to a different placement policy. The access pattern of each class type can be used to guide the block placement policy. From the analysis of the access pattern of each access class, we make the following conclusions: (1) The initial placement of prefetch-write blocks should be to SRAM lines. (2) Write-burst core-write data should be placed in SRAM lines while other types of core-write data should remain in their original cache lines. (3) Dead-on-arrival demand-write blocks should bypass the LLC while the other types of demand-write blocks should be placed in the STT-RAM lines. (4) When a block is evicted from SRAM, if it is live, it should be migrated to STT-RAM lines for avoiding LLC misses.

5.1.3 Policy Design

The design of the block placement and migration policy is guided by the access pattern of each write access type. An access pattern predictor is proposed to predict write-burst blocks and dead blocks. The information provided by the access pattern predictor is used to direct bypass and migration of blocks between STT-RAM lines and SRAM lines. The

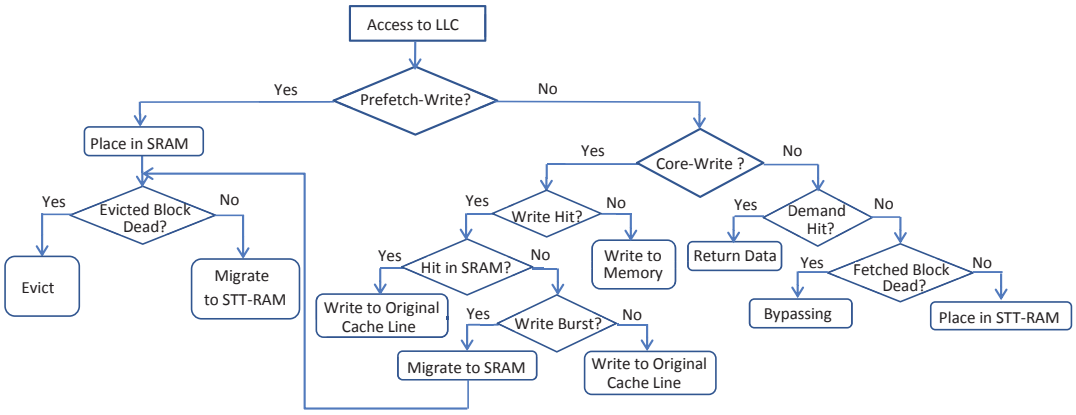


Figure 5.4: Flow-chart of the adaptive block placement and migration mechanism

policy targets reducing write overhead by allowing the SRAM portion to service as many write requests as possible and attain high performance by benefiting from the high density of the STT-RAM portion.

Figure 5.4 shows a flow-chart for the technique. In our design, a prediction bit is associated with each block in the LLC for representing whether the block is predicted dead. An access to the LLC searches all the STT-RAM lines and SRAM lines in the set. On a prefetch miss, the prefetched data is placed into an SRAM line and the prediction bit is set to 1 meaning we assume the prefetch block is dead on arrival. For every demand hit request in the SRAM lines, the access pattern predictor makes a prediction about whether the block is dead. Once the block is evicted from the SRAM lines, if it is predicted dead, it will be evicted from the LLC. Otherwise, it will be migrated to the STT-RAM lines. In this case, if a prefetch block is never accessed before it is evicted from the SRAM lines, it is taken as an inaccurately prefetched block and evicted based on the observation that accurately prefetched blocks are usually accessed soon by subsequent demand requests.

On a core-write request to the LLC, if it is an LLC miss, it will be written to main memory directly. For the core-write hit request in the STT-RAM lines, if it is predicted

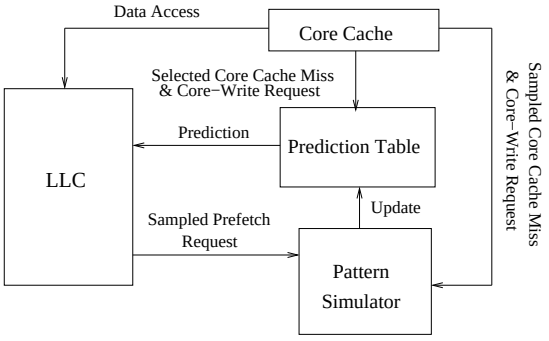


Figure 5.5: System structure

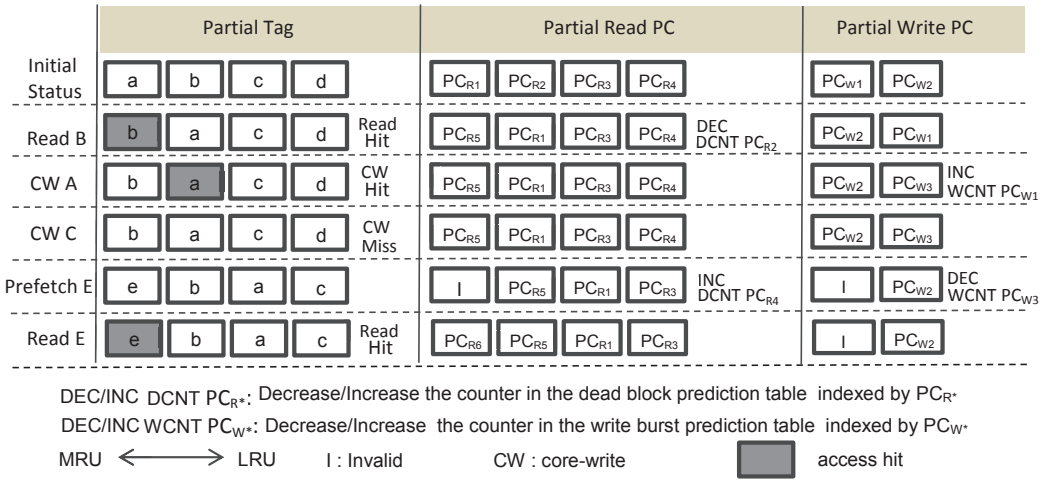


Figure 5.6: An example illustrating the set behavior of pattern simulator

to be a write burst access, then data will be written to the SRAM lines with the prediction bit set to 0 indicating the block is predicted live; the prior position in the STT-RAM line is set to invalid. Then, as with prefetched blocks, once the data is evicted from the SRAM portion, a predicted dead block is evicted from the LLC while a live block is migrated to the STT-RAM portion.

On a demand miss to the LLC, the data is fetched from the main memory. If the fetched block is predicted to be a dead-on-arrival block, it will bypass the LLC. Otherwise, the

block will be placed in the STT-RAM lines.

Minimizing Write Overhead The proposed scheme reduces write operations to STT-RAM portion in the following ways. First, bypass can reduce write operations to STT-RAM lines caused by dead-on-arrival requests. Second, SRAM lines filter the write operations caused by the inaccurate and immediate-read-range prefetch requests. Finally, the core-write-intensive blocks are placed in the SRAM lines, reducing the write operations to STT-RAM lines caused by write burst behavior.

Attaining High Performance The block placement policy can attain high performance for the hybrid cache by benefiting from the high density of the STT-RAM portion. Specifically, the distant-read-range blocks are placed in STT-RAM lines that can reduce cache misses by making use of the large capacity of STT-RAM. Also, bypassing zero-range demand blocks saves space in the LLC for other useful data. Moreover, filtering zero-range prefetch blocks and immediate-read-range blocks using SRAM lines can improve cache efficiency by reducing inaccurately prefetched blocks and dead blocks in the LLC.

The technique relies on an access pattern predictor for directing block bypassing and migration.

5.1.3.1 Access Pattern Predictor

The goal of the access pattern predictor is to predict dead blocks and write burst blocks for guiding block placement. The predicted dead blocks are used to direct bypassing and block migration from SRAM lines to STT-RAM lines. Write burst blocks are used to guide block migration from STT-RAM lines to SRAM lines for core-write accesses. The access pattern predictor consists of a prediction table and a pattern simulator as shown in Figure 5.5. The prediction table is composed of a dead block prediction table and a write burst prediction table having the same structure but making predictions for different types of accesses. The pattern simulator is used to learn the access pattern for updating the

prediction table.

The design of the access pattern predictor is inspired by the sampling dead block predictor (SDBP) [34]. However, the SDBP predicts dead blocks only taking into account demand accesses, while the access pattern predictor predicts both dead blocks and write-intensive blocks by considering all types of LLC accesses. The predictor predicts access pattern using the Program Counter (PC) of memory access instructions. The intuition is the cache access pattern can be classified based on the instructions of the memory accesses. Specifically, if a given memory access instruction PC leads to some access pattern in previous accesses, then the future access pattern of same PC will be similar.

Making Prediction The access pattern predictor makes a prediction in the following three conditions: (1) When a core-write request hits in the STT-RAM lines, the write burst prediction table will be accessed to predict whether it is a write burst request. (2) For each read hit request in the SRAM lines, the dead block prediction table will be accessed to predict whether it is a dead block. (3) On a demand-write request, dead block prediction table will be accessed to predict whether it is a dead-on-arrival request.

The dead block prediction and write burst prediction tables have the same structure. Each entry in a prediction table has a two-bit saturating counter. When making a prediction, bits from the related memory access instruction PC are hashed to index a prediction table entry. The prediction result is generated by thresholding the counter value in the prediction table entry. In our implementation, we use a skewed organization [85, 34, 49] to reduce the impact of conflicts in the hash table.

Updating Predictor The pattern simulator samples LLC sets and simulates the access behavior of the LLC by using sampled sets. It updates the prediction table by learning the access pattern of the PCs from the simulated LLC access behavior. It targets learning the dead/live behavior and write burst behavior of LLC blocks.

Each simulated set in the pattern simulator has a tag field, a read PC field, an LRU field, a write PC field, a write LRU field and a valid bit field. The partial tag and partial read/write PC which are the lower 16-bit of the full tag and full read/write PC are stored in the tag field and read/write PC field. The read PC field is for learning the dead/live behavior of the block while the write PC field is for learning the write-burst behavior of the block.

A write burst occurs within a small number of cache lines. Thus the write PC field should have a small associativity. The pattern simulator consists of two parts: the tag array and its related read PC field, LRU field and valid bit field which have the same associativity while the write PC field and its write LRU field have a smaller associativity. In our implementation, we found 4-way associativity of the write PC field and 12-way associativity of the read PC field yield the best performance while the associativity of LLC is 18.

The behavior of a pattern simulator set is illustrated in Figure 5.6 using an example access pattern. In the example, the associativity of the tag and read PC fields is 4 while the associativity of the write PC field is 2. On each demand hit request, the pattern simulator updates the dead block prediction table entry indexed with the related read PC by decreasing the counter value indicating "not dead." When a block is evicted from the simulator set, the pattern simulator updates the dead block prediction table entry indexed with the related read PC by increasing the counter value indicating "dead." The LRU recency is updated for every demand request and prefetch-write request. The write PC field is for learning the write-burst behavior for core-write requests. On each core-write hit request, the simulator increments the counter value stored in the write burst prediction table entry indexed with the related write PC indicating "write burst." When a block is evicted from the write field, the simulator decrements the counter value stored in the write burst prediction table entry indexed with the related write PC indicating "not write burst."

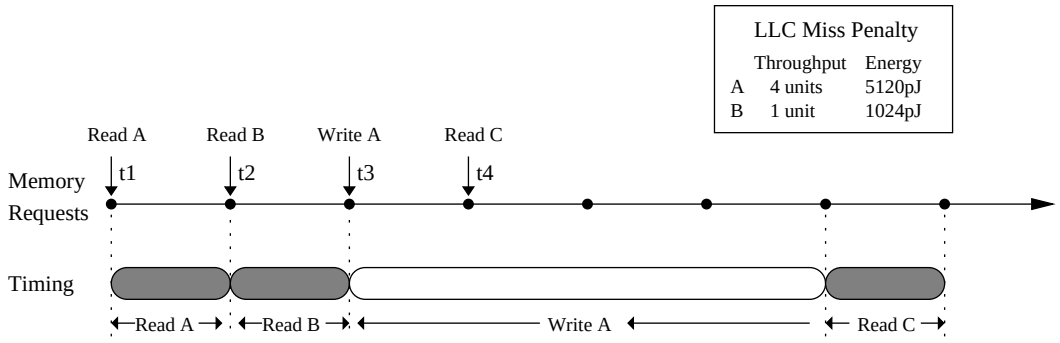


Figure 5.7: LLC miss penalty on throughput and energy for dirty cache block and clean cache block

5.2 WADE: Writeback-Aware Dynamic Cache Management for NVM-based Main Memory System

5.2.1 Motivation

Dirty and clean cache blocks in the LLC have different properties. When dirty cache blocks are evicted from the LLC, they will be written into main memory incurring performance and energy overhead, while clean cache blocks will not affect the system when they are evicted.

