The Dissertation Committee for Jee Ho Ryoo
certifies that this is the approved version of the following dissertation:

Improving System Performance by Utilizing Heterogeneous Memories

Committee:

Lizy K. John, Supervisor

Earl E. Swartzlander, Jr.

Nur A. Touba

Mattan Erez

Mitesh R. Meswani

Improving System Performance by Utilizing Heterogeneous Memories

by

Jee Ho Ryoo, B.S., M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2017

Dedicated to my family.

Acknowledgments

I would like to thank my advisor, Professor Lizy K. John, for being a great mentor throughout the Ph.D. program. She has always been very supportive and provided me invaluable guidance, and research funding throughout my years here. She has motivated me to continue hard work and was always available to provide research feedback. I would also like to thank Earl Swartzlander, Nur Touba, Mattan Erez and Mitesh Meswani for serving in my Ph.D. committee. Their constructive criticism helped improve the quality of this dissertation.

I also would like to thank my undergraduate advisors at Cornell. Professor David H. Albonesi and Professor Christopher Batten have always been supportive during my undergraduate as well as my graduate studies. Throughout my Ph.D., I have met a number of great people. First, I need to thank all of the members of the LCA research group for providing a creative and helpful environment. Through multiple internships, I also met many mentors and researchers at AMD Research, Oracle, and Samsung, whom I shared my research interests with in great extent.

I would also like to thank the organizations that supported me financially during my Ph.D. studies. National Science Foundation (NSF), Semiconductor Research Corporation (SRC), Oracle, and the University of Texas at

Austin provided grants for my research.

I am very grateful to my family for their emotional support throughout my lengthy Ph.D. program. They have always been a great source of encouragement and motivating factors in my pursuit towards this degree.

Improving System Performance by Utilizing Heterogeneous Memories

by

Jee Ho Ryoo, Ph.D.
The University of Texas at Austin, 2017

Supervisor: Lizy K. John

As device technologies scale in the nanometer era, the current off-chip DRAM technologies are very close to reaching the physical limits where they cannot scale anymore. In order to continue the memory scaling, vendors are beginning to use new emerging memory technologies such as die-stacked DRAM. Although each type of memory technology has advantages and disadvantages, none has characteristics that are identical to conventional DRAM. For example, High Bandwidth Memory (HBM) has better bandwidth but lower capacity than DRAM whereas non-volatile memories offer more capacity, but are much slower than DRAM. With the emergence of such disparate memory technologies, future memory systems are certain to be heterogeneous where the main memory is composed of two or more types of memory.

Recent and current trends show that the number of cores in a processor has been rising constantly. On the other hand, the off-chip DRAM bandwidth has not been scaling at the same rate as the bandwidth is limited mainly by the pin count. This trend has resulted in lower bandwidth per core in today's system where the bandwidth available to each core is lower than the system in the past. This has effectively created the "Bandwidth Wall" where the bandwidth per core does not scale anymore. Consequently, memory vendors are introducing HBM in order to enable the bandwidth scaling. The adoption of such high bandwidth memory and other emerging memory technologies has provided rich opportunities to explore how such heterogeneous main memory systems can be used effectively.

In this dissertation, different ways to effectively use such heterogeneous memory systems, especially those containing off-chip DRAM and die-stacked DRAM such as HBM, are presented. First, hardware as well as software driven data management schemes are presented where either hardware or software explicitly migrates data between the two different memories. The hardware managed scheme does not use a fixed granularity migration scheme, but rather migrates variable amount of data between two different memories depending on the memory access characteristics. This approach achieves low off-chip memory bandwidth usage while maintaining a high hit rate in the more desirable memory. Similarly, a software driven scheme varies the migration granularity without any additional hardware support by monitoring the spatial locality characteristics of the running applications. In both solutions, the goal is to migrate just the right amount of data into capacity constrained memory to achieve the low off-chip memory bandwidth usage while still keeping the high hit rate.

While the capacity of die-stacked DRAM is not sufficient to meet the demands of a server-class system, it is still non-trivial in size ranging from 8 to 16 GBs in typical configurations, so a fraction of the storage can be used for non-conventional uses such as storing address translations. With increasing deployment of virtual machines for cloud services and server applications, one major contributor of performance overheads in virtualized environments is memory address translation. An application executing on a guest OS generates guest virtual addresses that need to be translated to host physical addresses. In x86 architectures, both the guest and host page tables employ a 4-level radix-tree table organization. Translating a virtual address to physical address takes 4 memory references in a bare metal case using a radix-4 table, and in the virtualized case, it becomes a full 2D translation with up to 24 memory accesses. A method to use a fraction of die-stacked DRAM as a very large TLB is presented in this dissertation, so that almost all page table walks are eliminated. This leads to a substantial performance improvement in virtualized systems where the address translation takes a considerable fraction of execution time.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

DRAM technology is facing fundamental physical design challenges with shrinkage of the cell size [1]. Thus, it is becoming harder to continue growing the capacity of conventional DRAM-only homogeneous memory systems. Fortunately, several non-volatile memory (NVM) technologies like Phase Change Memory (PCM) [2, 3], memristor [4], and MRAM [5] are quickly emerging as a credible supplement to DRAM since they scale well with technology shrinkage. In an orthogonal but concurrent trend, the advancement in 3D integration techniques has allowed high-bandwidth stacked memories [6, 7] to be integrated on commercial processors [8–10]. Each of these disparate memory technologies has distinct access characteristics – each has different access latencies, access energy and/or bandwidth. With the emergence of such disparate memory technologies, future memory systems are certain to be heterogeneous.

1.1 Problem Description

Although the capacity of die-stacked DRAM is much greater than conventional SRAM caches, many emerging applications often have a working

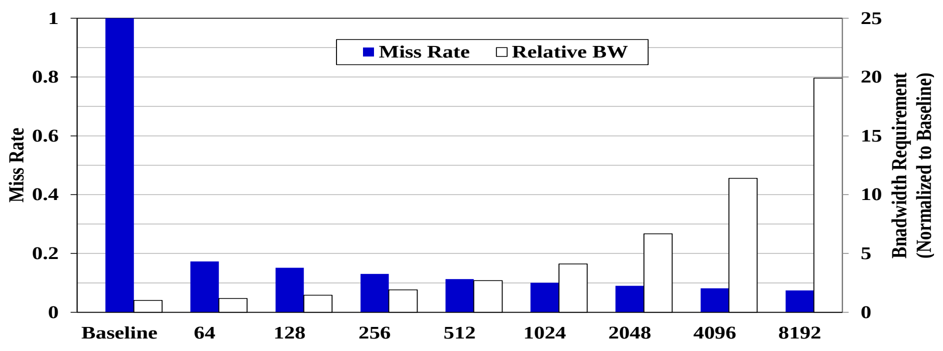


Figure 1.1: Effect of Large Block Size on Hit Rate/Bandwidth

set size of several gigabytes [11–13]. Therefore, only a subset of pages can be present in die-stacked DRAM at any one time. Using this memory as a large Last Level Cache (LLC) can hold significantly more cacheline blocks than on-chip SRAM caches, yet the metadata area overheads such as tags become extremely high. Managing data at different cacheline sizes can reduce the metadata storage overheads, yet the bandwidth consumption becomes high. Keeping tags in SRAM is infeasible as 1GB of the die-stacked DRAM requires approximately 20MB SRAM storage just for tags.

Using a large cacheline size can bring down the tag area overheads to a manageable size as in [14]. However, Figure 1.1 shows the tradeoff data gathered from simulation involving this large cacheline approach. Due to spatial locality, the number of misses decreases as the cacheline size increases. However, at the same time, the size of data brought in on every eviction increases. This relationship is shown by the relative bandwidth used by each cacheline granularity. With each incremented cacheline size, the bandwidth used doubles as a cache miss brings in twice as much data. Consequently, very large page sizes not only hurt performance due to high overheads associated with moving large amounts of data, but also place more pressure on the already saturated off-chip memory bus, exacerbating the problem that this new memory technology is trying to solve. Sectored caches [15] can solve the bandwidth problem by only fetching demanded sectors, or subblocks.

The Operating System (OS) can be involved to explicitly manage such capacity via page placement [16–19]. The OS monitors page usage and swaps hot pages from off-chip DRAM to die-stacked DRAM at some fixed time intervals or epochs. In an epoch based scheme, the OS explicitly manages the die-stacked DRAM capacity as a special region of memory. The OS moves frequently accessed pages, or hot pages, into die-stacked DRAM and the Page Table Entry (PTE) is updated accordingly. Upon an access, the physical address provided by Translation Lookaside Buffer (TLB) is directly used to access data in die-stacked DRAM. Unlike hardware managed schemes, it does not have additional hardware structures like tag checking logic or modified addressing schemes. The major drawbacks include the high software related overheads. To amortize the high overheads, the OS management is done at coarse granularity (hundreds of milliseconds). As a result, software schemes are typically slow to react to changes in the working set.

To address these issues, recent proposals [20, 21] expose the die-stacked DRAM capacity while transparently migrating data between two memories without involving the OS. The migration is done using hardware remapping tables, so some off-chip memory data are remapped to die-stacked DRAM. When die-stacked DRAM is used as a part of memory rather as a cache, there are several sophisticated design challenges. First, when swapping is done without OS intervention, a remap table is necessary to allow dynamic migration of data between the two memories. However, keeping an SRAM based remap table for every small block incurs high remap table storage overheads [20]. As die-stacked DRAM capacity scales up, the remap table storage overheads become a more important issue. Therefore, efficiently storing such large metadata is challenging and important. Increasing the remapping granularity to a large block (e.g., 2KB) can mitigate this issue, yet on each remapping migration, the bandwidth usage becomes exorbitant as many unused subblocks also have to be migrated. Furthermore, recent work [22] has shown that as die-stacked DRAM capacity scales, bandwidth is problematic even at the large remap granularity.

1.2 Overview of Proposed Research

A solution to fully utilize the die-stacked DRAM capacity by intelligently placing hot data in die-stacked DRAM is presented in this dissertation. The solution adopts subblock level data movement that reduces the bandwidth usage compared to a large cacheline scheme and prevents fetching data that may not get utilized much. In effect, the proposed scheme gets the benefits of both small and large block hardware managed schemes. Furthermore, the solution interleaves hot subblocks between on-chip and off-chip DRAM, so that subblocks originally from on-chip and off-chip DRAM can coexist together in die-stacked DRAM. The solution is robust to common problems associated with recent proposals since it incorporates a locking feature that prevents hot pages from being involved in undesirable data migration operations. Also, associativity protects those pages that are not locked and are actively participating in data migrations from being frequently swapped out. Finally, the memory bandwidth usage is optimized by utilizing both on-chip and off-chip memory bandwidth. When die-stacked DRAM is used as a cache, the system would achieve the maximum performance when all requests are serviced from die-stacked DRAM. However, experiments in this dissertation have found that splitting the bandwidth, or bypassing in other words, achieves higher overall performance than getting everything from the slightly faster memory when it is used as a part of memory.