Figure 5.7 shows an example demonstrating the disparity in LLC miss penalties for dirty data and clean data on PCM throughput and energy. Assuming a request 'read A' missed in the LLC and is sent to PCM for service, servicing request 'read A' takes one time unit and A is brought into the LLC. Then a request 'read B' missed in the LLC and is serviced by PCM for one time unit. In the LLC, 'block A' is accessed by a write hit and the dirty bit is set. After 'dirty block A' is evicted from the LLC, it will be written back to PCM. Assuming servicing write request 'write A' takes 4 time units. At time $t4$, a request 'read C' is sent to PCM that targets to the same device with 'request A'. Then C has to wait until the completion of servicing 'A'. In this case,

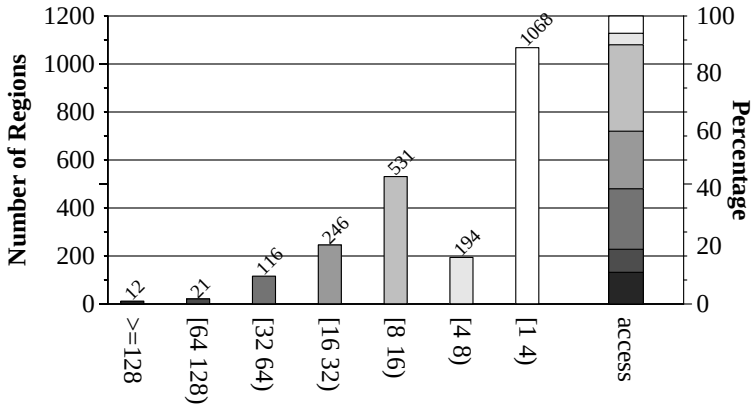


Figure 5.8: Region-based memory write access pattern in PCM for *483.xalancbmk* for 500 million instructions. One region contains 16 contiguous blocks. X-axis shows the number of region access times ($[M\ N)$ means the region is accessed by X times and $M <= X < N$). Very few regions are accessed frequently (e.g., only 12 regions are accessed more than 128 times).

C is delayed by servicing request 'write A' for 3 units. Therefore, the LLC performance miss penalty of 'clean data B' takes one time unit while the LLC performance miss penalty of 'dirty data A' takes 4 units: 1 unit for reading 'A' and 3 unit for delaying 'C'. Assuming the PCM read/write energy is $2/8$ pJ/bit. Then the energy miss penalty for 'A' and 'B' is (64bytes$\times$8bits$\times$2pJ/bit)+(64bytes$\times$8bits$\times$8pJ/bit)=5120pJ and 64bytes$\times$8bits$\times$2pJ/bit=1024pJ, respectively. Therefore, the miss penalty for dirty data is more significant than the clean data.

Based on the observation, we propose to adapt the cache management technique to reduce the writeback requests. Since the performance and energy cost is more significant for the dirty cache blocks, the system could benefit by keeping frequent writeback cache blocks in the LLC. However, blindly allocating large cache capacity to frequent writeback data can evict the more critical cache blocks that will be re-referenced soon. This will result in performance degradation. Consequently, there are two questions that need to be answered: *(1) are the frequent writeback blocks predictable? (2) what is the optimal cache*

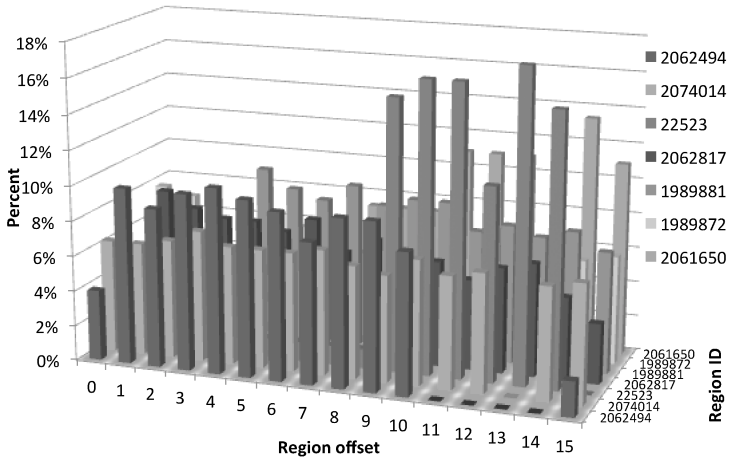


Figure 5.9: 3D view for write access pattern in PCM within seven hot regions for *483.xalancbmk*. The X-axis shows the 16 cache blocks within a region. The Z-axis shows 7 regions that the number of writeback accesses larger than 64.

capacity that should be allocated to frequent writeback data? We performed experiments and have the following two observations:

Observation 1 : The writeback accesses have spatial and temporal locality. A small percentage of regions account for a large percentage of writeback accesses. Within a heavily accessed region, the writeback accesses are clustered.

Figure 5.8 shows access patterns for writeback requests to PCM for the benchmark *xalancbmk* for 500 million instructions. We evaluate the access pattern at the region level. One region includes $1/4$ size of memory page which has 16 contiguous blocks. X-axis shows the number of region access times, such as $[32\ 64)$ means the region is accessed by no less than 32 times and less than 64 times. Y-axis gives the number of regions that correspond to the access times on the X-axis. For instance, the first bar shows there are 12 regions have been accessed more than 128 times. The last bar shows the percentage of number of accesses for each type of region account for all the writeback accesses. We

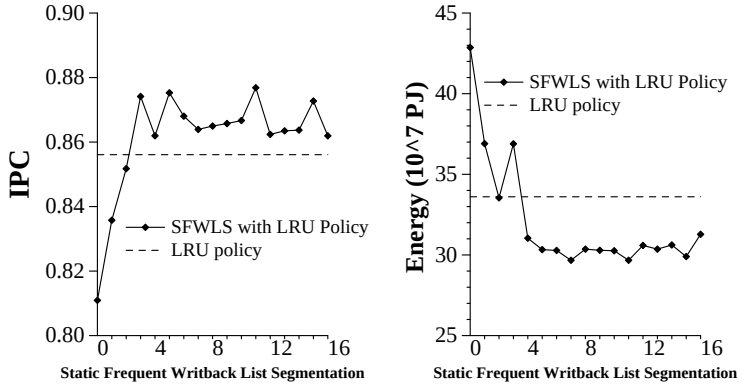
can see the writeback accesses have temporal locality. Less than 18% percent of regions account for 60% writeback accesses. Figure 5.9 shows the $3D$ graph for writeback access pattern within the frequent writeback regions. The X-axis shows the 16 cache blocks within a region. The Z-axis shows seven regions that the number of writeback accesses larger than 64. The Y-axis gives the percentage of total write accesses for each block within the region. We can see the writeback accesses for blocks are clustered within the region.

Based on this observation, we propose a two-stage predictor for frequent writeback cache blocks, at both coarse-granularity and fine-granularity: The *region granularity* prediction predicts the hot region by capturing the spatial locality and temporal locality. The *cache line granularity* prediction identifies the frequent writeback blocks within the hot region.

Observation 2 : The segment size of frequent writeback list for cache set significantly affects the performance and energy consumption for workloads.

The last-level cache set is partitioned into *frequent writeback list* and *non-frequent writeback list*. The *frequent writeback list* consists of frequent writeback cache blocks, while the *non-frequent writeback list* consists of the remaining cache blocks in the set. Figure 5.10 shows the performance and energy impact for various sizes of frequent writeback list for benchmark *perlbench*. For a 16-way LLC, the best segment size for *perlbench* is 11 which generates the best performance and lowest energy cost. We can see the segment size of frequent writeback list do significantly affect the performance and energy consumption.

Based on this observation, we propose to segment the cache set into frequent writeback



(a) SpeedUp (b) Energy

Figure 5.10: The impact on performance and energy for various size of writeback list for *400.perlbench*. For a 16-way LLC, the optimal segmentation size for frequent writeback list is 11.

list and non-frequent writeback list. A segment predictor [33] is used to dynamically learn an optimal size of each list in the set according to the miss penalty for dirty and clean cache blocks.

5.2.2 Policy Design

The WADE technique improves system efficiency by reducing frequent writes to main memory. Figure 5.11 shows the structure of WADE. It uses a frequent write predictor (FWP) to predict LLC blocks that are written back to main memory with high frequency within a certain access interval. The insight of the technique is that frequent writeback data is also highly reused dirty data in the LLC. If frequent writeback data can be stored in the LLC, it can reduce write-induced interference as well as energy consumption of PCM. However, blindly replacing LLC blocks with frequent writeback data can evict more critical cache blocks that have a larger miss penalty, such as clean cache blocks accessed more frequently than the predicted frequent writeback cache blocks. This can lead to performance degradation. In WADE technique, the LLC set is partitioned into frequent writeback list and non-frequent writeback list. A segment predictor [33] is used to intelligently

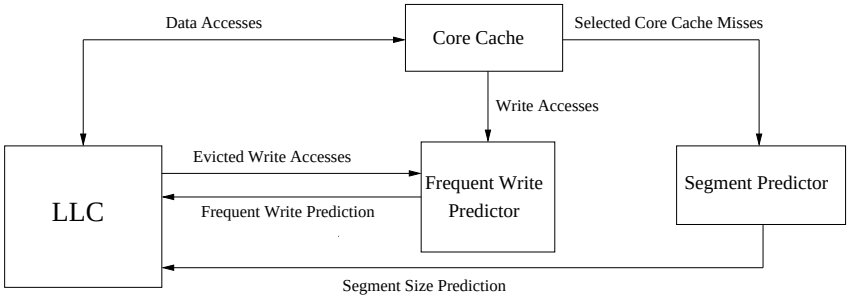


Figure 5.11: System structure

learn the best partition size of each list.

5.2.2.1 Frequent Write Prediction

A frequent write predictor is proposed to keep track of the frequent writeback data and predict the frequent writeback block in the LLC. Figure 5.12 shows the structure of the FWP which is located on chip along with the LLC tag arrays. FWP is organized as a set associative structure. Every m LLC sets map to n FWP sets. Figure 5.13 shows the address mapping scheme for FWP. In our experiment, we set $m = 16$, $n = 4$. This address mapping scheme allows FWP keeping track of the frequent writeback data in region granularity where each region consists of m cache blocks.

Each entry in the FWP set has a partial tag field (PTag), an LRU field, a frequency counter field indicating how often the region data being written back and a set flag field that each flag bit corresponding to each LLC set that map to this FWP set. The set flag field allows the technique to keep track of frequent writeback data at the cache line granularity. Thus, the FWP table keeps track of the frequent writeback data in both coarse granularity and fine granularity: region granularity and cache line granularity. Since applications often have spatial and temporal locality, tracking data in coarse granularity (region granularity) can minimize the capacity overhead as well as improve prediction accuracy.