In software-oriented perspectives, the software driven page migration is advantageous as it does not require hardware modifications and can be implemented on existing systems with heterogeneous memories. The OS directed migration relies on using a fixed OS page size (e.g., 4KB default on x86-64). If a larger migration granularity is used, then the amount of overhead increases proportionally. However, it is advantageous for high spatial locality pages to be migrated together as a large chunk, so that the overheads can be amortized. If the migration granularity is increased based on the spatial locality of an application, then some software overhead, like TLB shootdowns, can be reduced. Since different applications show different spatial locality behavior, a scheme that can dynamically adjust the migration granularity can improve the overall performance without any extra hardware or any source code modifications. Therefore, this dissertation proposes two dynamic granularity-aware schemes implemented in Linux kernel and evaluate their performance on a real system.

Orthogonally, the entire die-stacked DRAM does not have to be used for data storage purposes. Since it is much larger than conventional on-chip SRAM structures, a fraction of it can be allocated for other uses. In today's computing, a structure that suffers from a lack of capacity is the translation lookaside buffer (TLB). The effectiveness of TLBs directly affects the address translation performance. As virtualized platforms become increasingly popular, one of the largest contributors of performance overhead in virtualized environments is memory virtualization. An application executing on a guest OS generates guest virtual addresses that need to be translated to host physical addresses. Since physical memory is under the exclusive control of a hypervisor, every guest *physical* address needs to be translated to host physical before the guest application-issued memory access can complete. This requires navigating through two sets of page tables: a guest page table where the guest OS performs the virtual-to-physical address translation on the guest side, and a host page table where the hypervisor performs virtual-to-physical address translation on the host side. In x86 architectures, both the guest and host page tables employ a 4-level radix-tree table organization. Translating a virtual address to physical address takes 4 memory references in a bare metal case using a radix-4 table, and in the virtualized case, it becomes a full $2D$

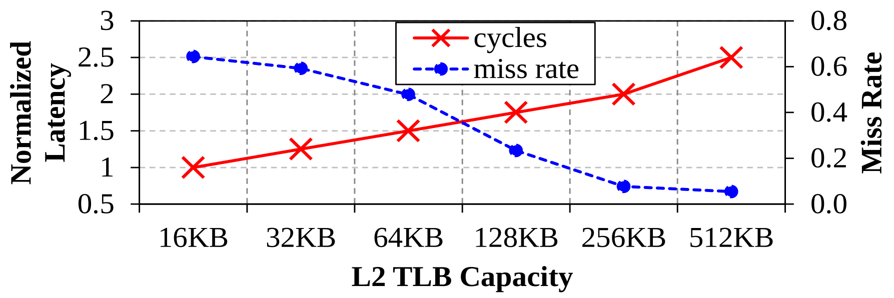


Figure 1.2: L2 TLB Scaling Trend

translation with up to 24 memory accesses. With the increased number of cores and big data sets, the conventional two-level SRAM TLBs cannot hold translations of all the pages in the working set. Increasing L2 TLB sizes to sufficiently reduce TLB misses is not feasible because larger SRAM TLBs incur higher access latencies. Figure 1.2 shows the access latency trend with larger L2 TLB capacities. The access latency is normalized to that of 16KB SRAM, and as seen, naively increasing the SRAM capacity does not scale.

In light of the above discussion, it would be desirable to have a TLB with a large reach at tolerable latency, so that a large majority of translations can be handled by TLBs rather than by Page Table Walkers (PTW). A novel solution in this direction is presented in this dissertation, a very large level-3 TLB that is part of memory. While TLBs are dedicated structures and not ordinarily addressable, the proposed solution is mapped into the memory address space, and is stored in die-stacked DRAM. It is large enough to house translation entries for significantly huge working sets. By making it part of memory, it becomes possible to automatically take advantage of the growing

L2 and L3 data cache capacities. While data caches already cache page table entries, multiple page table entries will be required per each translation, whereas a single TLB entry will be sufficient to accomplish the virtualized translation. Hence, caching TLB entries is more effective and beneficial than caching page table entries.

1.3 Thesis Statement

Using high bandwidth on-package memories as a part of memory rather than caches is beneficial from a capacity perspective, and using interleaved data mapping and variable migration granularity rather than a fixed migration granularity suits the needs of applications better. In addition, a fraction of the high bandwidth on-package memory capacity can be used for address translations in order to reduce translation overheads, especially in the context of virtualization.

1.4 Contributions

This dissertation makes the following contributions:

- The high bandwidth costs typically associated with large cacheline data placement and migration in prior proposals can be eliminated by using subblocking. **Subblocking interleaves** subblocks from off-chip DRAM to reside in die-stacked DRAM, and thus, increase the usefulness of the die-stacked DRAM capacity.

- Hot pages benefit from being placed in die-stacked DRAM as those pages benefit from high bandwidth. A mechanism is presented in this dissertation to identify hot pages and to **lock** them in die-stacked DRAM, so that conflicting data does not swap hot pages out to off-chip DRAM. Unlike epoch based approaches, the locking does not need to occur at coarse grain time intervals, and thus it can react more quickly to the changes in the hot working set.

- When die-stacked DRAM is a part of the address space, 100% hit rate does not give the maximum performance. It is beneficial to service a fraction of requests from the off-chip DRAM to utilize the overall available system bandwidth. If the bandwidth available from the two memory levels are N:1, it is beneficial to service 1/(N+1) of the accesses from off-chip DRAM. Therefore, the proposed mechanism achieves high performance improvement with a slightly lower number of requests serviced from die-stacked DRAM. This leads to **balanced** bandwidth utilization, which in turn provides performance improvement.

- By analyzing a wide range of applications on real hardware, this dissertation shows that different applications perform best under different page migration granularities and that dynamic granularity adjustment provides a considerable amount of performance improvement in heterogeneous memory systems.

- Two simple granularity-aware schemes to dynamically select the preferred page migration granularity for a given application at runtime without any hardware or application modifications are presented in this dissertation.

- The feasibility of using die-stacked DRAM for address translation is analyzed in this dissertation. The proposed TLB uses a very large capacity (albeit slow) that can house nearly all required translations, eliminating a large number of expensive page table walks.
- The challenges encountered while implementing a TLB in die-stacked DRAM are resolved in this dissertation. A low overhead TLB location predictor and other enhancements make a shared L3 TLB a feasible option.

1.5 Dissertation Organization

The dissertation is organized as follows: Chapter 2 presents background and related work. It presents the details of die-stacked DRAM as well as address translation in virtual platforms. Prior work related to die-stacked DRAM as well as various address translation optimizations are also presented. Chapter 3 presents the evaluation framework used in this dissertation and explains the set of benchmarks that were used. Chapter 4 evalutes various dimensions of data migration such as the hit rate, bandwidth usage and migration granularity in order to extract the most performance from die-stacked DRAM. It also presents performance challenges associated with address translation in virtualized systems. Chapter 5 describes the novel data migration scheme that achieves high hit rate while maintaining low bandwidth usage. Additionally, it explains several die-stacked DRAM specific features that can be exploited to improve performance. Chapter 6 presents the design and evaluation of a granularity-aware data migration scheme that adjusts the amount of data migration by detecting the spatial locality of an application at runtime. Chapter 7 presents another way to use die-stacked DRAM other than data storage. More specifically, it uses die-stacked DRAM as a large TLB to address TLB reach issues faced by today's virtualized computing environments. Finally, Chapter 8 presents the conclusions and the areas of future work.

Chapter 2

Background Terminology and Related Work

Efficient uses of emerging memory technologies have been explored extensively by both industry and academia. However, there has not been any concrete consensus on the best use of the technologies. This dissertation mainly focuses on studying various mechanisms to use an emerging memory technology, namely die-stacked DRAM, rather than attempting to propose a single scheme. This chapter first provides a brief introduction to the terminology used in heterogeneous memory system and virtual address translation designs, followed by the related work on these fields. Detailed description of related work for the specific schemes is discussed in depth in the corresponding chapters.

2.1 Die-Stacked DRAM

Current DRAM technology is reaching its physical limit, and thus, it does not scale in performance or capacity. At the same time, the number of cores is constantly increasing. Therefore, the bandwidth demand from the processor side is on the rise as a higher number of cores place more pressure on the memory bus. This trend has created the "Bandwidth Wall" [23]. To address

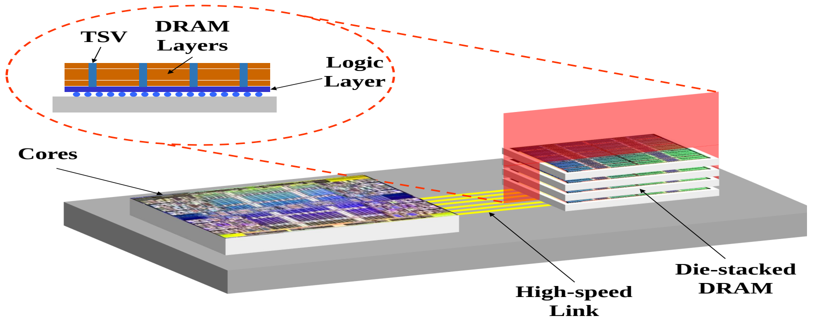


Figure 2.1: Die-Stacked DRAM Structure

the bandwidth wall, memory vendors have introduced promising technology, namely die-stacked DRAM.

Die-stacked DRAM is a new emerging memory technology that is composed of DRAM layers that are stacked on top of each other vertically. Figure 2.1 shows the structure and layout of die-stacked DRAM. The bottom layer is a logic layer that can be configured to the vendor's preferences. It is placed on the same silicon interposer as the cores, and is connected to them by high-speed links. This link operates at a much higher frequency due to its lower wire capacitance. The physical distance from the cores to die-stacked structure is much shorter than that to the off-chip DRAM, so it can provide slightly lower latency than today's DRAM [7, 24]. The die-stacked structure itself also provides significant benefits. DRAM layers are stacked on top of each other and connected by low resistance Through Silicon Vias (TSVs). Unlike off-chip DRAM where one of the major bandwidth bottlenecks is the number of pins, this structure does not need any pins, so the bandwidth is not limited by the pin count. The fact that all traffic to this storage device never travels

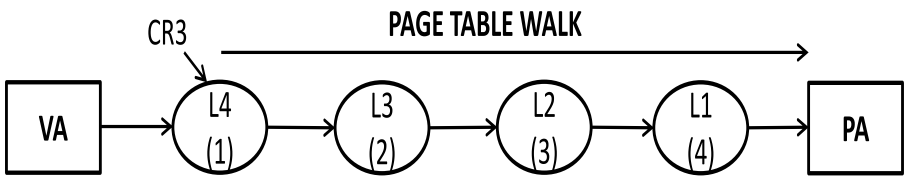


Figure 2.2: x86 1D Page Walk In Native Environment

off-chip makes it an ideal complement for DRAM.

In this dissertation, the die-stacked DRAM is referred as Near Memory (NM) and off-chip DRAM as Far Memory (FM) since NM is physically located closer to the cores with high bandwidth. Also, we call 64B worth of contiguous address space as a small block or subblock and 2KB worth of contiguous address space as a large block.

2.2 Address Translation

The capacity of die-stacked DRAM ranges from hundreds of megabytes to several gigabytes. Thus, a subset of this large capacity can be used for other purposes such as storing address translations. Even though it is a small fraction from die-stacked DRAM's perspectives, this is much larger than conventional on-chip TLBs, and thus, provides much larger TLB reach for applications whose memory footprint is very large.

One of the largest performance overhead in virtualized environments is memory virtualization. In non-virtualized systems, an address translation in x86 architectures requires up to 4 memory accesses as shown in Figure 2.2 since the architecture employs a four-level page table walk. Many of these

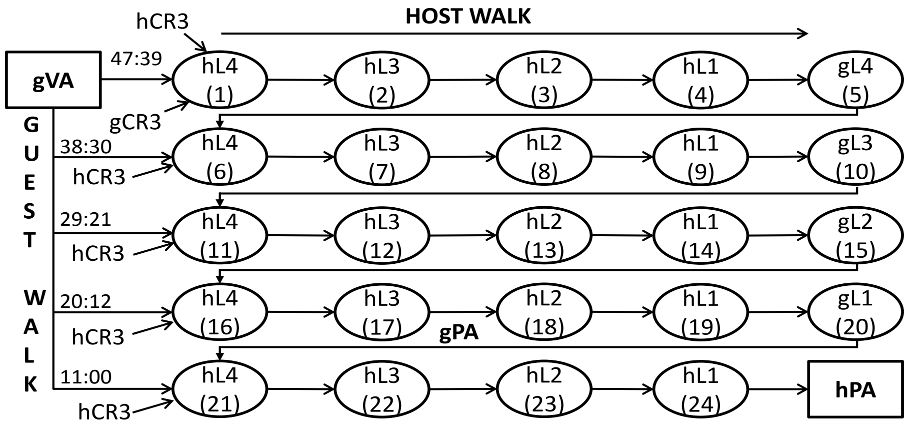


Figure 2.3: x86 2D Page Walk In Virtualized Environment

intermediate page table walk entries are stored in data caches, so they do not require accesses to long latency off-chip DRAM. The case is little different in virtualized systems. An application executing on a guest OS generates guest virtual addresses (gVA) that need to be translated to host physical addresses (hPA). Since physical memory is under the exclusive control of a hypervisor, every guest physical address (gPA) needs to be translated to host physical before the guest application issued memory access can complete. This requires navigating through two sets of page tables: a guest page table that the guest OS implements ($gVA \rightarrow gPA$), and a host page table that the hypervisor implements ($gPA \rightarrow hPA$). In x86 architectures, both the guest and host page tables employ a 4-level radix-tree table organization. Translating a virtual address to physical address takes 4 memory references in a bare metal case using a radix-4 table, and in the virtualized case, it becomes a full $2D$ translation with up to 24 memory accesses as depicted in Figure 2.3.

In order to bridge the performance gap associated with address translations, recent processors have added architecture supports in the form of nested page tables [25] and extended page tables [26] that cache guest-to-host translations. Processor vendors have also added dedicated MMU/page walk caches [27, 28] to cache the contents of guest and host page tables. Additional techniques to reduce the overhead of page walks include caching of page table entries in data caches, agile paging [29], TLB prefetching [30], shared L2 TLBs [31], transparent huge pages (THP) [32], translation storage buffers (TSB) [33], speculative TLB fetching [34] and splintering [35]. These page walk enhancements have significantly reduced translation costs; however, the translation overhead continues to be a source of inefficiency in virtualized environments.

2.3 Related Work

2.3.1 Die-Stacked DRAM Data Management

Emerging memory technologies have provided opportunities for creating interesting memory system designs. Much of the work has been focused on efficiently storing metadata overheads for scalability of this multi-gigabyte memory technology. Since memory bandwidth is often the bottleneck, some work [14, 22, 36–45] has focused on reducing the FM bandwidth usage while managing a high hit rate to NM. These proposals manage NM as hardware caches, and thus, do not take advantage of added NM capacity. With growing NM capacity, the capacity expected to be a considerable amount of the total memory capacity in the near future, and thus, PoM [21] and CAMEO [20] are two state-of-the-art schemes that have focused on using this capacity as a part of OS visible space to take advantage of added capacity. However, their schemes are based on conventional caching techniques, and thus, they are susceptible to problems that exist in SRAM caches. Tagging using TLBs has been proposed [46], yet their focus was different as the work reduces the address translsation energy.

Previous software managed proposals focused on detecting what to migrate to NM. Meswani et al. [18] introduced per-page access counters to migrate pages whose counts are higher than a threshold at regular time intervals. The next-touch algorithm proposed by Goglin et al. [47] and the one proposed by Oskin et al. [48] use the demand request to trigger a migration. Non-Uniform Memory Architecture (NUMA) proposals [49–51] also attempt to solve data management challenges in memory systems with different access latencies. However, these proposals use placement strategies that are tailored towards placing hot data that benefits local nodes in a NUMA system. In the context of the memory system in this dissertation, the hot data placement must benefit all nodes since all nodes are considered local in heterogeneous memory systems. Other prior work [52–54] explored application/user feedback to guide page migrations. Lin et al. [52] proposed asynchronous migration triggered by user request. Similarly, Meswani et al. [53] explicitly manage die-stacked DRAM under application's direction, and Cantalupo et al. [54] proposed a explicit user level heap manager. In contrast to these proposals, the proposed scheme in this dissertation does not require application modifications. Kwon et al. [55] proposes an efficient transparent huge page support to better coalesce 4KB OS pages into a huge page, but they focus on finding as many contiguous pages as possible.

Prefetching techniques [56–58] have been explored extensively for masking the long memory access latencies by proactively fetching data into SRAM caches. Such techniques are done to move data from one level of memory to another whereas this dissertation treats both FM and NM as a part of the address space of the same memory hierarchy.

2.3.2 Address Translation Optimization

Caching and speculation techniques have been proposed to improve the two dimensional address translation overheads in virtualized platforms [25, 27, 29, 34, 35, 59]. Caching schemes such as page walk cache [27, 29, 60] attempt to bypass the intermediate level walks. Some processors such as Power8 adopts an inverted page table to reduce the number of page walks [61]. Also, recent work [62] showed that hashing PTEs can reduce the number of page walks. However, these approaches do not solve the fundamental small capacity problem of TLBs. Speculation schemes [34, 35] let the processor execution continue with speculated page table entries and invalidate speculated instructions upon detecting the misspeculation. These schemes are motivated by the fact that conventional TLBs are likely to cause more page table walks [25] for emerging big data workloads with large memory footprints. Therefore, they focus on reducing/hiding the overheads of page table walks. This dissertation focuses on a more fundamental solution. Using very large TLBs that can withstand increased address translation pressure from virtualization by virtue of offering a translation storage with high capacity and high bandwidth significantly reduces the number of walks.

Some other schemes [29, 63, 64] attempt to reduce the levels of page table walks in either native or virtualized systems. Moreover, TLB prefetching techniques [30, 65, 66] improve the TLB hit rate by fetching entries ahead of time. However, the fundamental problem of current TLB's insufficient capacities is still not addressed. Thus, by increasing the capacity significantly, the inherent structural bottleneck in today's system is solved in this dissertation. However, increasing the TLB capacity is orthogonal to aforementioned schemes, as the existing or proposed page walk structures are not altered by our proposal. Thus, the large TLB proposal can easily be added to these schemes.

The Linux Transparent Huge Page [32] along with various schemes [59, 63, 67–70] try to increase the fraction of large pages by either using hardware or by the OS to reduce the number of TLB misses. This dissertation merely uses such features as is because the memory traces contain the page size information. Yet, improving them can even further increase the TLB hit rate in this dissertation as a single 2MB entry incorporates 512 4KB entries, thereby effectively further increasing the reach of TLBs.

Chapter 3

Methodology

This dissertation uses a combination of a system simulator, Sniper [71], along with a detailed memory simulator, Ramulator [72], to evaluate the proposed schemes that efficiently use die-stacked DRAM. Even though each proposed scheme in this dissertation modifies the components of the existing simulator, and various system configurations, the overall experimental methodology and setup remain the same. For the majority of the dissertation, SPEC CPU2006 [73], PARSEC [74] and graph500 [75] benchmark suite was used as a representative, general purpose benchmark suite. In addition, the real system experiments used in this dissertation uses the modified Linux kernel to emulate the heterogeneous memory system where the memory latency is modified to model heterogeneous memories with different latencies. The evaluation on a real system uses multithreaded application suites to drive the multiple cores. Finally, the TLB related proposals are done by using a combination of a trace driven and a custom simulator to evaluate performance. A heavily modified Pin instrumentation tool [76] is used to extract necessary information such as the page size in order to drive the custom TLB simulator, which eventually computes the performance after simulating various steps in the memory access or address translation. The remainder of this chapter presents an introduction to each tool and a description of workload suites that are used to evaluate the proposed schemes.

3.1 Simulation Details

3.1.1 Memory Timing Simulation

In order to evaluate the memory system performance, a detailed memory timing simulator is adopted in this dissertation. Although there are many detailed memory simulators, they all slow down the processor simulation. A memory simulator that is reasonably fast and accurate is needed. Ramulator is a detailed timing simulator that decouples the work of querying/updating the state-machines that are used in many other memory simulators. Its speed is approximately 3X faster than a popular memory simulator, DRAMSim [77]. Ramulator is internally built using a collection of lookup-tables, which are computationally inexpensive to query and update. This enables Ramulator to reduce the simulation runtime and provide cycle accurate performance models for a wide variety of standards such as DDR3/4, LPDDR3/4, GDDR5, WIO1/2, HBM, SALP, ALDRAM, TL-DRAM, RowClone, and SARP. Since the unmodified version of Ramulator does not support heterogeneous memory systems, the simulator is modified to model a heterogeneous setup. Different memory technologies have their own independent memory controllers with appropriate timing configurations.

3.1.2 System Timing Simulation

This dissertation uses Sniper multicore simulator as a front-end simulator that can feed memory requests into Ramulator. This simulator uses Pin tool as an instruction emulator, which provides x86 instructions. Sniper uses an interval simulation model where the simulation is done at a higher level of abstraction in comparison to traditional detailed cycle-accurate simulation.

The interval simulation abstracts the core performance, which takes a significant amount of simulation time in cycle accurate simulations, by leveraging a mechanistic analytical model. It improves the simulation performance, namely simulation time, by driving the timing simulation without the detailed tracking of individual instructions. This model tracks miss events (e.g., branch mispredictions, cache and TLB misses) and divides instruction streams into intervals. The analytical model derives the timing from miss events for each interval. The interaction between the analytical model and the miss events increases the accuracy in multi-core simulations. The simulator maintains a window of instructions, which corresponds to the reorder buffer. The overlap of long-latency load misses is found using miss events. The functional emulator, Pin tool, feeds instructions into this window at the window tail. For example, when there is I-cache miss, the core simulation time is increased by the miss latency. These long latency operations add the miss latency to the core simulated time, and the window for independent miss events that are overlapped by the long-latency load is scanned.