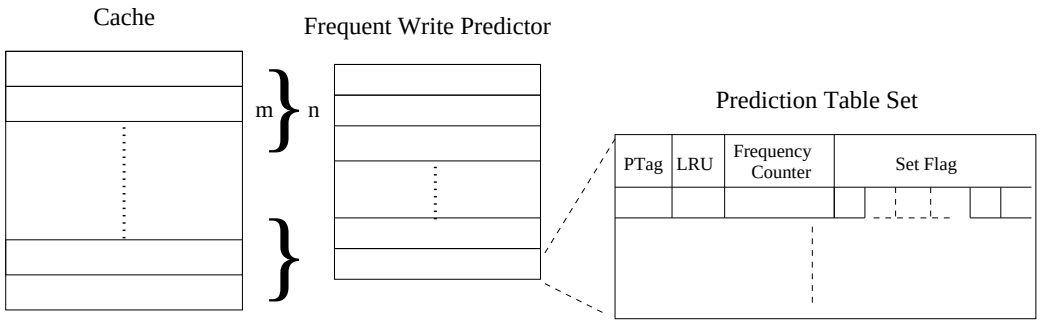


Figure 5.12: Illustration of frequent write predictor. FWP is a set associative structure, each set has multiple entries with multiple fields

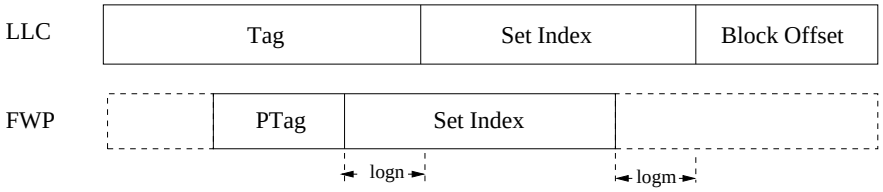


Figure 5.13: FWP address mapping scheme. Every m LLC sets map to n FWP

Making a Prediction For each cache block in the LLC, one Fbit is added for indicating that block is a frequent writeback block. Once a write request accesses the LLC, it will also access the FWP set for partial tag matching. Since correctness of matches is not necessary in the tag array, only 16 bits of tag are stored in the FWP set entry to conserve area and energy. If it is a partial tag hit and the corresponding set flag bit is set, the Fbit for this cache block is set indicating that the cache block is a frequent write cache block. Otherwise the Fbit of the cache block is unset.

Updating Predictor Once a dirty cache block is evicted from the LLC, the FWP is updated. The evicted dirty cache block accesses the FWP. The LRU recency in the corresponding FWP set is updated for each access. On a partial tag hit, the frequency counter value in the entry is increased by 1. The corresponding set flag bit is set to 1. On a miss,

a new entry is allocated in the FWP set. The initial frequency counter value is reset to 0. The corresponding set flag bit is set to 1 while all the other set flag bits in the set flag fields for the newly allocated entry are reset to 0. The replacement candidate is chosen by taking into account both recency and frequency information. The frequency information is used to recognize the frequent writeback region. The recency information can be used to remove the stale data in the FWP table. Assuming the LRU recency value is $R(i)$ where the highest value indicates MRU position and the frequency counter value is $F(i)$. Then the replacement victim is chosen as follows:

$$Victim = arg\min_{i}\{F(i) + \gamma R(i)\} \tag{5.1}$$

The parameter γ gives the weight of $R(i)$. It determines the access interval for computing the frequency for writeback data. The larger the value, the smaller the access interval. If the access interval is too small, it could result in local optimal prediction result instead of global optimal prediction result. If the access interval is too large, the stale data stored in FWP prevent the learning process. In our experiment, we found $\gamma = 4$ gives the best performance.

5.2.3 Frequent Writeback List Cache Segmentation

The LLC set is logically segmented into frequent writeback and non-frequent writeback lists. The cache blocks with the Fbit set belong to the frequent writeback list, the remaining cache blocks belong to the non-frequent writeback list. The segment predictor [33] is used to predict the optimal segment size of frequent writeback list for all sets. Figure 5.14 illustrates the mechanism of the technique. It tries to keep the optimal segment size that minimizes the LLC miss penalties. The technique is decoupled from LLC replacement policy. Any replacement policy can be applied to each list.

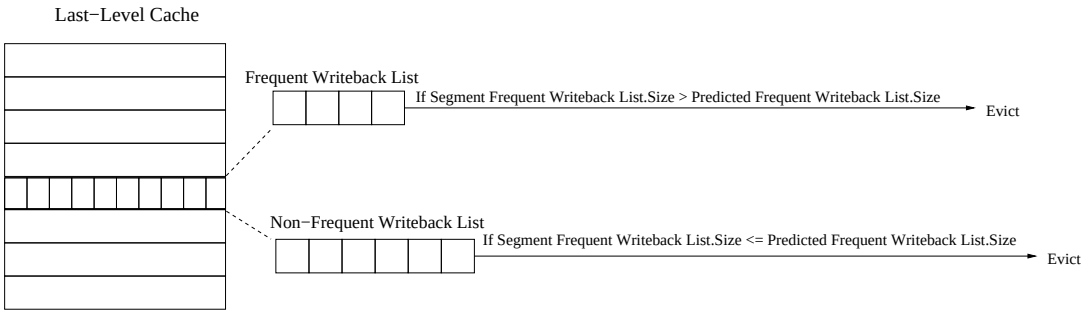


Figure 5.14: The logical view of frequent writeback list segmentation mechanism. Each set is partitioned into frequent writeback list and non-frequent writeback list

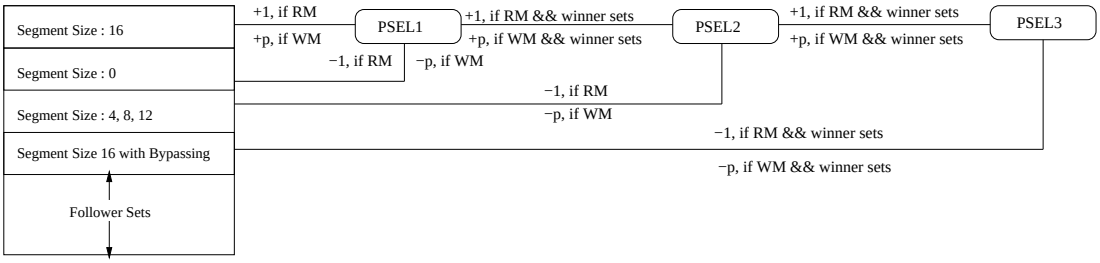


Figure 5.15: The mechanism of segment predictor. It consists of six leader sets with segment size 0, 4, 8, 12, 16 and segment size 16 with bypassing.

Once a request accesses the LLC, all the ways in the set are searched. On a miss, the size of frequent writeback list of the set is calculated. If it is larger than the predicted optimal size, the replacement candidate will be chosen from the frequent writeback list. Otherwise it will be chosen from the non-frequent writeback list.

5.2.3.1 Optimal Segment Size Prediction

The segment predictor [33] uses set duelling to determine optimal segment size. It estimates the miss penalty for any given segment size by always dedicates a few "leader sets" follow that segment size. As shown in Figure 5.15, we evaluate five segment sizes for 16-way associative set: 0, 4, 8, 12, 16. The leader sets use decision tree analysis to pairwise set duel at each level as proposed in [33]. For instance, segment size 8 duel with segment

size 16 in first level. The policy selection counter 1 (PSEL1) increases on a miss in leader sets following segment size 8 and decreases on a miss in leader sets following segment size 16. The PSEL1 estimates which segment size is the winner size in the first level. If size 8 is the winner size, the second level duel will be between segment size 0 and 8. Otherwise, the second level duel will be between size 12 and 16. The process will continue until the optimal segment size is found. In our experiment, we use an out-of-cache segment predictor, that is a set associative structure is added to simulate the sampled leader sets. The LLC sets follows the optimal segment size predicted by the segment predictor.

5.2.3.2 Bypass Incoming Read Blocks

If a block to be placed in a set will not be reused before it is evicted from the set, it should bypass the cache. Bypassing can improve cache efficiency by allocating the capacity to other reused blocks in the cache. Our segment predictor also considers bypassing the read requests. If the predicted optimal segment size is 16, the leader sets with bypassing the read requests duel with the leader sets of segment size 16 without bypassing. The LLC sets will follow the winner policy indicated by PSEL3.

5.2.3.3 Determining Miss Penalty

The traditional cache replacement policy assumes the absolute number of cache misses is fully correlated with memory-related stall cycles [63]. It assumes the same miss penalty for dirty and clean cache blocks. In the traditional set duelling technique, for each leader cache set miss whether the data is dirty or clean, the PSEL is increased/decreased by 1. Our technique is different from previous work in that it is aware of the write inefficiency problem and assign miss penalty according to the type of cache blocks. If a clean cache block is evicted from the leader set, the PSEL is increased/decreased by 1. If a dirty cache cache block is evicted from the leader set, the PSEL is increased/decreased by p, defined as follows:

$$p = 1.5 + 0.5 \times l \tag{5.2}$$

l is defined as:

$$l = W/R \tag{5.3}$$

In the formula, W is the write latency while R is the read latency. For a certain memory system, l is a constant. Then l is quantized into 2 bits value by divided by 8. The larger the value of l, the larger the write latency p. p is measured in steps of 0.5. For each leader set, we add 1 bit even write flag. If p is not a integer, such as $p = 1.5$, then for every two write misses, the PSEL is increased by three.

5.3 Evaluation Methodology for APM Technique

Execution core	4.0GHZ, 1-core/4-core CMP, out of order, 128 entry reorder buffer, 48 entry load queue, 44 entry store queue, 4 width issue/decode
Caches	L1 I-cache: 64KB/2 way, private, 2-cycle 64 bytes block size, LRU L1 D-cache: 64KB/2 way, private, 2-cycle 64 bytes block size, LRU L2 Cache: shared, 64 bytes block size, LRU
DRAM	DDR3-1333, open-page policy, 2-channel, 8-bank/channel, FR_FCFS [68] policy, 32-entry/channel write buffer, drain_when_full write buffer policy

Table 5.2: System configuration

We use MARSSx86 [57], a cycle-accurate simulator for the 64-bit x86 instruction set. The DRAMSim2 [69] simulator is integrated into MARSSx86 to simulate DDR3-1333 system. Table 5.5 lists the system configuration. We model a per-core LLC middle-of-

Name	Technique
SRAM	SRAM-based LLC
STT-RAM	STT-RAM-based LLC
OPT-STT-RAM	STT-RAM-based LLC Assuming symmetric read/write overhead
Sun-Hybrid	Hybrid LLC technique as described in [79]
APM	Adaptive placement and migration based hybrid cache as described in Section 5.2.2

Table 5.3: Legend for various LLC techniques.

Name	Benchmarks
Mix 1	milc gcc xalancbmk tonto
Mix 2	gamess soplex libquantum perlbench
Mix 3	gcc sphinx3 GemsFDTD tonto
Mix 4	lbm mcf cactusADM GemsFDTD
Mix 5	zeusmp bzip2 astar libquantum
Mix 6	mcf soplex zeusmp bwaves
Mix 7	omnetpp lbm cactusADM sphinx3
Mix 8	bwaves libquantum mcf GemsFDTD
Mix 9	omnetpp cactusADM tonto gcc
Mix 10	soplex mcf bzip2 gcc
Mix 11	perlbench sphinx3 libquantum lbm

Table 5.4: Multi-Core workloads

the-road stream prefetcher [76] with 32 streams for each core. The prefetcher looks up the stream table at each LLC request for issuing eligible prefetch requests. The LLC is configured with multiple banks. Requests to different banks are serviced in parallel. Within the same bank, requests are be pipelined. The LLC is implemented with single-port memory bitcell. We obtain STT-RAM and SRAM parameters using NVSim [12] and CACTI [51] as shown in Table 5.1.

The SPEC CPU2006 [19] benchmarks are used for the evaluation. We evaluate five LLC techniques based on the same area configuration. Table 5.3 shows the legends for these techniques referred to in the graphs that follow. The OPT-STT-RAM technique assumes write operations have similar access latency to read operations which is the op-

timistic case. The Sun-Hybrid [79] technique assumes that a cache block that has been consecutively written to the LLC twice is a write-intensive block. The technique migrates the write-intensive blocks to SRAM lines for reducing the write operations to STT-RAM lines. The APM technique is our proposed adaptive block placement and migration policy based hybrid cache technique as described in section 5.2.2. We modify MARSSx86 to support all types of LLC listed in Table 5.3.