3.1.3 Measurement on Real Machine

The granularity-aware migration scheme in this dissertation is evaluated on a real machine. Since the scheme attempts to optimize the system overheads such as TLB shootdowns, which cannot be easily modeled in simulators, the real machine experiment is adopted. The experiments are performed on AMD A10-7850K processor [78] running Linux OS (kernel version 3.16.36) [79]. While systems with heterogeneous memory are imminent (e.g., Intel 3DXpoint [80]), they are yet to be available commercially. The heterogeneous memory system is emulated using Linux's NUMA emulation feature [79]. This feature allows us to divide the aggregate physical memory available in the system (32GB) in equally sized contiguous physical memory zones. Pages can then be migrated across these zones. Specifically, four physical memory zones with 8GB each are created. This configuration allowed each physical memory zone to be mapped on to separate DIMMS since the experimental machine has 4 DIMMs. Of these, one of the zones acts as die-stacked DRAM and another as the off-chip memory. It is important to ensure that applications' memory footprint exceeds the capacity of the die-stacked DRAM to trigger the page migration. Another Linux feature called Memory HotPlug [81] is utilized since it allows part of physical memory to be offlined as if it does not exist. The effective (usable) capacity of the die-stacked DRAM is reduced to 400MB in the baseline in order to emulate the real machine scenario where the die-stacked DRAM is a fraction of the application memory footprint.

3.1.4 TLB Performance Modeling

The performance of TLB is evaluated using a combination of real system measurement, Pin-based simulation, and performance models. The virtualization platform is QEMU 2.0 with KVM support and the host system is Ubuntu 14.04 running on Intel Skylake [28] with Transparent Huge Pages (THP) [32] turned on. It also has Intel VT-x with support for Extended Page Tables while the guest OS is Ubuntu 14.04 with THP turned on. The system has separate L1 TLBs for each page size (4KB, 2MB, and 1GB) though the applications do not use the 1GB size. The L2 TLB is a unified TLB for both 4KB and 2MB pages. Finally, the specific performance counters (e.g., 0x0108, 0x1008, 0x0149, 0x1049) that are used to read page walk cycles take MMU cache hits into account, so the page walk cycles used in this dissertation are the average cycles spent after a translation request misses in L2 TLB.

First, workloads are executed to completion and the Linux *perf* utility is used to measure the total instructions (I_{total}), cycles (C_{total}), number of L2 TLB misses (M_{total}) and total L2 TLB miss penalty cycles (P_{total}) in a manner similar to the methodology in prior work [29–31, 60, 82]. The baseline IPC is obtained as: $IPC_{baseline} = I_{total}/C_{total}$. Then, the ideal cycles C_{ideal} and the average translation penalty cycles per L2 TLB miss $P_{Avg}^{Baseline}$ are computed as:

$$C_{ideal} = C_{total} - P_{total} \tag{3.1}$$

$$P_{Avg}^{Baseline} = P_{total}/M_{total} \tag{3.2}$$

Note that the effects of various caching techniques like page walk caches, caching of PTEs in data caches, and Intel EPTs are already included in the performance measurement. Next, the Pin instrumentation and the Linux pagemap are used to generate memory traces for the workloads. For each workload, all load and store requests are recorded. The Linux pagemap is used to extend the Pin tool to include page size and other OS related meta-data. The trace contains virtual address, instruction count, read/write flag, thread ID and page size information of each reference. Memory instructions are traced in detail while the non-memory instructions are abstracted. The memory traces for 20 billion instructions are recorded.

Furthermore, this dissertation uses a detailed memory hierarchy simulator that models two levels of private TLBs, two levels of private data caches, and a 3rd level shared data cache. The simulator executes memory references from multiple traces while scheduling them at the proper issue cadence by using their instruction order. Information on the number of instructions in between the memory instructions are captured in the traces, and thus, the memory level parallelism and overlap/lack of overlap between memory instructions are simulated. Note that the simulator is using both the address translation traffic as well as data request traffic that go into underlying data caches. Finally, it reports the L2 TLB miss cycles and detailed statistics such as hits and misses in the L1/L2 TLBs, and data caches. The DRAM simulation accounts for access latencies resulting from row-buffer hits and misses. It may also be noted that, since the baseline performance, obtained from real system measurements, already includes the benefits of hardware structures such as large pages, EPT and Page Structure Caches, these are not modeled in the simulator. Instead, the baseline ideal cycles is used together with the estimated cost incurred by various TLB schemes.

Total cycles taken by the simulator and the resulting IPC for each core are obtained as:

$$C_{total}^{TLB} = C_{ideal} + M_{total} * P_{Avg}^{TLB} \tag{3.3}$$

$$IPC_{TLB} = I_{total}/C_{total}^{TLB} \tag{3.4}$$

P_{Avg}^{TLB} denotes the average L2 TLB miss cycles in TLBs obtained from simulation. Having obtained the baseline and IPCs for each core, the overall performance improvement of the TLB is calculated. It may be observed that the linear additive formula adds the L2 TLB miss cycles to the ideal cycles. This linear performance model ignores potential overlap of TLB processing cycles with execution cycles, but is similar to models used in previous research [29–31, 60, 82].

3.2 Benchmark Suites

This dissertation uses a number of benchmark suites to provide the necessary workloads for the simulations. The suites include single/multi threaded CPU, scientific and other typical representative applications that can drive multi-core simulations. This section provides a brief description of each benchmark suite to help aid the explanations.

3.2.1 SPEC CPU 2006

SPEC CPU2006 is a popular standard set of benchmarks used in the evaluation of the CPU performance. A total of 29 benchmarks consists of integer and floating point suites ranging from CPU intensive to memory bound applications. The workload does not use extensive system IO traffic and is single threaded. In order to use this suite on multicore platforms, multiprogrammed workloads are formed where each instance of an application is run on each core. Each instance is independent, and thus, does not share the memory address space.

3.2.2 PARSEC

The PARSEC benchmark suite [74] includes programs to evaluate multicore processors. It is multi-threaded and focuses on emerging desktop and server applications. Its diverse set of benchmarks is not skewed towards HPC programs. The suite includes benchmarks from computer vision, media processing, computational finance, enterprise servers and animation physics. In this dissertation, the number of threads used is equal to the number of processors, and the native input set is used to drive the simulations.

3.2.3 Other Benchmarks

Graph500: Graph 500 [75] is a compact application with multiple kernels that accesses a single data structure. It is designed with a scalable data generator, which produces edge tuples containing the start vertex and end vertex for each edge.

NAS Parallel Benchmarks: The NAS parallel benchmarks [83] are a set of programs designed to evaluate the performance of parallel supercomputers. It is based on computational fluid dynamics applications and consists of five kernels and three pseudo-applications.

HPC Challenge: The HPC Challenge suite [84] evaluates the performance of HPC architectures using kernels. The kernels use memory access patterns that are more challenging than those of the High Performance Linpack (HPL) benchmark. The benchmarks are scalable with the data set sizes being a function of the largest HPL matrix for a system.

Mantevo Benchmark Suite: Mantevo [85] consists of applications that are performance proxies known as miniapps. They encompass the dominant numerical kernels contained in stand-alone applications.

CORAL Benchmark Suite: CORAL [86] is a collaborative effort from National Laboratories to deliver three preexascale HPC. The benchmark ranges from complex applications to single node tests. It is scalable and provides performance improvement with weak and strong scaling.

Chapter 4

Challenges in Heterogeneous Memory System

This chapter describes the challenges associated with current die-stacked DRAM usage as well as address translation techniques in virtualized systems.

4.1 Challenges with Transparent Data Management in Die-Stacked DRAM

4.1.1 Architecting Near Memory as Part of Memory

Using die-stacked DRAM as a part of memory rather than caches poses sophisticated design challenges. First, when swapping is done without OS intervention, a remap table is necessary to allow dynamic migration of data between NM and FM. However, keeping an SRAM based remap table for every small block will easily exceed today's on-chip SRAM capacity [20]. As NM scales to a larger capacity, the remap table storage overheads become a more important issue. For example, 1GB requires 96MB of tag storage and approximately 100 cycles of lookup latency [42].Therefore, efficiently storing such large metadata is challenging and important. Increasing the remapping granularity to a large block (2KB) can mitigate this issue, yet on each remapping migration, the bandwidth usage becomes exorbitant as unnecessary subblocks also have to be migrated. Furthermore, recent work [22] has shown that as NM capacity scales, even making the remap granularity larger becomes problematic. Using the OS to intervene in the system and to perform the remapping can completely eliminate such remapping overheads. Yet, this has to be done at a large time interval to minimize the OS related overheads, so its adaptation to memory behavior changes is inherently slow. Therefore, carefully choosing the remap granularity and the frequency is crucial. Similarly, identifying what to place in NM is an important task since only a subset of the entire memory space can be held in NM. Currently, the OS main memory activity monitoring is done at a page granularity level based on the reference bit in Page Table Entry (PTE). This method limits the OS ability to accurately identify hot blocks to place in NM [18]. Therefore, a more detailed monitoring method is necessary.

4.1.2 Hardware Managed Schemes

Hardware data management schemes use different block sizes to migrate data between NM and FM. A block-based scheme swaps either small or large size blocks between two memories upon a request. This category of schemes attempts to exploit temporal locality as they expect swapped blocks in NM to be used frequently in the near future. To manage the mapping of swapped blocks, each block in NM has a remap table entry, which identifies whether a requested block resides in NM or FM. Schemes differ in their data management granularities from a small block (64B) to a large block (2KB). The key idea in this category of schemes is that, while it allows the NM capacity to be exposed to the OS, the dynamic remapping between NM and FM allows the data to move between two memories without OS intervention at low overheads.

CAMEO manages data at a small subblock granularity, and each block must have an accompanying remap table entry. The metadata, namely the remap table, is stored next to data within the same row in NM. During an NM access, the burst length is increased to fetch the extra bytes of metadata, and this saves latency as only one memory request is required per access instead of two. The direct-mapped organization is preferred as prior work [44] showed the difficulties involved with associative structures in NM as fetching multiple data in parallel from the same NM row is not possible. CAMEO has several disadvantages such as high metadata overheads and conflict misses. First, since the number of subblocks are high due to NM's large capacity, the remap table entries occupy a considerable capacity. Second, since CAMEO adopts a direct-mapped organization, conflicts misses are inherent. Third, by only swapping one small block at a time from FM, this scheme does not take advantage of abundant spatial locality at a large block level. The original CAMEO proposal does not implement any prefetching scheme, which might benefit high spatial locality workloads, thus it achieves a lower hit rate. Other work [22, 41, 87, 88] has shown CAMEO lost performance opportunities by only fetching 64B at a time. Therefore in addition to original CAMEO, this dissertation has also evaluated CAMEO with prefetching to see higher spatial locality effects.

PoM manages data at 2KB granularity unlike CAMEO to exploit spatial locality. It can perceive the performance improvement through migrating a large block into NM and getting a high hit rate. On the other hand, naively fetching a large block size for low spatial locality workloads is harmful as a large part of 2KB data that is brought into NM will not be used by an application. This results in wasted bandwidth as well as NM capacity. In order to mitigate such effects, the PoM scheme adopts a threshold based migration scheme where each block keeps track of the number of accesses made to that particular block. Once the counter goes over a certain predefined threshold, which seems beneficial based on the cost-benefit analysis, is migrated into NM. Until then, the block is not migrated and stays in FM. By doing so, the scheme is able to migrate only blocks that will overcome the migration overheads and benefit the system performance. The drawback of this scheme is that many applications do not need all subblocks within the 2KB segment. Since 2KB blocks are migrated all together, the bandwidth and capacity is still wasted. A better approach is selectively fetching subblocks that are needed, so the rigid 2KB requirement of PoM results in suboptimal performance. Finally, the scheme can be slow to phase changes as each 2KB requires the block to accumulate a certain number of accesses prior to become a candidate for a migration. Since it requires a certain period of time to elapse to get migrated, the potential performance improvement opportunities during this period is lost.