5.3.1 Single-Core Workloads and LLC Configuration

Of the 29 SPEC CPU2006 benchmarks, 22 can be compiled and run with our infrastructure. We use all 22 of these benchmarks for evaluation including both memory-intensive benchmarks and non-memory-intensive benchmarks. For each workload, we simulate 250 million instructions from a typical phase identified by SimPoint [74].

Various LLC techniques are evaluated with the same area. A 2MB SRAM has similar area to a 6MB STT-RAM. Thus, we evaluate a 16-way SRAM with 2MB capacity and 24-way STT-RAM/OPT-STT-RAM with 6MB capacity. The hybrid cache design for the APM technique has 16 STT-RAM lines and 2 SRAM lines in each set and hence we evaluate a 4.5MB APM hybrid cache which has the same area with a 2MB SRAM. In the Sun-Hybrid technique, each cache set allocates 1 SRAM line. Thus we evaluate a 20 STT-RAM lines and 1 SRAM line hybrid cache with a 5.25MB capacity for Sun-Hybrid technique. We implement a 1MB SRAM cache bank and 2MB STT-RAM cache bank for a single-core configuration yielding the best trade off of access latency and bank level parallelism. If the capacity of SRAM is smaller than 1M, it is configured as one bank.

5.3.2 Multi-Core Workloads and LLC Configuration

We use quad-core workloads for evaluation. Table 5.6 shows eleven mixes of SPEC CPU2006 benchmarks with a variety of memory behaviors. For each mix, we run the experiment with 1 billion instructions total for all four cores starting from the typical

phase. Each benchmark runs simultaneously with others. For the multi-core configuration, we evaluate a 16-way SRAM with 8MB capacity, 24-way STT-RAM/OPT-STT-RAM with 24MB capacity, 16-way STT-RAM and 2-way SRAM Hybrid APM technique with 18MB capacity, and 21MB 20-way STT-RAM and 1-way SRAM Sun-Hybrid technique. In the multi-core configuration, we use 2MB SRAM cache bank and 4MB STT-RAM cache bank which yield best performance. If the capacity of SRAM is smaller than 2M, it is configured as one bank.

5.4 Evaluation Methodology for WADE Technique

Execution core	4.8GHZ, 1-core/ 4-core CMP, out of order 256 entry reorder buffer, 4 width issue/decode 15 stages, 256 physical registers
Caches	L1 I/D-cache: 64KB, 2 way, private 64 bytes block, 2-cycle, LRU, L2 Cache: 2MB/1core, 8MB/4core 16-way, shared, 64 bytes block, 14-cycle
PCM	1 channel/1core, 2 channels/4-core CMP 8 banks per channel, 8K bytes row buffer 32-entry write buffer per channel read prioritize write scheduling policy
PCM Timing	row hit (clean miss, dirty miss) =200 (450, 5000) cycles
PCM Energy	array read (write) = 2.47 (16.82) pJ/bit row buffer read (write) = 0.93 (1.02) pJ/bit

Table 5.5: System configuration. Memory timing and energies are adapted from [41]

We use the MARSSx86 [57], a cycle-accurate simulator for the x86-64 architecture. We modify the DRAMSim2 [69] simulator to simulate PCM memory and incorporate it into MARSSx86. The system configuration is shown in Table 5.5. We use the SPEC CPU 2006 [19] benchmarks for the evaluation. Each benchmark is run with the first *ref* input

Name	Benchmarks
Mix 1	milc gcc xalancbmk tonto
Mix 2	GemsFDTD namd bzip2 gamess
Mix 3	gamess soplex libquantum perlbench
Mix 4	zeusmp lbm xalancbmk calculix
Mix 5	gamess milc namd soplex
Mix 6	astar lbm gobmk calculix
Mix 7	soplex calculix tonto lbm
Mix 8	lbm mcf cactusADM GemsFDTD
Mix 9	mcf soplex zeusmp bwaves
Mix 10	lbm milc astar libquantum
Mix 11	xalancbmk lbm perlbench tonto

Table 5.6: Workloads

provided by the *runspec* command.

5.4.1 Single-Thread Workloads

We use 15 memory intensive benchmarks for this study. A 2MB LLC is simulated for the single thread workloads. For each workload, we made a checkpoint by running the benchmark to a typical phase identified by SimPoint [74]. Then we run the experiment starting from the checkpoint,the infrastructure simulates 200 million instructions from the checkpoint.

5.4.2 Multi-Core Workloads

Table 5.6 shows eleven mixes of SPEC CPU 2006 benchmarks chosen four at a time with a variety of memory behaviors. We use these mixes for quad-core simulations. Each benchmark runs simultaneously with the others. For each mix, we made a checkpoint by running the one of the memory intensive benchmarks to a typical phase . Then we run the experiment for 1 billion instructions total for all four cores starting from the checkpoint. We simulate an 8MB shared LLC for the multi-core workloads.

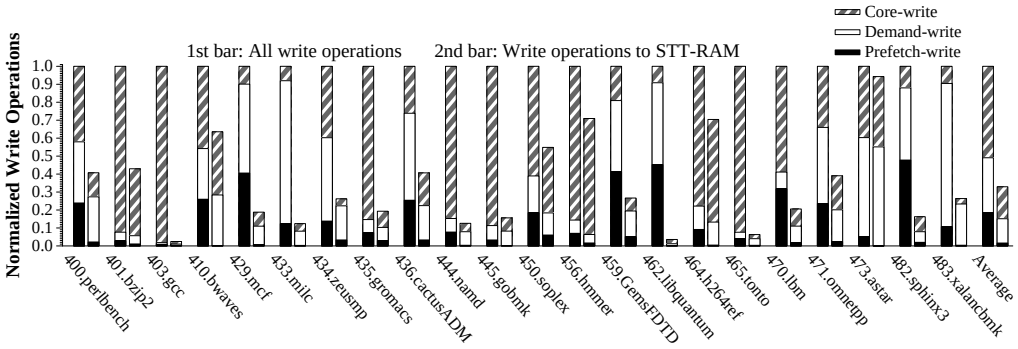


Figure 5.16: The distribution of write accesses to STT-RAM lines in APM LLC for single-core applications

5.5 Evaluation Results for APM Technique

5.5.1 Single-Core Evaluation Results

5.5.1.1 Reduced Writes Evaluation

The APM technique allows SRAM lines to service as many write requests as possible, thus reducing write operations to STT-RAM lines. Figure 5.16 shows the distribution of write operations to STT-RAM lines in the APM technique normalized to all write operations to the LLC for single-core workloads. We can see the APM LLC reduces write operations to STT-RAM lines for each type of write accesses. In APM LLC, only 32.9% of the total LLC write requests are serviced by the STT-RAM portion, significantly reducing the write overhead of the STT-RAM portion and translating into performance improvement and power reduction of the LLC.

5.5.1.2 Performance Evaluation

Figure 5.17 shows the speedup for various techniques compared with baseline technique which is 2MB SRAM LLC. The 6MB STT-RAM LLC has similar area to 2MB SRAM LLC. It improves the performance by 6.2% on average due to the increased capacity. Most of the benchmarks can benefit from the increased capacity of STT-RAM.

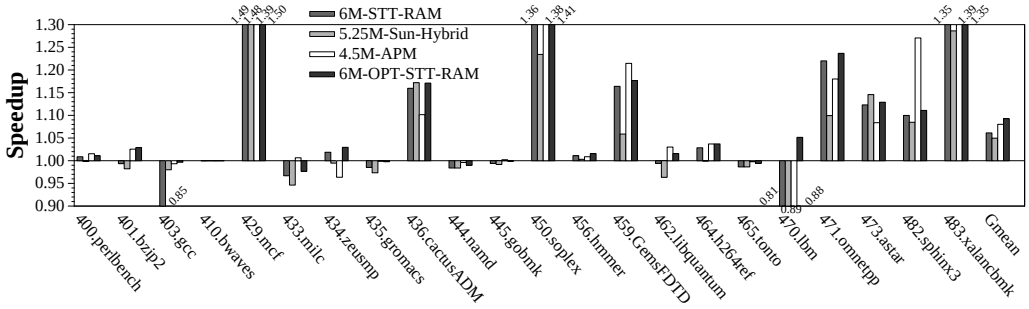


Figure 5.17: The comparison of IPC for single-core applications (normalized to 2M SRAM LLC)

However, several benchmarks such as `gcc`, `milc`, `libquantum` and `lbm` suffer more from the large write overhead of STT-RAM. The OPT-STT-RAM LLC assumes symmetric read/write latency meaning large write-induced interference to read request is removed. It yields a geometric mean speedup of 9.3%. The performance difference between OPT-STT-RAM and STT-RAM is caused by the long write latency of STT-RAM. The 4.5MB APM LLC reduces the write overhead of the STT-RAM portion, delivering a geometric mean speedup of 8.0%. It yields even higher speedup for benchmarks `462.libquantum`, `482.sphinx3`, and `483.xalancbmk` than the 6MB OPT-STT-RAM LLC. Because those workloads generate a large number of dead blocks or inaccurate prefetch blocks in the LLC and hence reducing the LLC efficiency. The APM technique reduces the LLC pollution cased by dead blocks and inaccurate prefetch blocks, thus improving the LLC efficiency for those workloads. The 5.25MB Sun-Hybrid LLC improves the performance by 5.0% on average.

5.5.1.3 Power Evaluation

Figure 5.18 shows the normalized power consumption for various techniques due to leakage power, dynamic power caused by reads and dynamic power caused by writes. The baseline technique is a 2MB SRAM. For the SRAM technique, the leakage power dom-

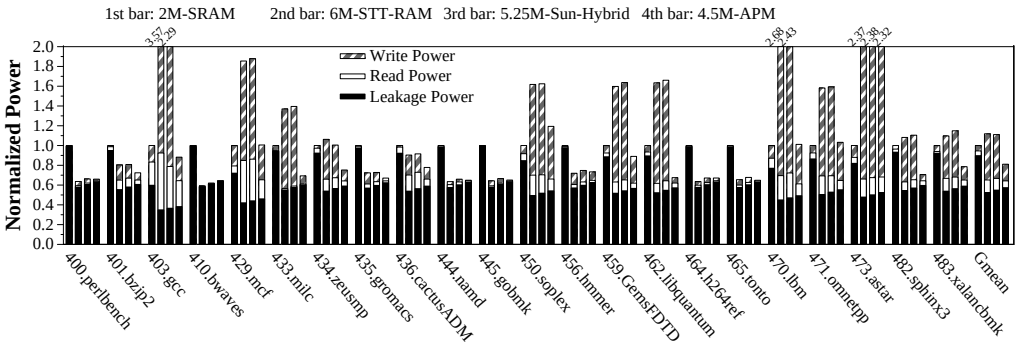


Figure 5.18: The power breakdown for single-core applications (normalized to 2MB SRAM)

inates the total power consumption. The STT-RAM technique consumes lower leakage power. However, the dynamic power caused by writes is significantly increased due to the large write energy of STT-RAM. Thus the STT-RAM technique increases the overall power consumption by 11.9% on average. The APM technique reduces write operations to the STT-RAM portion, thus reducing the dynamic power caused by writes. It reduces the overall power consumption by 18.9% on average compared with the baseline. The Sun-Hybrid technique does not significantly reduce the write power because the Sun-Hybrid technique does not reduce write operations to STT-RAM caused by LLC replacement.

5.5.1.4 Extra LLC Traffic Evaluation

The APM technique migrates blocks between SRAM lines and STT-RAM lines. Migrating blocks from SRAM lines to STT-RAM lines causes extra cache traffic. We evaluate the LLC traffic caused by migration. Migration causes only 3.8% extra LLC traffic. Most of the blocks evicted from SRAM ways are dead blocks, thus only a small number of distant-read-range blocks need to be migrated to STT-RAM. Thus, the small percentage of traffic caused by migration will not cause significant traffic overhead.