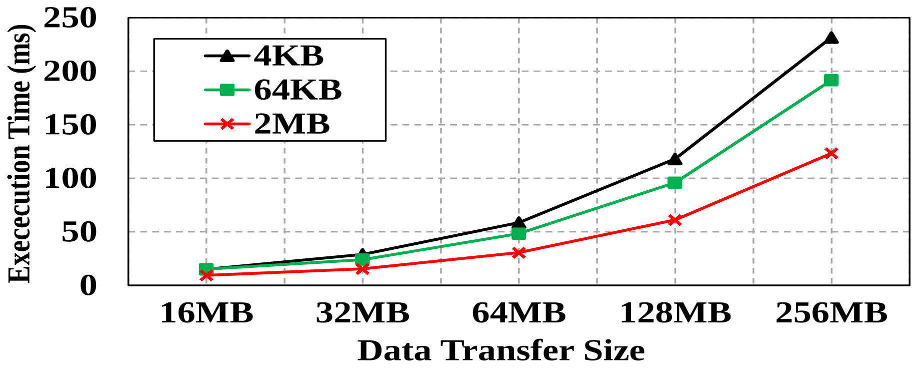
4.1.3 Software Managed Schemes

In this section, the term an "epoch" is used to describe a fixed time quanta (e.g., 100 million cycles). In an epoch based OS scheme, the OS explicitly manages the NM capacity as a special region of memory. In this dissertation, the state-of-the-art epoch based scheme, the HMA scheme [18], is used to describe the advantages and disadvantages. The HMA scheme relies on hot page detection to achieve performance improvement without additional hardware to perform sophisticated operations such as dynamic remapping. HMA uses a dynamic threshold based counter where the pages, whose access counts are higher than a set threshold, are marked using an unused bit in PTE. The page migration occurs at a large epoch. At each epoch boundary, the OS sweeps through the PTEs to select those pages that are marked, and the bulk page migration occurs between NM and FM. In addition to the time spent on physically transferring pages, this operation requires the system to update PTEs and invalidate corresponding TLB entries.

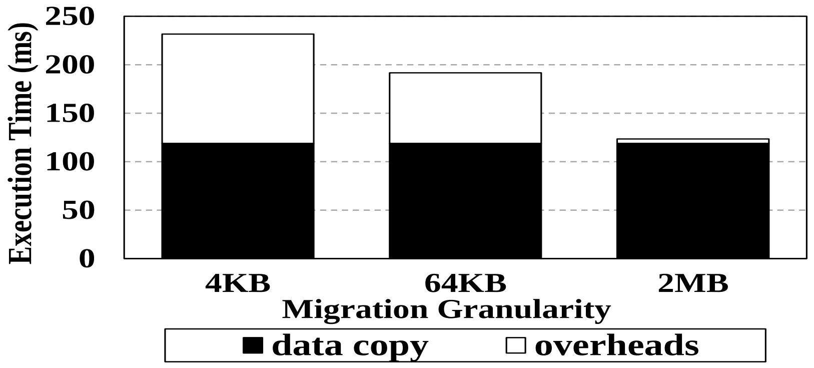
The HMA scheme has design and access latency advantages. Unlike hardware managed schemes, it does not have additional hardware structures like the remap table or modified addressing schemes, which result in unconventional data layout in a row. In CAMEO, the addressing scheme in the memory controllers has to be modified to fetch the remap table entry located next to every data block. Yet, the NM data layout and addressing of epoch based OS schemes is the same as in the conventional DRAM, so it does not require specialized logic in memory controllers. In addition, the HMA scheme reacts slowly to changes in the hot working set, which is a common behavior in many applications with different execution phases. For example, a page can become hot in FM, yet this page cannot be serviced from NM until the next epoch boundary, which may take millions of cycles to reach. Until then, the potential to get performance benefits from placing the page in NM is not exercised. Lastly, the working set coverage by NM is fixed during an epoch as no data migration occurs between NM and FM except at epoch boundaries. In contrast, if a larger working set coverage is desired, the hardware management scheme can swap blocks from FM and a larger amount of data can be served from NM at the end. Lastly, costs related to operations such as updating PTEs are extremely high, and thus the benefit of performing the block migration has to be large enough to offset the expensive costs [18, 21, 36].

4.2 Challenges with Data Migration in Heterogeneous Memory

The OS driven data migration in heterogeneous memory is not free of costs. In a migration, any time spent beyond the copying of the data from one type of memory to the other is considered as overheads of migration. TLB shootdowns and the process of invoking migrations [18, 48, 89] contribute significantly to this overhead. Whenever a PTE is updated, TLB shootdowns ensure that TLBs are coherent across all cores. The granularity at which the data is migrated impacts this overhead. For example, if a 2MB contiguous memory region is migrated at 4KB (default) granularity, then it incurs 512 TLB shootdowns – one each for 512 4KB pages. However, if the same memory region is migrated at 2MB granularity, then it will incur only one shootdown. Thus, migrating in larger granularity could amortize the overheads better.



(a) Time to Migrate Contiguous Virtual Memory with Varying Migration Granularities.



(b) Breakdown of Migration Latency for Different Granularities.

Figure 4.1: Effect of Migration Granularity in Cost of Migration.

The effect of migration granularity on the migration overhead is quantified using a system with AMD A10-7850K processor [78] running Linux OS. Figure 4.1a shows the time to migrate varying amount of data at different granularities when pages are moved from one *emulated* NUMA node to the other (although the system has 4 NUMA nodes, only 2 nodes are used to emulate fast and slow memory). The x-axis shows the varying amount of contiguous virtual memory being migrated while the y-axis shows the time required to migrate. The measured time includes the time to physically migrate pages

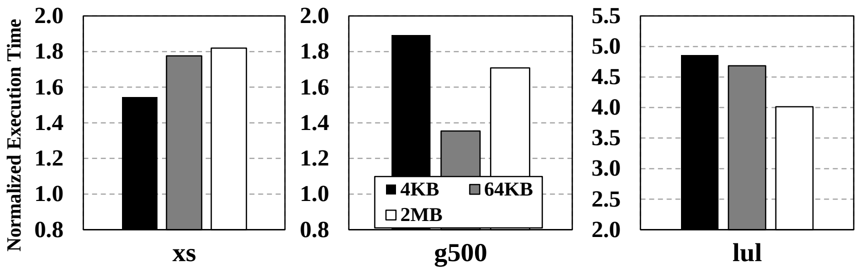


Figure 4.2: Impact of Migration Granularity on Application Performance

as well as accompanying overheads such as TLB shootdowns and OS context switches. Three lines in the graph represent three different migration granularities. It is apparent that as the amount of data being migrated increases, the larger granularity takes less time to migrate a given amount of data by amortizing overheads.

Figure 4.1b shows the breakdown of execution time while migrating 256MB of contiguous virtual memory. Linux's *perf* utility [90] is used for this measurement. When a smaller granularity is used, then the overhead of migration dominates while, with a larger granularity, the overhead goes down significantly.

However, a larger migration granularity does not necessarily improve an application's overall performance in a heterogeneous memory system. Figure 4.2 shows the normalized runtime of three representative applications with different migration granularities. There are three subgraphs, one for each individual application. Each subgraph has three bars representing the normalized runtime (y-axis) under the migration granularity of 4KB (default), 64KB and

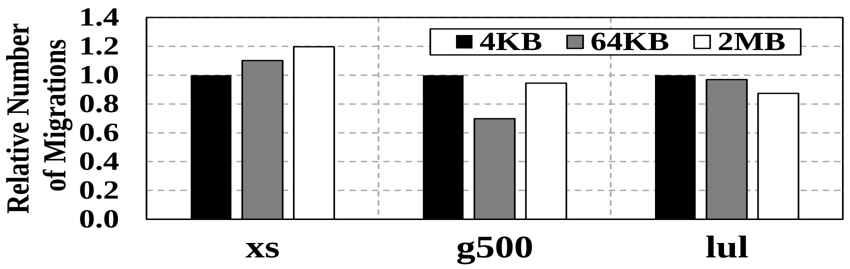


Figure 4.3: Relative Number of Migrations Between Fast and Slow Memory (lower is better)

2MB respectively. The y-axis values are normalized to the runtime when there is no migration, i.e., when the die-stacked DRAM capacity is configured to fit the entire memory footprint of a given application. The experiment was performed with the die-stacked DRAM capacity of 400MB where the application footprints were 5.5GB, 761MB, and 696MB for *xsbench*, *graph500*, and *lulesh* respectively. The application *xsbench* ("xs") performs best when the migration granularity is set to 4KB while *lulesh* ("lu") prefers 2MB granularity. The *xsbench* benchmark has near-random memory access pattern that rarely shows spatial locality across multiple neighboring pages. At the same time, the main loop of *lulesh* is close to purely streaming-like memory access behaviors. Application *graph500* ("g500"), on the other hand, performs best with 64KB granularity. Though this application performs pointer-chasing on a graph, it shows some access locality across a few neighboring 4KB pages, but locality quickly falls off beyond a handful of pages.

Further analysis is performed to quantify why some applications lose performance with larger migration granularity. The number of migrations is collected for each application with different migration granularities as shown in Figure 4.3. Like the previous figure, this figure also has three subgraphs, one for each application. Each subgraph has three bars representing the normalized number of migrations with different migration granularities. The height of each bar is normalized to the number of migrations with 4KB granularity. The application *xsbench* has the highest number of migrations when the migration granularity is the largest since unnecessary pages occupy the die-stacked DRAM capacity and cause more capacity misses. Moving the data in a larger granularity in die-stacked DRAM wastes its capacity by evicting otherwise useful pages from die-stacked DRAM. These evicted pages would later need to be migrated back again to die-stacked DRAM when demanded by the application; increasing the number of migrations. Furthermore, extra data migrated into die-stacked DRAM due to a larger granularity is rarely useful to this application due to its near-random accesses. Application *lulesh* makes better use of a larger amount of data migrated with a larger granularity, and thus, lowers the total number of migrations. As expected, *graph500* is again somewhere in the middle. It can make use of a larger granularity up to a certain point (here 64KB), but the usefulness of even a larger granularity (2MB) falls off due to the larger number of migrations. In hindsight, this is intuitive as the benefit of lowering migration overhead with a larger migration granularity could be offset by extra migrations if an application does not make good use of additional migrated data.