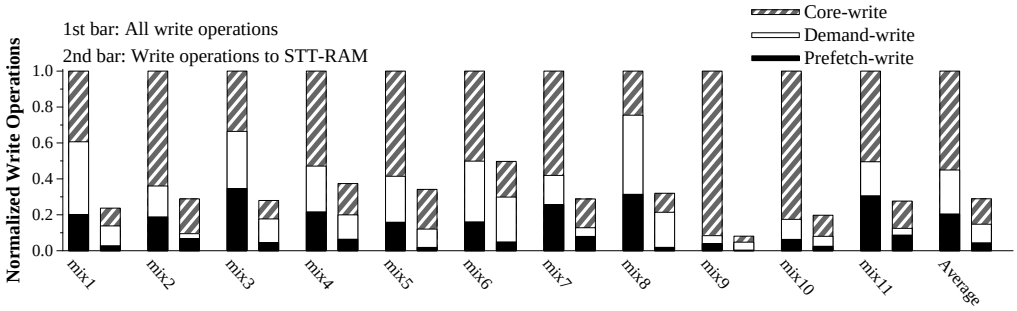


Figure 5.19: The distribution of write accesses to STT-RAM lines in APM LLC for multi-core applications

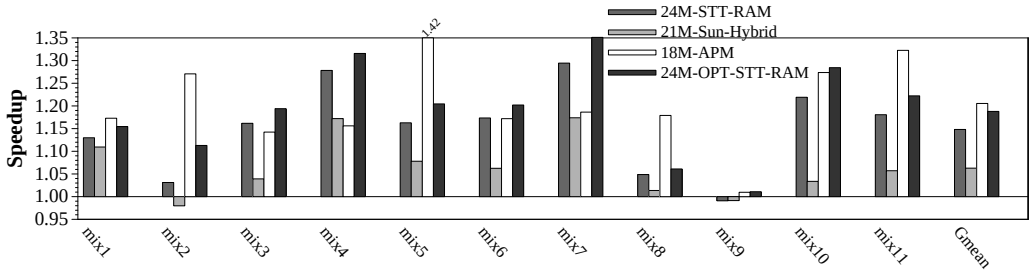


Figure 5.20: The comparison of IPC for multi-core applications (normalized to 8MB SRAM)

5.5.2 Multi-Core Evaluation Results

5.5.2.1 Reduced Writes, Endurance, Performance and Power Evaluation

Figure 5.19 shows the distribution of write operations to STT-RAM lines normalized to all write operations to LLC for the multi-core workloads. The APM technique reduces write operations to the STT-RAM portion to 28.9% on average of the total number of write operations.

Figure 5.20 shows the speedups of the various techniques for the multi-core workloads normalized to 8MB SRAM LLC. The 24MB STT-RAM LLC improves performance by 14.8% on average. Removing write-induced interference caused by asymmetric writes

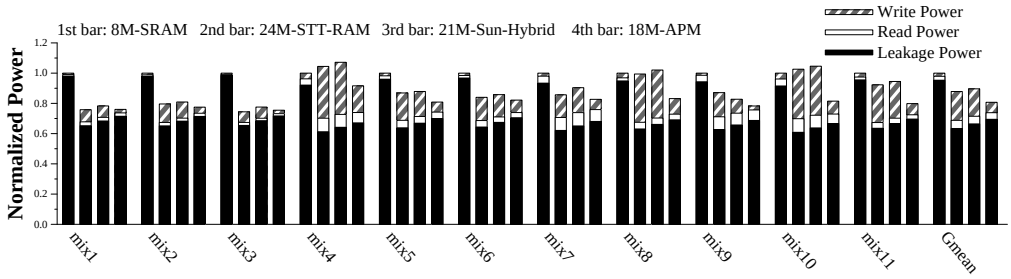


Figure 5.21: The LLC power breakdown for multi-core applications (normalized to 8MB SRAM)

improves the average performance by 18.7% in the OPT-STT-RAM LLC. The 18M APM technique reduces write overhead of the STT-RAM portion. It achieves a geometric mean speedup of 20.5% which is higher than the 24MB OPT-STT-RAM LLC. In multi-core workloads, the large number of dead or inaccurately prefetched blocks generated from one workload can also negatively affect the performance of other workloads, significantly reducing performance. The APM technique reduces cache pollution caused by dead blocks and inaccurately prefetched blocks. Our evaluation shows the 18MB APM LLC yields better performance and fewer misses for 5 out of 11 multi-core workloads compared with 24MB OPT-STT-RAM LLC.

Figure 5.21 shows the distribution of normalized power consumption for various techniques. The baseline is the 8MB SRAM cache. For a large LLC, the majority of power consumption comes from leakage power. The 24MB STT-RAM technique reduces overall power consumption to 87.8% of baseline due to low leakage power. The APM technique reduces dynamic power caused by the STT-RAM write operations. Thus, it further reduces power consumption to 80.7% of baseline on average.

5.5.2.2 Prediction Evaluation

We evaluate the access pattern predictor using false positive rate and coverage. Mispredictons can be false positives and false negatives. False positives are more harmful for

two reasons: mispredicting a live block as a dead block can cause bypass or eviction of a live block from LLC early and generate LLC misses, and mispredicting a non-write-burst request as a write-burst request can cause extra migrations between STT-RAM lines and SRAM lines. False positive rate is measured as the number of mispredicted positive predictions divided by the total number of predictions.

Among the 11 multi-core workloads, the access pattern predictor yields a low false positive rate ranging from 2.1% to 14.8%, with a geometric mean of 8.3%.

The access pattern predictor achieves an average coverage of 71.7%. Thus, the majority of dead blocks and write burst blocks can be predicted by the access pattern predictor.

5.5.2.3 Memory Energy Evaluation

Figure 5.22 shows the memory energy evaluation results normalized to 8MB SRAM LLC. The 24MB STT-RAM LLC reduces the average memory energy to 72.9% of the baseline. The 18MB APM LLC technique reduces average memory energy to 72.4%.

Compared with 24MB STT-RAM LLC, the 18MB APM LLC increases average memory traffic by 5.6% due to its smaller capacity. However, it does not increase the dynamic energy consumption because it consumes less activation/precharge energy. The APM LLC achieves a higher DRAM row-buffer hit rate for write requests than the STT-RAM LLC which can reduce the activation/precharge energy. The large LLC can filter the locality of dirty blocks, so the dirty blocks have low spatial locality when they are evicted from the LLC and written back to the main memory. However, in the APM LLC, a significant fraction of dirty blocks are written back to the main memory when they are evicted from the SRAM portion where the small capacity of SRAM allows the evicted dirty blocks to have higher spatial locality. Our evaluation result shows the DRAM row-buffer hit rates for writes are 21.1% and 35.6% for 24M STT-RAM LLC and 18M APM LLC respectively.

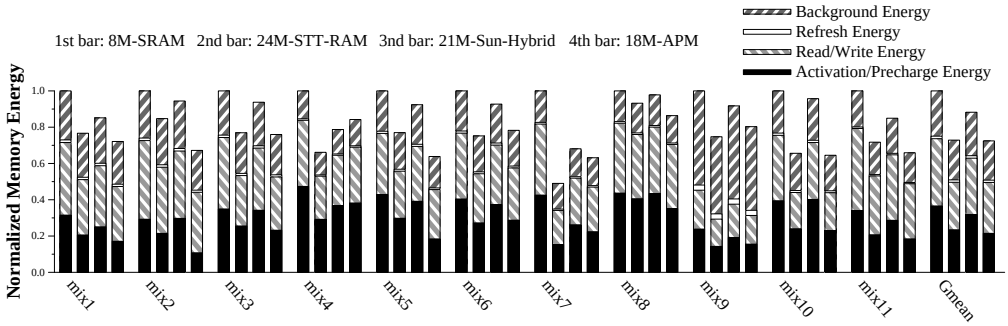


Figure 5.22: The memory energy breakdown for multi-core applications (normalized to 8MB SRAM)

5.5.3 Storage Overhead and Power

The technique uses an access pattern predictor to predict the dead blocks and write burst blocks which cause extra storage and power overhead.

5.5.3.1 Storage Overhead

Each cache block in the LLC adds 1 bit for representing whether it is a dead block, using 9.2K storage total. For the pattern predictor, each prediction table has 4,096 entries with a 2-bit counter in each entry. There are 6 tables for the skewed structure for the dead block prediction table and write burst prediction table using a total of 6KB of storage. In the pattern simulator, one simulated set corresponding 32 LLC sets, each simulated set has 12-entry 16-bit partial read PC, 12-entry 16-bits partial tag, 12-entry 4-bit LRU position, 12-entry 1-bit valid flag, 4-entry 16-bits partial write PC and 4-entry 2-bits write LRU position. For a 4.5MB hybrid cache, it consumes 8.5K storage. Thus, the storage overhead for single-core configuration is 6K+8.5K+9.2K=23.7K which is only 0.53% capacity overhead of the hybrid 4.5MB LLC. For the quad-core configuration, the storage overhead of the APM technique is 6K+8.5K$\times$4+9.2K$\times$4=84.8K, which is 0.43% of the hybrid 18MB cache capacity.

5.5.3.2 Power Overhead

We evaluate the power overhead of our technique using NVSim [12] and CACTI [51]. For the single-core configuration, the extra dynamic and leakage power consumed by the access pattern predictor is 1.6% and 1.9% of the LLC dynamic and leakage power respectively. It induces a power overhead of 1.8% of the total LLC power consumption. For the quad-core system configuration, the extra dynamic and leakage power consumed by the access pattern predictor is 1.1% and 0.60% of the LLC dynamic and leakage power respectively. The overall power overhead caused by the access pattern predictor is 0.76% of the overall LLC power consumption for quad-core configuration.

5.6 Evaluation Results for WADE Technique

5.6.1 Single-Core Evaluation Results

5.6.1.1 Performance Evaluation

We evaluate three cache replacement polices: LRU, WADE with LRU and Memory Level Parallelism (MLP) aware cache replacement technique [63]. The MLP technique takes into account the memory level parallelism dependent cost differential between different misses. The replacement decision is made by considering the MLP-based cost for each cache miss as well as the recency information. The baseline technique is LRU replacement policy. Our technique segments the cache set into two lists. Within the list, any replacement policies could be applied. So it is decoupled with LLC replacement polices. We use LRU replacement policy with our techniques for simplicity. Figure 5.23 shows the performance evaluation results for single core applications. MLP provides a speedup on some benchmarks and a slow-down on others, resulting in a geometric mean speedup of approximately 0.6%. The long write latency in the PCM system makes it hard to learn the memory level parallelism cost, thereby the MLP replacement policy does not perform

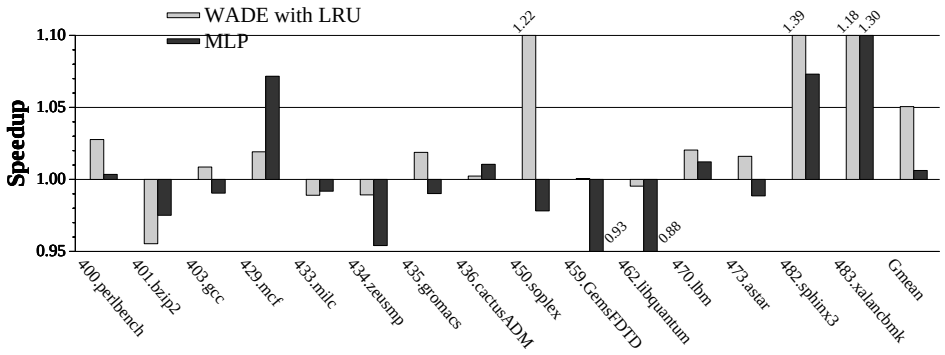


Figure 5.23: The comparison of IPC for single-core applications (normalized to LRU)

well in the context of PCM system. The WADE technique delivers a geometric mean speedup of 5.1%. The technique significantly improves system performance for benchmark *450.soplex*, *482.sphinx3* and *483.xalancbmk* by 22%, 39% and 18%. Because the writeback requests for these three benchmarks are highly reused. For benchmarks that do not benefit from our techniques, there are two categories: first, they do not have significant highly reused access requests such as for streaming benchmarks, *libquantum* and *milc*, the writeback requests are not re-written frequently. Second, the frequent writeback requests are hard to predict mainly because they do not have good spatial and temporal locality, such as *436.cactusADM*.

5.6.1.2 Reduced Write Requests Evaluation

The WADE technique takes into account the disparity in miss penalty of clean data and dirty data. It keeps an optimal size of frequent writeback list in the LLC. Thereby it can reduce the writeback requests to the PCM. Figure 5.24 shows the writeback requests normalized to LRU policy. The MLP technique only reduces 0.05% writeback requests compared with LRU policy. The WADE technique reduces 16.5% writeback requests on average. This large percent of writeback requests reduction leads to performance improvement and energy reduction. It can also improve the endurance of the PCM based

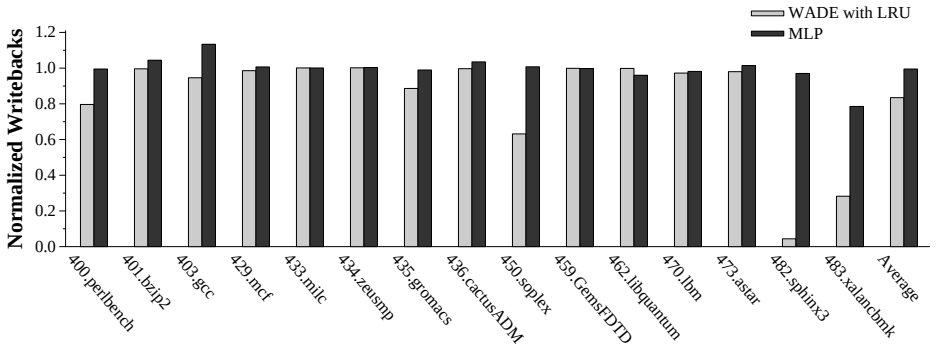


Figure 5.24: The number of writeback requests to PCM for single-core applications (normalized to LRU)

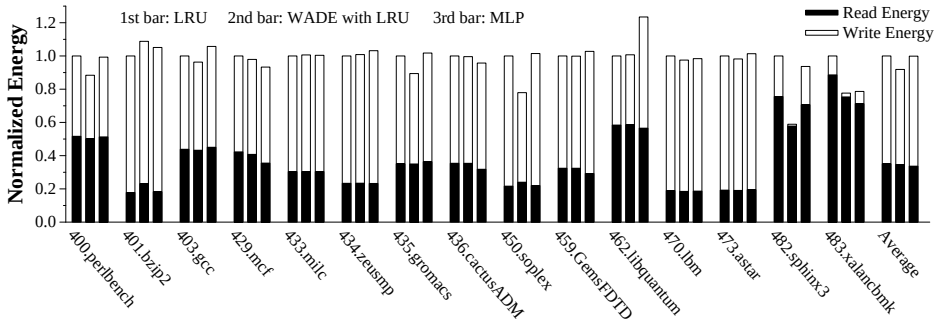


Figure 5.25: The comparison of energy consumption in PCM for single-core applications (normalized to LRU)

main memory. Compared with figure 5.23, we can see for the benchmarks that have large percent of reduced writeback requests also have significant performance improvements.

5.6.1.3 Energy Evaluation

The obvious reduction in the writeback requests can lead to reduced energy consumption in PCM based main memory. Figure 5.25 shows the energy evaluation results for various techniques. The figure shows the energy consumption normalized to LRU policy. It also gives the percentage of read energy and write energy consumption for each workload. In the PCM based main memory, the write energy consumption dominates the main

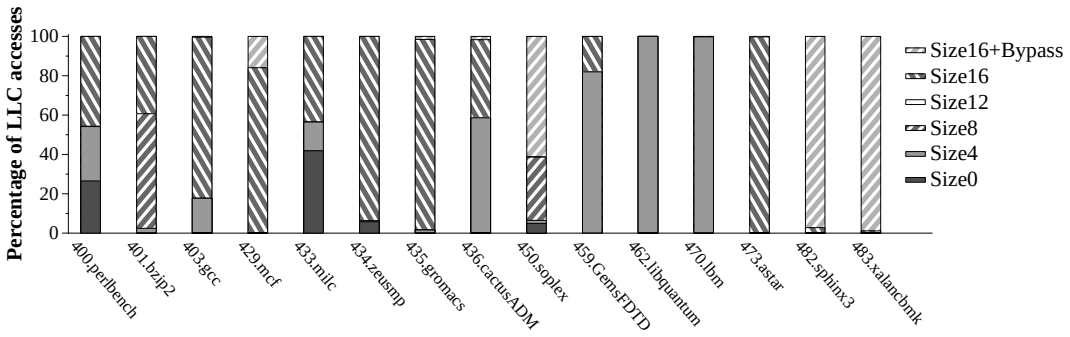


Figure 5.26: Runtime predicted best frequent writeback list size

memory energy consumption. It accounts for about 65% of all main memory consumption in the LRU policy. The WADE technique achieves an energy reduction by 8.1% on average. The MLP technique only reduces the energy by 0.01%. We can see most of the energy reduction of our techniques comes from the write energy reduction. The average read energy consumption for WADE technique is similar with LRU.

5.6.1.4 Dynamic Segment Size

Figure 5.26 shows the runtime predicted best frequent writeback list size for each of the benchmarks. Benchmarks *483.sphinx3* and *483.xalancbmk* are thrashing workloads that benefit from bypassing incoming read blocks. Segment size 16 dominates the running phase of benchmarks *403.gcc, 429.mcf, 434.zeusmp, 435.gromacs,*and 473.astar. The runtime predicted best segment size of benchmarks *462.libquantum* and *470.lbm* is 4. The running phase of other benchmarks go through various segment sizes.

5.6.2 Multi-Core Evaluation Results

The write problem is worse in multi-core system since the performance of an application is affected not only by its own write requests but also by write requests from other applications.

Figure 5.27 shows the speedup achieved by various techniques on the multi-core work-

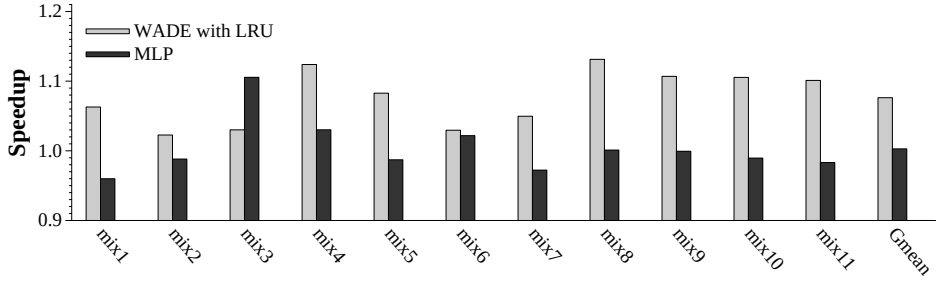


Figure 5.27: The comparison of IPC for multi-core applications (normalized to LRU)

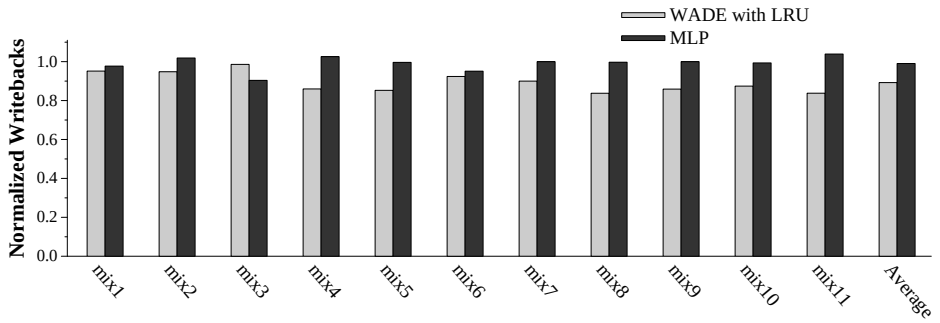


Figure 5.28: The number of writeback requests to PCM for multi-core applications (normalized to LRU)

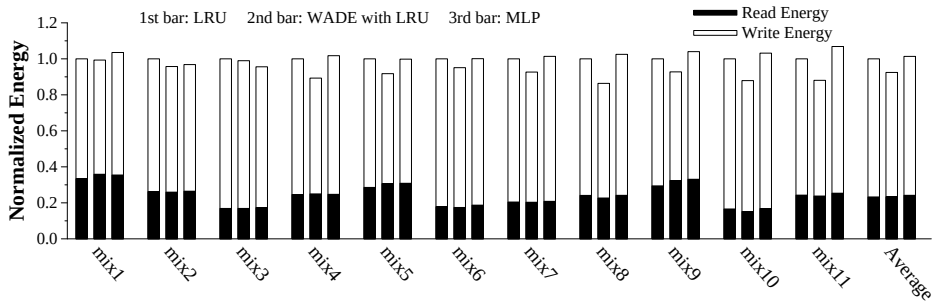


Figure 5.29: The comparison of energy consumption in PCM for multi-core applications (normalized to LRU)

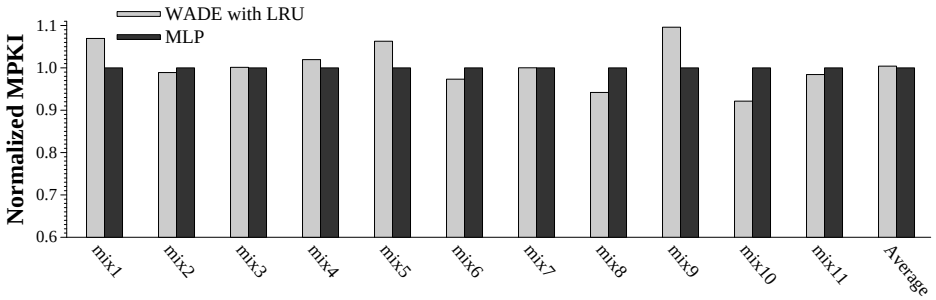
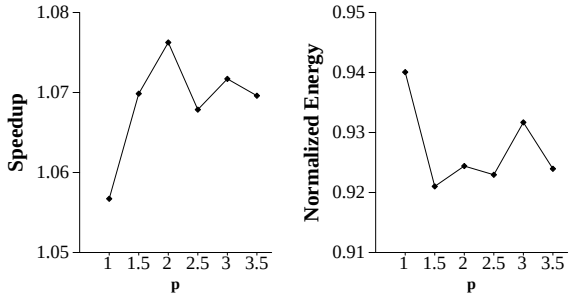


Figure 5.30: LLC misses per kilo-instruction (MPKI) for multi-core applications (normalized to LRU)

loads with an 8MB last-level cache. The speedups are still normalized to a default LRU cache. The normalized speedup for WADE technique over all 11 workloads ranges from 2.2% to 13.1% for the WADE, with a geometric mean speedup of 7.6%. The technique significantly improves the system performance for five workloads by more than 10%. The MLP technique only yields a geometric mean speedup of 0.3%.

Figure 5.28 shows the normalized writeback requests evaluation results for multi-core application. The WADE technique achieves a writeback requests reduction by 10.9% on average. Figure 5.29 shows the energy evaluation results normalized to LRU policy. The WADE technique reduces energy by 7.6% on average.

We also evaluate the misses per 1000 instructions (MPKI) for multi-core workloads. Figure 5.30 shows the MPKI for various techniques normalized to LRU policy. The average normalized MPKIs are 1.00 for WADE, and 0.99 for MLP. We can see the WADE technique does not reduce the miss rate. In WADE technique, the performance benefits actually come from the reduced write requests which generate a large write-induced interference.



(a) Speedup (b) Energy

Figure 5.31: The impact on performance and energy for parameter p

5.6.3 Sensitivity Study

5.6.3.1 Miss Penalty Sensitivity Study

An LLC miss for dirty cache block is more harmful than for a clean cache block. Our technique assigns different miss penalty according to the type of data. The miss penalty for clean data is set to 1 while the miss penalty for dirty data is p. In our experiment setting, we get $p = 2$ calculated by equation (2). We also did an experiment to test the change in performance and energy when p ranges from 1 to 6. Figure 5.31 shows the performance speedup and energy consumption for various values of p in multi-core workloads in WADE technique. We can see the performance and energy consumption varies significantly with different values of p. The best performance is achieved when $p = 2$, and the lowest energy consumption when $p = 1.5$. Generally, the p value that gives better performance is also the value that yields lower energy, is because the reduced write requests could lead to both performance improvement and energy reduction. In our experiment, we choose $p = 2$.