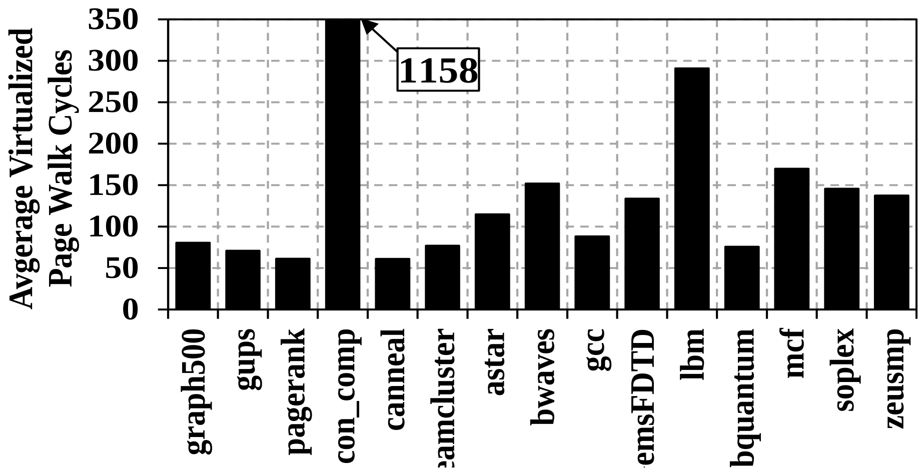


Figure 4.4: Average Translation Cycles per L2 TLB Miss (Virtualized Platform)

4.3 Challenges with Address Translation in Virtualized Systems

In this section, the performance overheads associated with address translation in virtualized systems is presented. Figure 4.4 is based on our experiments on a state-of-the-art Intel Skylake system (i7-6700). It shows the average number of cycles spent in address translation (per L2 TLB miss) in several SPEC, PARSEC and graph workloads. The workloads are run on a VM with Linux THP support enabled, and the performance overhead is measured using the Linux *perf* utility. The translation overhead per L2 TLB miss is seen to range from 61 cycles in the canneal benchmark to 1158 cycles in the connected component graph benchmark. The translation overhead running into 100+ cycles has also been reported in prior work [59, 91, 92].

The experiments on the Intel Skylake platform also shed light on the virtualization overhead compared to native execution of the same workload.

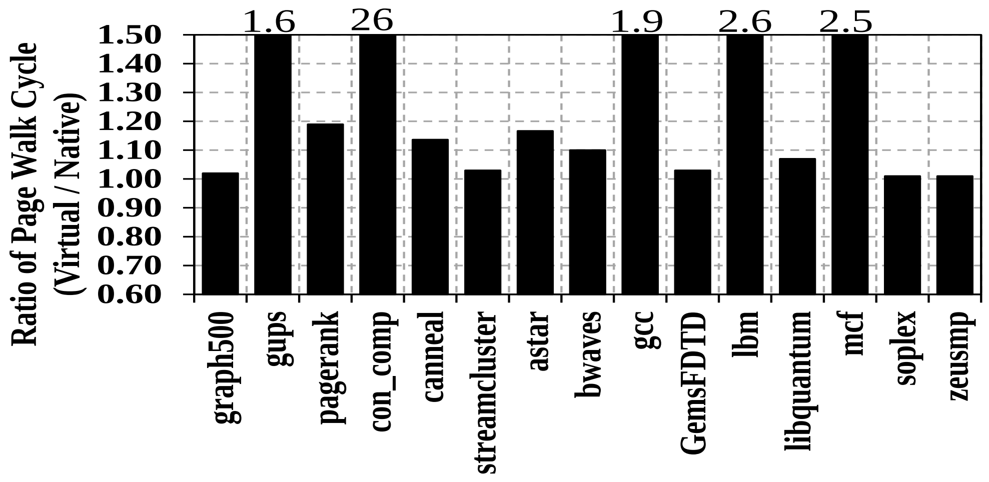


Figure 4.5: Ratio of Virtualized to Native Translation Costs

Figure 4.5 plots the ratio of translation cycles in virtualized and native setups. Workloads such as gups (1.5x), con_comp (26x), gcc (1.9x), lbm (2.5x) and mcf (2.5x) have far higher translation overhead in virtualized execution compared to native execution. Many benchmarks spend up to 14% execution time in translation even in the native case.

With the increased number of cores and big data sets, the conventional two-level SRAM TLBs cannot hold translations of all pages in the working set. Increasing L2 TLB sizes to reduce TLB misses is not practically feasible because larger SRAM TLBs incur higher access latencies. Using the CACTI [93] tool, the access latency sensitivity study is performed with larger L2 TLB capacities in Figure 1.2. The access latency is normalized to that of 16KB SRAM. As seen previously, naively increasing the SRAM capacity does not scale, and it is not a solution to the limited reach problem of today's TLBs.

Chapter 5

SILC-FM: Subblocked InterLeaved Cache-Like Flat Memory Organization[3]

With current DRAM technology reaching its limit, emerging heterogeneous memory systems have become attractive to keep memory performance scaling. This dissertation argues for using a small, fast memory closer to the processor as part of a flat address space where the memory system is composed of two or more memory types. OS-transparent management of such memory has been proposed in prior works such as CAMEO [20] and Part of Memory (PoM) [21] work. Data migration is typically handled either at coarse granularity with high bandwidth overheads (as in PoM) or at fine granularity with low hit rate (as in CAMEO). Prior work uses restricted address mapping only from congruence groups in order to simplify the mapping. At any time, only one page (block) from a congruence group is resident in the fast memory.

A flat address space organization called Subblocked InterLeaved Cache-like Flat Memory (SILC-FM) [87, 88] that uses a large granularity but allows

[3]J. Ryoo, A. Prodromou, M. R. Meswani, and L. K. John, "SILC-FM: Subblocked Inter-Leaved Cache-Like Flat Memory Organization", in *Proceedings of the 23rd Symposium on High Performance Computer Architecture*, 2017. Jee Ho Ryoo contributed to the idea development and implementations. Andreas Prodromou was involved in the initial experimental simulation setup. Mitesh R. Meswani and Lizy K. John supervised the overall project.

subblocks from two pages to coexist in an interleaved fashion in NM is presented in this dissertation. Data movement is done at subblocked granularity, avoiding fetching of useless subblocks and consuming less bandwidth compared to migrating the entire large block. SILC-FM can get more spatial locality hits than CAMEO and PoM due to page-level operation and interleaving blocks respectively. The interleaved subblock placement improves performance by 55% on average over a static placement scheme without data migration. The scheme also selectively locks hot blocks to prevent them from being involved in the hardware swapping operations. Additional features such as locking, associativity and bandwidth balancing improve performance by 11%, 8%, and 8% respectively, resulting in a total of 82% performance improvement over a no migration static placement scheme. Compared to the best state-of-the-art scheme, SILC-FM gets a performance improvement of 36% with 13% energy savings. The technique provides the following insights:

- The high bandwidth costs typically associated with page-level data placement and migration in prior PoM can be eliminated by using subblocking. **Subblocking** together with **interleaving** subblocks from two different pages increases the usefulness of the fast memory layer.
- It is preferred that hot pages reside in on-chip memory as those pages benefit from high bandwidth. A mechanism is developed to identify hot pages and to **lock** them in fast memory, so that conflicting data does not swap hot pages out to off-chip DRAM. Unlike epoch based approaches,

the locking does not need to occur at coarse grain time intervals, and thus the proposed scheme can react quicker to the changes in the hot working set.

- In memory bandwidth limited systems, the bandwidth is the scarce resource. Thus, instead of maximizing the total number of requests serviced from the fast memory layer, it is beneficial to service a fraction of requests from the off-chip DRAM to utilize the overall available system bandwidth. If the bandwidth available from the two memory levels are N:1, it is beneficial to service 1/(N+1) of the accesses from the slower memory layer. Therefore, SILC-FM achieves high performance improvement with a slightly lower number of requests serviced from die-stacked DRAM. This leads to **balanced** bandwidth utilization, which in turn provides performance improvement.

SILC-FM is evaluated against the state-of-the-art scheme, which also fully utilizes the added fast memory capacity as a part of memory. It outperforms the state-of-the-art scheme by 36%. Also, with die-stacked DRAM's low energy consumption, it reduces the Energy-Delay Product (EDP) by 13% in comparison to the state-of-the-art scheme.

5.1 SILC-FM Memory Architecture

In this section, the details of SILC-FM are presented. The SILC-FM scheme uses NM as OS visible space while internally operating with subblock

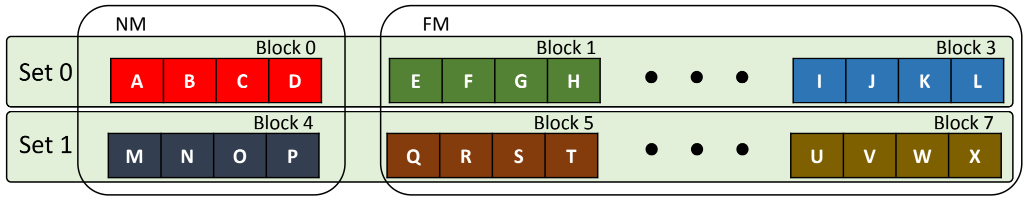


Figure 5.1: NM Set Mapping (Each Set Contains 32 Subblocks in Implementation)

based mechanisms. NM is organized as associative structure where subblocks are swapped between NM and FM. A locking feature prevents hot blocks from being swapped out to FM while a bypassing feature utilizes the overall system bandwidth.

Figure 5.1 shows the initial memory state where each row is a congruence set. The congruence set is the similar concept used in prior work [20, 21]. In this example, each block contains four subblocks and the mapping is direct-mapped. This implies that multiple subblocks from only one large block (from the same set) in FM can swap into corresponding subblocks in NM at any one point of time. For example, subblock A and C are in the same large NM block whereas another subblock F is in different large block in FM, so the subblocks F and J can only swap into subblock B in NM. At any point in time, either only one of subblock F or J can reside in NM. In Figure 5.1, the migration between two pages within the same set occurs at subblock granularity. This is bandwidth efficient as only 64B worth of data is migrated, yet managing the remap table at a large block granularity reduces the remap table overheads. The subblock tracking is done using a bit vector per NM block where individual bits validate the corresponding subblock's residency in NM. When a block is swapped out of NM, the history of bit vectors is stored in a small SRAM structure called a bit vector table. The bit vector history table has a total of 1 million entries (approximately 4MB). Multiple subblocks are fetched using this bit vector when a block is swapped in again. This exploits spatial locality as previously used subblocks are swapped at the same time, so any subsequent request to either of the subblocks results in a subblock serviced from NM. In comparison to CAMEO, SILC-FM can achieve higher spatial hits. Since this scheme does not swap any other undesirable subblocks, it is more bandwidth efficient than large block based schemes such as PoM, which have to swap every subblock within a large block.