5.6.3.2 Cache Size Sensitivity Study

Figure 5.32 and 5.33 show the performance and writeback reduction evaluation results with various cache sizes. We evaluate LRU and WADE LLC replacement policies with cache sizes 2M, 4M and 8M. Compared with the LRU replacement policy with the

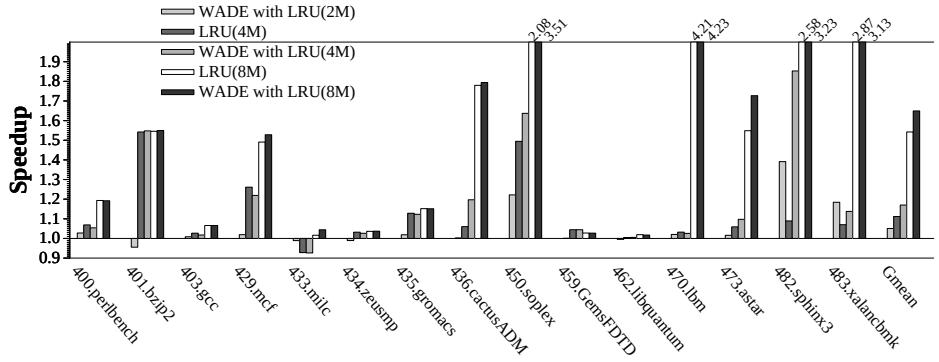


Figure 5.32: Performance evaluation with various cache size (normalized to LRU with 2M LLC size)

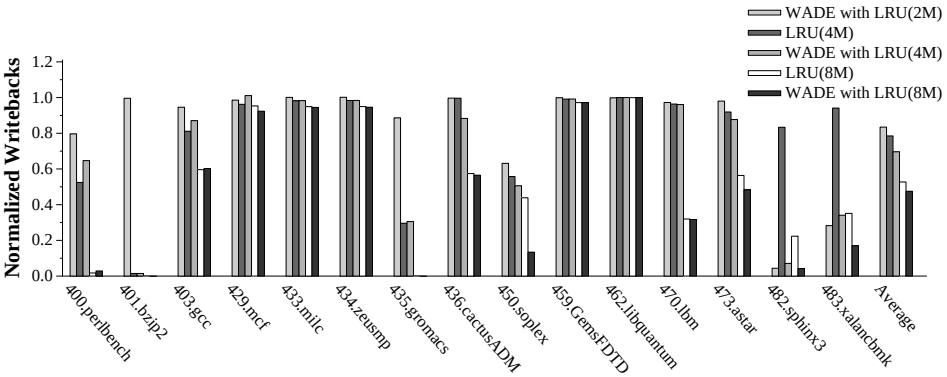


Figure 5.33: The number of writeback requests to PCM with various cache size (normalized to LRU with 2M LLC size)

same capacity 2M, 4M and 8M cache sizes, the WADE technique improves the system performance by 5.1%, 5.4% and 6.9% and reduces the writeback requests to PCM by 16.5%, 11.4% and 9.9% respectively.

5.6.4 Storage and Power Overhead

5.6.4.1 Storage Overhead

The technique uses a frequent write predictor (FWP) and an optimal segment predictor. For the FWP, every 16 LLC sets map to 4 FWP sets. Each FWP set has 6 entries. Each

entry in the set has a 16-bit partial tag field, a 3-bit LRU field, a 6-bit frequent counter field, and a 16-bit set flag field. For each cache block in the LLC, we add one bit to represent whether it is a frequent writeback block. The FWP consumes extra state equivalent to about 0.95% of LLC capacity. We use an out-of-cache segment predictor. This set associative structure is added to simulate sampled leader sets. It uses four types of leader sets as shows in figure 5.15. For each type of leader set, one set is sampled for every 128 LLC sets. Each leader set has one bit even write counter. Each entry in the leader set has 16-bit partial tag field, 1 Fbit field, and 3-bit LRU fields. The segment predictor uses three 12-bit PSEL counters. Thus, it consumes less than 0.13% of LLC capacity. All together, The WADE technique takes about 1% of LLC capacity.

5.6.4.2 Power Overhead

We use CACTI [51] to measure the potential impact of the segment predictor and frequent write predictor on power. The segment predictor is modeled as a tag array of extra LLC sets. We model the LLC both with and without the extra cache sets, and report the difference of the tag power between the two. We model the frequent write predictor as a tag array of a cache, with only the tag power being reported. A 2MB LLC in a single-core configuration consumes $1.99W$ power. The segment predictor consumes only $0.0025W$ dynamic power which is only 0.13% of LLC power consumption. The power for frequent write predictor is $0.024W$. The total power for structures required by the WADE technique is about 1.3% of LLC power. An 8M LLC in a multi-core configuration consumes $3.73W$. The structures needed by the WADE technique take $0.035W$ which is 0.93% of LLC power. Although the segment predictor and frequent write predictor consume extra power, the WADE technique reduces the execution cycles of applications, thus reducing the leakage energy of LLC.

6. CONCLUSIONS

Recall the thesis statement from the introduction:

Programs exhibit significant performance variance in their access to microarchitectural structures. To the extent that this variance is predictable, it can be exploited to improve processor design.

In this dissertation, we have analyzed three types of performance variance: performance variance caused by microarchitectural structures, performance variance caused by phase change and performance variance caused by operation types. By exploiting the three types of performance variance, we propose various techniques to improve processor design. In this section, we review the contribution of our techniques.

6.1 Developing Performance Model by Exploring Performance Variance

In this dissertation, we demonstrate how to develop a performance model for branch predictor using real systems. The technique perturbs benchmark executables to yield a wide variety of performance points without changing program semantics or other important execution characteristics. By observing the behavior of the benchmarks over a range of branch prediction accuracies, we can estimate the impact of a new branch predictor by simulating only the predictor and not the rest of the microarchitecture.

Using measurements of the Intel Xeon E5440 Processor, we quantify the impact of branch prediction on a set of benchmarks, developing regression models that estimate the performance given by changes in the branch predictor. We incorporate these models into a simulator allowing us to estimate the impact of several branch predictors.

This study points the way to future work on estimating the impact of other microarchitectural structures. We demonstrate the potential for interferometry to estimate the impact of L1 and L2 caches by perturbing data layouts.

6.2 Reducing Write-induced Interference by Exploring Performance Variance

In memory systems, write requests can cause significant performance loss by increasing memory access latency for subsequent read requests targeting the same device.

In the dissertation, we propose to use a rank idle time predictor to predict when a rank will have significant idle time. “Rank idle” means that there will be no read request for this rank that will be delayed by scheduling writeback events. The scheduled write requests can be written back during this idle rank period. We incorporate the rank idle time predictor into the parallelism-aware LLC scheduling technique and propose a prediction driven parallelism-aware LLC writeback technique. The proposed technique applies to the DRAM system that maps the rank and channel into the higher order bits than the column in the physical address. Write-induced interference is significantly reduced by our technique.

We also propose a decoupled last-write predictor guided LLC writeback technique. It uses a last-write predictor to predict last-write blocks in LLC. The predicted last-write blocks are exposed to the memory controller for scheduling. Our technique can balance the memory bandwidth and effectively expands the scheduling space of the memory controller, thus significantly reducing write-induced interference. It is completely decoupled from LLC replacement policy. Our techniques are evaluated for various DRAM configuration by using MARSSx86 Simulator together with DRAMSim2. Experiment results show a significant performance improvement over traditional writeback technique.

6.3 Reducing NVM Write Overhead by Exploring Performance Variance

Write-induced interference in the memory system can significantly degrade performance. This large write overhead is a more severe problem in NVM-based memory. We propose techniques to mitigate the write overhead in NVM-based memory.

In this dissertation, we propose a new block placement and migration policy for a hybrid STT-RAM-based LLC. LLC writes are categorized into three classes: core-write,

prefetch-write, and demand-write. We analyze the access pattern for each class of LLC writes and design a block placement policy that adapt to the access pattern of each class. A low cost access pattern predictor is proposed for guiding the block placement. Experimental results show our technique can improve performance and reduce LLC power consumption compared with both SRAM LLC and STT-RAM LLC with the same area configuration.

We also propose a dynamic cache management policy in the context of PCM- based main memory. The technique improves system performance and energy efficiency by reducing the writeback requests to PCM. It keeps highly reused dirty cache blocks in the LLC. A frequent write predictor is proposed to predict the frequent writeback cache blocks. The cache set is partitioned into frequent writeback and non- frequent writeback lists. It dynamically determines the optimal size of each list ac- cording to the miss penalty. Our evaluation shows the proposed techniques reduce the writeback requests which could result in improved performance as well as reduced energy consumption.

REFERENCES

[1] Manu Awasthi, David W. Nellans, Kshitij Sudan, Rajeev Balasubramonian, and Al Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 319–330, New York, NY, USA, 2010. ACM.

[2] John E. Baldwin and Christopher A. Haniff. The application of interferometry to optical astronomical imaging. *Philosophical Transactions of The Royal Society*, 360(1794):969–986, May 2002.

[3] Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 242–251, New York, NY, USA, 1994. ACM.

[4] Yu-Ting Chen, Jason Cong, Hui Huang, Bin Liu, Chunyue Liu, Miodrag Potkonjak, and Glenn Reinman. Dynamically reconfigurable hybrid cache: An energy-efficient last-level cache design. In *DATE'12*, pages 45–50, 2012.

[5] Yu-Ting Chen, Jason Cong, Hui Huang, Chunyue Liu, Raghu Prabhakar, and Glenn Reinman. Static and dynamic co-optimizations for blocks mapping in hybrid caches. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, ISLPED '12, pages 237–242, New York, NY, USA, 2012. ACM.

[6] Youngdon Choi, Ickhyun Song, Mu-Hui Park, Hoeju Chung, Sanghoan Chang, Beakhyoung Cho, Jinyoung Kim, Younghoon Oh, Duckmin Kwon, Jung Sunwoo, Junho Shin, Yoohwan Rho, Changsoo Lee, Min-Gu Kang, Jaeyun Lee, Yongjin

Kwon, Soehee Kim, Jaehwan Kim, Yong-Jun Lee, Qi Wang, Sooho Cha, Sujin Ahn, H. Horii, Jaewook Lee, Kisung Kim, Hansung Joo, Kwangjin Lee, Yeong-Taek Lee, Jeihwan Yoo, and G. Jeong. A 20nm 1.8v 8gb pram with 40mb/s program bandwidth. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 46–48, Feb 2012.

[7] An chow Lai. Dead-block prediction and dead-block correlating prefetchers. In *In Proceedings of the 28th International Symposium on Computer Architecture*, pages 144–154, 2001.

[8] Gilberto Contreras and Margaret Martonosi. Power prediction for intel xscale processors using performance monitoring unit events. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ISLPED '05, pages 221–226, New York, NY, USA, 2005. ACM.

[9] Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge. High-performance drams in workstation environments. *IEEE Trans. Comput.*, 50:1133–1153, November 2001.

[10] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 266–277, New York, NY, USA, 2001. ACM.

[11] Rajagopalan Desikan, Doug Burger, Stephen W. Keckler, Llorenc Cruz, Fernando Latorre, Antonio González, and Mateo Valero. Errata on "measuring experimental error in microprocessor simulation". *SIGARCH Comput. Archit. News*, 30(1):2–4, 2002.

[12] Xiangyu Dong, Cong Xu, Yuan Xie, and N.P. Jouppi. Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *Computer-Aided*

Design of Integrated Circuits and Systems, IEEE Transactions on, 31(7):994–1007, 2012.

[13] Marius Evers, Po-Yung Chang, and Yale N. Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA '96, pages 3–11, New York, NY, USA, 1996. ACM.