Since NM is not a cache, the term "hit rate" is not used to describe the fraction of requests serviced from NM. Rather, the term, "access rate", is used as done in a recent die-stacked DRAM paper [20]. The access rate is defined as in Equation 5.1. In this dissertation, assume that NM uses the lower addresses in the physical address space and FM uses the higher addresses.

$$AccessRate = \frac{\substack{total\ number\ of\ requests \\ serviced\ from\ NM}}{\substack{total\ number\ of\ requests \\ missed\ from\ LLC}} \tag{5.1}$$

5.1.1 Hardware Swap Operations

Since NM is a part of memory space, when data is brought into NM, the existing data from NM needs to be swapped out. Unlike hardware caching schemes where there is always a copy in FM, SILC-FM needs to perform a

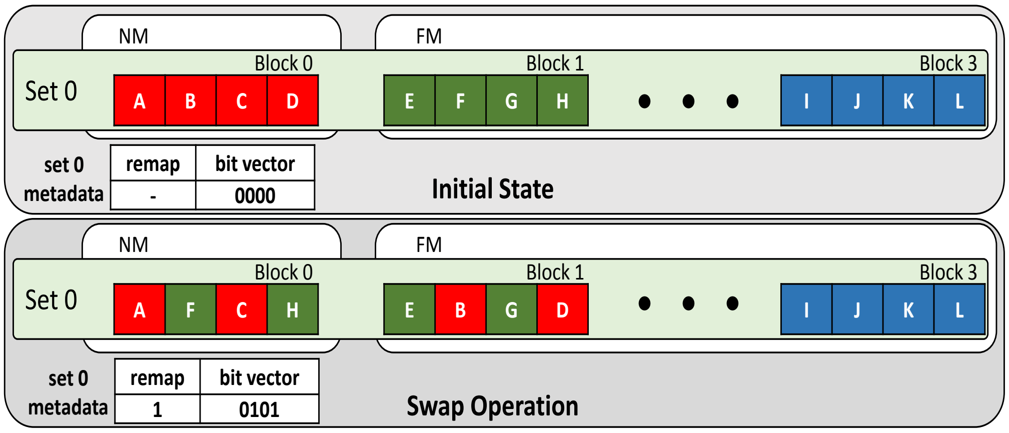


Figure 5.2: Example of Interleaved Swap

swap operation. Figure 5.2 shows a direct mapped scheme to describe the swapping operation for subblocks F and H. When two back to back requests are made to subblock F and H, they are brought in one by one from block 1 in FM into block 0 in NM. The corresponding subblocks (subblock B and D) in block 0 are swapped out to block 1. Any subsequent access to subblock F and H will be serviced from NM. There are no duplicate copies of data and hence the total memory capacity is the sum of NM and FM capacities, which is much greater than in cache schemes such as Alloy Cache [44]. In SILC-FM, the address is used to calculate an index, which refers to a set of unique NM blocks. Upon a memory request, the index is calculated by performing the modulo operation with the incoming address and the total number of blocks in NM. Then, the incoming address is again checked against the remap field of the metadata. The remap entry is only used for the swapped in data from FM since it contains the large block address of the swapped in block.

Remap	Bit Vector	NM Address	Action
match	1	-	service from NM
match	0	-	swap subblock from FM
mismatch	1	yes	swap subblock from FM
mismatch	0	yes	service from NM
mismatch	1	no	restore current block and swap subblock from FM
mismatch	0	no	restore current block and swap subblock from FM

Table 5.1: SILC-FM Metadata and Operation Summary

Subblocking is done via a bit vector, which consists of valid bits to indicate whether a particular subblock in this NM block has been swapped in from FM. In Figure 5.2, when subblock H is brought into NM, the corresponding bit, which is calculated using the block offset, is set in the bit vector. Also, the remap table entry is updated, so it contains the block address of the swapped in block, which is 1. Note that SILC-FM does not have a valid bit at block granularity. Unlike caches, there is always data in NM since NM is used as flat memory, so the block is always valid. The difficulty here is to distinguish which block the data belongs to. SILC-FM can achieve that by using the NM address range and remap entry. Using the remap table entry, bit vector and the request address, there can be 6 scenarios of swap operations. Table 5.1 lists each operation and this table will be referred to explain each swap operation.

First, the example begins with the case where the remap entry matches with the request address. If a bit in the bit vector is not set and the request address belongs to the NM address space, then the subblock (original NM subblock) is resident. Otherwise, if the bit is set, then the original subblock is swapped out to FM.

In the scenario with a remap entry mismatch, the bit is set, and the address falls in the NM address space, then the swapped out subblock (the one originally belonging to NM) is brought into NM. The FM location where the swapped out subblock currently resides is the block address in the remap entry, so the remap entry is consulted to bring the subblock back. In the same scenario with the bit not set, the original subblock is resident, so the request is serviced from NM. Lastly, if the remap entry mismatches and the requesting block address belongs to the FM space, the original mapping is restored. Although the bit vector is not used to perform an execution action, the bit vector is consulted to restore the original mapping. At the same time, the bit vector is saved in a bit vector table. This is a small SRAM structure that is indexed using the xor'ed PC and the address of the first swapped-in subblock within the block (first in timely sense). Thus, this PC and request address is stored along with other metadata for each block. These two variables have shown to have high correlation with the program execution [57, 94–96], so when this block is accessed again in the future, it is likely that a similar access pattern will repeat. Since the bit vector has a pattern of previous subblock usage, it will be used to fetch multiple subblocks when this block is

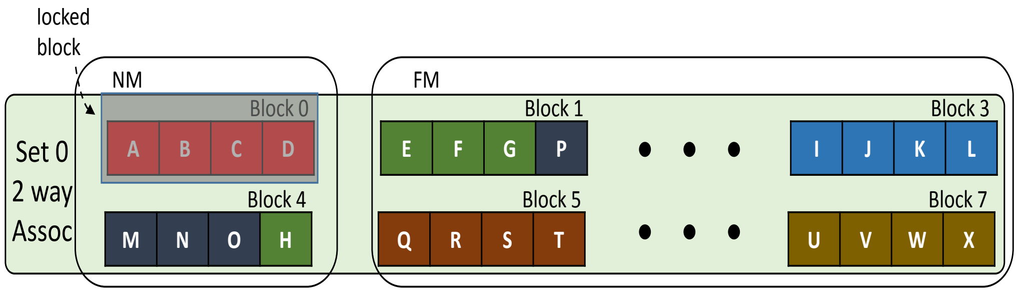


Figure 5.3: Locking and Associativity

swapped into NM again. The bit vector table is indexed and those subblocks corresponding to 1's in the bit vector are fetched together from FM, which takes advantage of spatial locality. The concept of using a bit vector is similar to prior work [15, 97]. By doing so, SILC-FM can achieve a higher access rate than small block schemes. Now, after successfully restoring a block, the new swapping occurs between NM and FM.

If a FM subblock has to be swapped into NM, the corresponding subblock in NM (the subblock that originally belongs to the NM address space) is swapped out to FM. The corresponding bit is set and the subblock is now resident. Since the large block size is 2KB and the subblock size is 64B, there are 32 bits per block. Each bit is responsible for each subblock position in NM. Unlike caches, dirty bits are not tracked in NM. In caches, dirty bits facilitate the eviction process by only writing back dirty data and invalidating clean data. However, since data in NM is the only copy of the data in the physical address space, all swapped in blocks need to be written back to FM when necessary. Therefore, SILC-FM does not need dirty bits.

5.1.2 Memory Activity Monitoring

SILC-FM monitors memory access activities to classify data into hot and cold data. The idea is to keep only hot pages in NM to benefit from NM's high bandwidth. The cold data should not interfere with hot data so that hot data is not inadvertently swapped out to FM. The activity tracking metadata, namely NM and FM counters, are used to gather memory access statistics, and each page in NM has its own dedicated set of NM and FM counters. Each NM page has two counters stored in NM, each with 6 bits, so the total area overhead is 1.5MB, which is negligible. If there are any swapped subblocks in NM, then two different blocks coexist in the same NM row; ones originally in NM space and the others swapped in from FM. Unlike the remap entry where only one entry is needed to distinguish those two sets, for monitoring activities, SILC-FM needs two sets of counters, each for NM and FM blocks. The counter is used to classify two coexisting blocks as either hot or cold blocks. Upon an access, the updated counter value is compared against a threshold and if it exceeds this value, the large block is hot. Otherwise, it is considered cold. This later helps to identify candidates to lock in NM. In order to distinguish between current and past hot blocks, these counters are implemented using aging counters where the counter value is shifted to the right every one million memory accesses.

5.1.3 Locking Pages

Once the system identifies hot blocks, SILC-FM locks hot blocks in NM as those blocks are responsible for high bandwidth usage, which benefits from being placed in NM. When the counter crosses the threshold, the block is locked. To reduce hardware complexity, the locking is done at large granularity although the unlocked pages still operate at subblock granularity. Unlike unlocked blocks, the locked blocks have all their subblocks in NM. Therefore, when locking the block, the missing subblocks, which are residing in FM, are swapped into NM. After locking, a complete large block remap has been performed as the large block originally belonging to NM is now completely remapped to a location in FM and vice versa. The counter for this locked page is still incremented upon each access, but the bit vector checking is ignored. The counter is still monitored to ensure that the locked block is still hot. If the locked block is no longer hot and the access count goes below the threshold, the lock bit is unset. Clearing the lock bit does not have an immediate effect as it operates as if the unlocked block has all subblocks swapped in from FM (all bits in the valid bit vector are set). If this block is, in fact, no longer hot, then other hot subblocks will be swapped into this place in NM. The locking and unlocking mechanism can react quicker to changes in the hot working set than epoch based schemes as migrating hot blocks do not have to wait until epoch boundaries. In fact, the proposed scheme does not have any notion of epochs, so locking and unlocking can happen at anytime for any number of blocks.

One drawback with locking is that it makes other subblocks in the same FM set inaccessible to NM. They can only be swapped into NM in the same set when a direct-mapped scheme is used. Therefore, SILC-FM allows swapped in subblocks to be placed with some flexibility by allowing block level associativity. Using a large block size (2KB), different associativities are experimented. A direct-mapped organization (1-way associativity) achieves the least performance improvement as several hot blocks get swapped out due to conflicts and thrashing. Increasing the associativity to two improves performance by removing many conflicts. Yet, the index is still calculated using a part of the address bits, so multiple hot blocks (more than two) are still mapped to the same set. Consequently, increasing the associativity to four further improves the performance. The latency overhead associated with the increased associativity is modeled in the simulation. As a result, SILC-FM adopts four-way associative structure. Therefore, the incoming address is checked again all ways to find the matching remap entry. Prior page-based work [21, 22, 39] showed similar results regarding associativity, yet the use of associativity is distinct from prior work in that depending on the number of pages locked, the associativity can vary from 1-way to 4-way.