[14] Viacheslav V. Fedorov, Sheng Qiu, A. L. Narasimha Reddy, and Paul V. Gratz. Ari: Adaptive llc-memory traffic management. *ACM Trans. Archit. Code Optim.*, 10(4):46:1–46:19, December 2013.

[15] Domenico Ferrari. Improving locality by critical working sets. *Communications of the ACM*, 17(11):614–620, November 1974.

[16] Nikolas Gloy and Michael D. Smith. Procedure placement using Temporal-Ordering information. *ACM Transactions on Programming Languages and Systems*, 21(5):977–1027, September 1999.

[17] Satoru Hanzawa, N. Kitai, K. Osada, A. Kotabe, Y. Matsui, N. Matsuzaki, N. Takaura, M. Moniwa, and T. Kawahara. A 512kB Embedded Phase Change Memory with 416kB/s Write Throughput at 100μA Cell Write Current. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 474–616, Feb 2007.

[18] D. J. Hatfield and J. Gerald. Program restructuring for virtual memory. *IBM Syst. J.*, 10(3):168–192, September 1971.

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

[20] Chunling Hu, John McCabe, Daniel A. Jiménez, and Ulrich Kremer. The camino compiler infrastructure. *SIGARCH Comput. Archit. News Special Issue on the 2005 Workshop on Binary Instrumentation and Application*, 33(5):3–8, 2005.

[21] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 209–220, Washington, DC, USA, 2002. IEEE Computer Society.

[22] Ibrahim Hur and Calvin Lin. Adaptive history-based memory schedulers. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 343–354, Washington, DC, USA, 2004. IEEE Computer Society.

[23] Intel Corporation. Intel Pentium 4 processor optimization. Technical Report Order Number: 248966, Intel Corporation, 2001.

[24] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 39–50, Washington, DC, USA, 2008. IEEE Computer Society.

[25] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[26] Amin Jadidi, Mohammad Arjomand, and Hamid Sarbazi-Azad. High-endurance and performance-efficient design of hybrid cache architectures through adaptive line replacement. In *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, ISLPED '11, pages 79–84, Piscataway, NJ, USA, 2011. IEEE Press.

[27] Daniel A. Jiménez. Code placement for improving dynamic branch prediction accuracy. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 107–116, New York, NY, USA, 2005. ACM.

[28] Daniel A. Jiménez, Stephen W. Keckler, and Calvin Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 67–76, New York, NY, USA, 2000. ACM.

[29] Adwait Jog, Asit K. Mishra, Cong Xu, Yuan Xie, Vijaykrishnan Narayanan, Ravishankar Iyer, and Chita R. Das. Cache revive: Architecting volatile stt-ram caches for enhanced performance in cmps. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 243–252, New York, NY, USA, 2012. ACM.

[30] Yongsoo Joo, Dimin Niu, Xiangyu Dong, Guangyu Sun, Naehyuck Chang, and Yuan Xie. Energy- and endurance-aware design of phase change memory caches. In *DATE*, pages 136–141, 2010.

[31] P. J. Joseph, Kapil Vaswani, and Matthew J. Thazhuthaveetil. A predictive performance model for superscalar processors. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 161–170, Washington, DC, USA, 2006. IEEE Computer Society.

[32] Samira M. Khan, Yingying Tian, and Daniel A. Jiménez. Sampling dead block prediction for last-level caches. In *MICRO*, pages 175–186, December 2010.

[33] Samira M. Khan, Zhe Wang, and Daniel A. Jimenez. Decoupled dynamic cache segmentation. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[34] 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.

[35] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Trans. Comput.*, 57:433–447, April 2008.

[36] C.H. Kim, Jae-Joon Kim, S. Mukhopadhyay, and K. Roy. A forward body-biased low-leakage sram cache: device, circuit and architecture considerations. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 13(3):349–357, 2005.

[37] Yoongu Kim, Dongsu Han, O. Mutlu, and M. Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, jan. 2010.

[38] Dan Knights, Todd Mytkowicz, Peter F. Sweeney, Michael C. Mozer, and Amer Diwan. Blind optimization for exploiting hardware features. In *CC '09: Proceedings of the 18th International Conference on Compiler Construction*, pages 251–265, Berlin, Heidelberg, 2009. Springer-Verlag.

[39] An-Chow Lai and Babak Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 139–148, New York, NY, USA, 2000. ACM.

[40] Benjamin C. Lee and David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *SIGPLAN Not.*, 41(11):185–194, October 2006.

[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] 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.

[43] Wei-Fen Lin, Steven K. Reinhardt, and Doug Burger. Designing a modern memory hierarchy with hardware prefetching. *IEEE Trans. Comput.*, 50:1202–1218, November 2001.

[44] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 222–233, Washington, DC, USA, 2008. IEEE Computer Society.

[45] 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 *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[46] Mengjie Mao, Hai (Helen) Li, Alex K. Jones, and Yiran Chen. Coordinating prefetching and stt-ram based last-level cache management for multicore systems. In *Proceedings of the 23rd ACM international conference on Great lakes symposium on VLSI*, GLSVLSI '13, pages 55–60, New York, NY, USA, 2013. ACM.

[47] Scott McFarling. Program optimization for instruction caches. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191. ACM, 1989.

[48] William Mendenhall, Dennis D. Wackerly, and Richrd L. Sheaffer. *Mathematical Statistics withh Applications, Fourth Edition*. PWS Publishers, Boston, MA, 1986.

[49] Pierre Michaud, André Seznec, and Richard Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 292–303, June 1997.

[50] Shirley Moore, David Cronk, Felix Wolf, Avi Purkayastha, Patricia Teller, Robert Araiza, Maria Gabriela Aguilera, and Jamie Nava. Performance profiling and analysis of dod applications using papi and tau. In *DOD_UGC '05: Proceedings of the 2005 Users Group Conference on 2005 Users Group Conference*, page 394, Washington, DC, USA, 2005. IEEE Computer Society.

[51] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 3–14, Washington, DC, USA, 2007. IEEE Computer Society.

[52] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 146–160, Washington, DC, USA, 2007. IEEE Computer Society.

[53] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 63–74, Washington, DC, USA, 2008. IEEE Computer Society.

[54] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 265–276, New York, NY, USA, 2009. ACM.

[55] Veynu Narasiman, Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. Dram-aware last level cache writeback: Reducing write-caused interference in memory system. In *HPS Technical Report*, TR-HPS-2010-002.

[56] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. Fair queuing memory systems. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 208–222, Washington, DC, USA, 2006. IEEE Computer Society.

[57] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. MARSSx86: A full system simulator for x86 CPUs. In *Proceedings of the 2011 Design Automation Conference*, June 2011.

[58] F. Pellizzer, A. Pirovano, F. Ottogalli, M. Magistretti, M. Scaravaggi, et al. Novel μTrench Phase-Change Memory Cell for Embedded and Stand-Alone Non-Volatile Memory Applications. In *VLSI Technology, 2004. Digest of Technical Papers. 2004 Symposium on*, pages 18–19, 2004.

[59] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[60] Moinuddin K. Qureshi, Michele M. Franceschini, Ashish Jagmohan, and Luis A. Lastras. Preset: improving performance of phase change memories by exploiting asymmetry in write times. In *Proceedings of the 39th International Symposium*

on Computer Architecture, ISCA '12, pages 380–391, Piscataway, NJ, USA, 2012. IEEE Press.

[61] Moinuddin K. Qureshi, Michele M. Franceschini, and Luis A. Lastras-montao. Improving read performance of phase change memories via write cancellation and write pausing. In *International Symposium on High Performance Computer Architecture*, HPCA '10, pages 1–11, 2010.

[62] 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.

[63] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A case for mlp-aware cache replacement. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 167–178, Washington, DC, USA, 2006. IEEE Computer Society.

[64] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *In International Symposium on Computer Architecture (ISCA*, 2009.

[65] 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.

[66] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. Phase-change random access memory: A scalable technology. *IBM J. Res. Dev.*, 52(4):465–479, July 2008.

[67] Scott Rixner. Memory controller optimizations for web servers. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 355–366, Washington, DC, USA, 2004. IEEE Computer Society.

[68] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 128–138, New York, NY, USA, 2000. ACM.

[69] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 10(1):16–19, Jan 2011.

[70] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996.

[71] Shai Rubin, Rastislav Bodík, and Trishul Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 140–153, New York, NY, USA, 2002. ACM.

[72] André Seznec. A 256 kbits l-tage branch predictor. *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, 9, May 2007.

[73] Jun Shao and Brian T. Davis. A burst scheduling access reordering mechanism. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 285–294, Washington, DC, USA, 2007. IEEE Computer Society.

[74] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[75] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 135–148, May 1981.

[76] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 63–74, Washington, DC, USA, 2007. IEEE Computer Society.

[77] 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.

[78] Kshitij Sudan, Niladrish Chatterjee, David Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. Micro-pages: increasing dram efficiency with locality-aware data placement. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 219–230, New York, NY, USA, 2010. ACM.

[79] Guangyu Sun, Xiangyu Dong, Yuan Xie, Jian Li, and Yiran Chen. A novel architecture of the 3d stacked mram l2 cache for cmps. In *HPCA*, pages 239–249, 2009.

[80] Shun-Ming Syu, Yu-Hui Shao, and Ing-Chao Lin. High-endurance hybrid cache design in cmp architecture with cache partitioning and access-aware policy. In *Pro-*

ceedings of the 23rd ACM international conference on Great lakes symposium on VLSI, GLSVLSI '13, pages 19–24, New York, NY, USA, 2013. ACM.

[81] Jue Wang, Xiangyu Dong, and Yuan Xie. Oap: An obstruction-aware cache management policy for stt-ram last-level caches. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 847–852, 2013.

[82] Xiaobin Wang, Yiran Chen, Hai Li, D. Dimitrov, and H. Liu. Spin torque random access memory down to 22 nm technology. *IEEE Transactions on Magnetics*, 44(11):2479–2482, 2008.

[83] Zhe Wang and Daniel A. Jiménez. Program interferometry. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 172–175, 2011.

[84] Zhe Wang, Daniel A. Jiménez, Cong Xu, Guangyu Sun, and Yuan Xie. Adaptive placement and migration policy for an stt-ram-based hybrid cache. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA-20)*, Orlando, FL, USA, February 2014. IEEE Computer Society.

[85] Zhe Wang, Samira M. Khan, and Daniel A. Jiménez. Improving writeback efficiency with decoupled last-write prediction. In *Proceedings of the 39th International Symposium on Computer Architecture*, ISCA '12, pages 309–320, Piscataway, NJ, USA, 2012. IEEE Press.

[86] Zhe Wang, Samira M. Khan, and Daniel A. Jiménez. Rank idle time prediction driven last-level cache writeback. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '12, pages 21–29, New York, NY, USA, 2012. ACM.

[87] Zhe Wang, Shuchang Shan, Ting Cao, Junli Gu, Yi Xu, Shuai Mu, Yuan Xie, and Daniel A. Jiménez. Wade: Writeback-aware dynamic cache management for nvm-

based main memory system. *ACM Trans. Archit. Code Optim.*, 10(4):51:1–51:21, December 2013.

[88] Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. Pacman: prefetch-aware cache management for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 442–453, New York, NY, USA, 2011. ACM.

[89] Yuan Xie. Modeling, architecture, and applications for emerging memory technologies. *IEEE Computer Design and Test*, 28:41–51, January 2011.

[90] T.-Y. Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th ACM/IEEE International Symposium on Microarchitecture*, pages 51–61, November 1991.

[91] HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael Harding, and Onur Mutlu. Row buffer locality aware caching policies for hybrid memories. In *International Conference on Computer Design*, ICCD '12, 2012.

[92] Cliff Young, David S. Johnson, David R. Karger, and Michael D. Smith. Near-optimal intraprocedural branch alignment. In *Proceedings of the SIGPLAN'97 Conference on Program Language Design and Implementation*, June 1997.

[93] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 32–41, New York, NY, USA, 2000. ACM.

[94] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th*

annual international symposium on Computer architecture, ISCA '09, pages 14–23, New York, NY, USA, 2009. ACM.