5.1.4 Metadata Organization

The overall metadata organization in SILC-FM is shown in Figure 5.4. The block address for those swapped in subblocks are stored in the remap field as shown in the figure. The FM counter tracks the number of accesses made

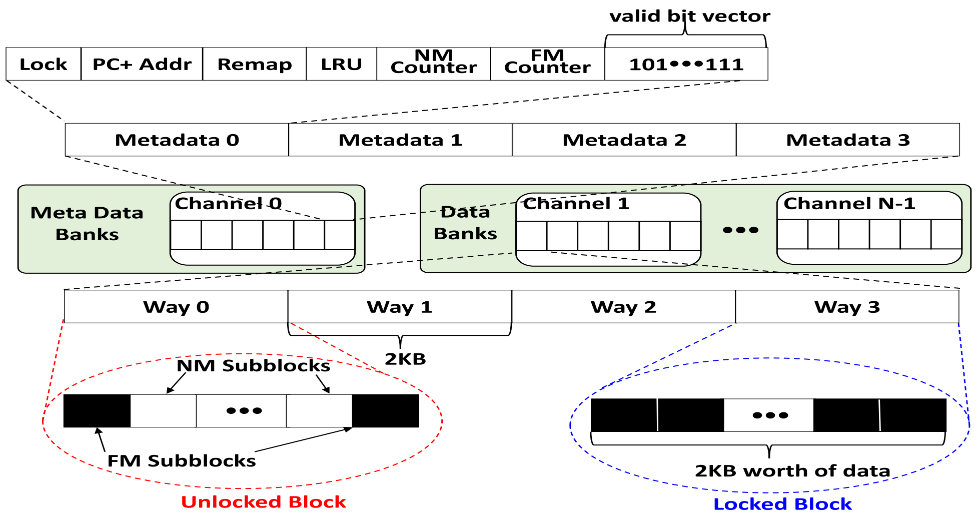


Figure 5.4: SILC-FM Data and Metadata Organization

to these swapped-in blocks. The FM counter is used since those subblocks originally belong to the FM address space. The block corresponding to Way-0 shows the layout of the data where subblocks shaded in black within 2KB block indicate subblocks which are swapped-in. Those subblocks which are swapped in originally belong to FM address space, but they are brought into NM upon a request. The white subblocks are data belonging to the original pages that reside in NM address space. Upon a request, this remap entry has to be checked to determine if the requested block has been swapped in or not. SILC-FM maintains one remap entry per large block/page while the residency of subblocks within a block is validated using a bit vector. SILC-FM also has other metadata fields such as lock and LRU, which are used for locked pages and finding the swapping candidate in NM. The total metadata size is 8 bytes per 2KB, so it occupies 0.39% of the NM capacity. The metadata is stored in a

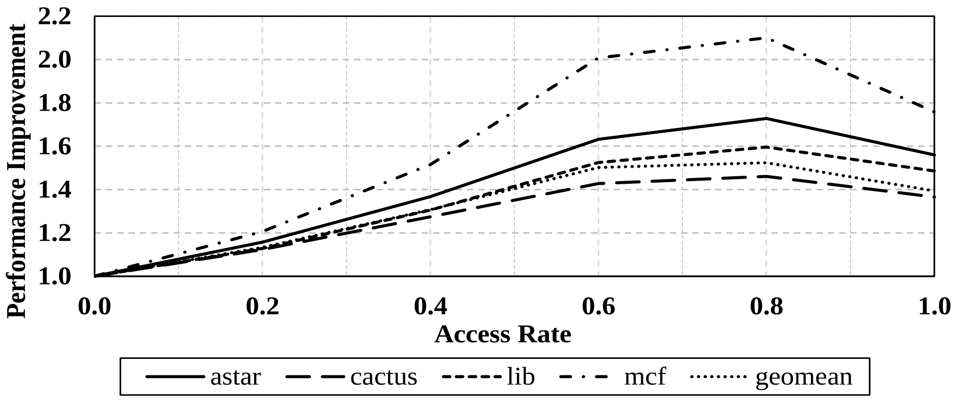


Figure 5.5: Performance Improvement with Varying Access Rate

separate channel to increase the NM row buffer hit rate of accessing metadata. Separating the metadata storage from data has been shown to increase the row buffer locality [39].

5.1.5 Bypassing and Bandwidth Balancing

Always swapping subblocks upon access increases the overall access rate. Although such an approach makes sense for caches where NM is considered another level in memory hierarchy, it may leave the available bandwidth to FM idle once the access rate becomes high. In cache-like organizations, FM being idle is actually beneficial since all requests are serviced from a layer of memory closer to the processor. Yet, in the situation where NM is a part of the memory space just like FM, having a portion of the memory being idle has similar effects as disabling a fraction of the memory in the same memory hierarchy. This is not a desirable outcome, and in this case, making use of the

FM bandwidth can increase the effective available bandwidth to the system. The effects of bypassing using the CAMEO scheme are performed and the FM is steered an appropriate amount of traffic to match the desired access rate. Unlike SRAM caches where the maximum performance is achieved at 100% hit rate, in this experiment, the optimal performance improvement point is at 0.8 instead of 1.0 as shown in Figure 5.5. In the experimental platform, the available bandwidth ratio between FM and NM is 4:1, so routing 80% of the traffic to NM and 20% to FM matches this bandwidth ratio. Since NM and FM are at the same memory hierarchy level, using this FM bandwidth makes the processor utilize all bandwidth available in the system at the NM and FM memory layer. Thus, it is able to gain extra performance from the memory system. Since SILC-FM focuses on memory bandwidth bottlenecked systems, having more bandwidth available to the application helps. In addition, prior work [98–100] also showed that bandwidth throttling is effective in bandwidth constrained systems. However, the bypassing in SILC-FM is at much finer granularity (at page level) than in prior work, which uses a segment of memory as bypassing zones. In SILC-FM, its goal is to make the access rate be 0.8, so it incorporates the bypassing feature if the access rate exceeds this value. When this happens, no more subblocks are swapped into NM. However, the unlocked blocks, which are already in NM, can still operate normally from NM. For example, subblock G in Figure 5.3 can be swapped into block 1 upon a request and still work under the unlocked condition. However, in the bypassing scenario, this swapping is not allowed, so the bit vector will not be updated, and subblock G will be serviced from FM. However, if the access rate again becomes lower than 0.8, this bypassing feature is turned off to increase the access rate.

5.1.6 Latency Optimization

Having an associativity structure adds to the NM access latency as fetching multiple remap entries is a serialized operation. In SRAM caches, it does not add significant latency as multiple entries can be fetched and checked simultaneously. However, that is not the case for NM since NM uses DRAM based technology. The implementation uses 4-way associativity, so in each access, four independent remap entries are fetched. This operation has to be serialized as the maximum fetch bandwidth is limited by the bus width. The metadata serialization problem was also addressed in prior work [42, 44]. In order to hide the long latency of fetching multiple remap entries, a small predictor is added to bypass this serialization. The predictor has 4K entries in total. The predictor uses the instruction and data addresses since they are known to have a strong correlation with the execution phase of a program [57, 94–96], and thus, they are widely used as predictors in DRAM caches [39, 41, 44]. The program counter and data address offset values are xor'ed to form an index into the predictor. This table keeps track of a recently accessed way for each particular index. On each access, the index is calculated to access this table. Since this table is a small structure, the access latency is negligible. However, the access to the table begins with LLC access, so by the time the LLC miss is identified, the predicted way is available to access NM. Furthermore, one more bit in each entry is added to speculate on the location of the data (NM or FM).

The latency on an access to FM is longer than on an access to NM since the remap entry has to be checked first in NM prior to accessing FM. The predictor attempts to ameliorate this issue. If the predictor speculates that the data is located in FM, then the request is sent to FM at the same time as the remap entry request is sent to NM. Upon correct speculation that the data is in FM, the latency is just a single FM access latency, hiding the NM remap entry fetching latency. Therefore, the saved time is the NM access latency. Note that the predictor only forwards the requests to FM when the block is speculated to exist in FM. In the case where the block is speculated to be in NM, no additional action is taken. If this prediction was not correct (e.g., FM was speculated, but data is in NM), the simultaneously forwarded request to FM is ignored.

5.1.7 Overall SILC-FM Operation

Now, the overall scheme is explained from the point of view of an LLC miss using Figure 5.6. The congruence set index is calculated using the modulo operator to access both the remap entry and data in NM. Also along with LLC access, the PC and the request address are used to access the predictor. The request is sent to NM using the calculated index and predicted way. If the lock bit is set and the remap field is a match, the NM data is fetched. In the

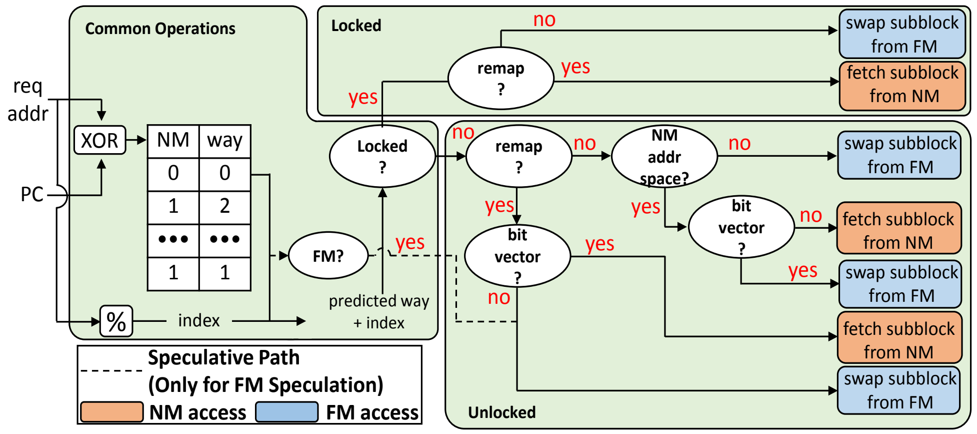


Figure 5.6: Overall Execution Flow of SILC-FM

case of a remap mismatch, the request address is checked to see whether it falls under the NM address space. If so, the bit vector is checked to determine the location of the requested subblock. If the subblock is resident in NM, then the bit has to be 0. The prediction, shown in a dotted line, can skip the previously mentioned metadata fetching steps. If the block address does not fall under NM space, then the update of the remap entry and swapping of the subblock from FM are initiated. The subblock is swapped to available ways within the set. A similar operation occurs for a remap mismatch. If the request was made to one of locked blocks, the remap entry is checked. If it matches, then the corresponding subblock is fetched from NM. If not, then the subblock is swapped from FM to NM blocks other than this locked block. Every swap from FM operation, as shown in Figure 5.6, checks the access rate. If it is enabled, then the swap from FM becomes a fetch from FM without any metadata update. Also, only the correct way speculation path is shown. In the

Processor	Values
Number of Cores (Freq)	16 (3.2GHz)
Width	4 wide out-of-order
Caches	**Values**
L1 I-Cache (private)	64 KB, 2 way, 4 cycles
L1 D-Cache (private)	16 KB, 4 way, 4 cycles
L2 Cache (shared)	8 MB, 16 way, 11 cycles
HBM	**Values**
Bus Frequency	800 MHz (DDR 1.6 GHz)
Bus Width	128 bits
Channels	8
Ranks	1 Rank per Channel
Banks	8 Banks per Rank
Row Buffer Size	8KB (open-page policy)
tCAS-tRCD-tRP-tRAS	7-7-7-28 (memory cycles)
DDR3	**Values**
Bus Frequency	800 MHz (DDR 1.6 GHz)
Bus Width	64 bits
Channels	4
Ranks	1 Rank per Channel
Banks	8 Banks per Rank
Row Buffer Size	8KB (open-page policy)
tCAS-tRCD-tRP-tRAS	11-11-11-44 (memory cycles)

Table 5.2: SILC-FM Experimental Parameters

case of a way prediction misspeculation, the remap entry check takes longer as four remap entries are checked in series.

5.2 Results

5.2.1 Experimental Setup

To evaluate the SILC-FM scheme, a Pin-based Sniper simulator [71] is used to model a 8-core server class processor similar to AMD's A10-7850K processor [78]. Each core is a 4-wide issue processor with 128 ROB entries per core. A detailed memory simulator, Ramulator [72], is configured with 32-entry read and write queues per channel. Timing and configuration parameters are listed in Table 5.2. The simulator performs the virtual-to-physical address translation and assumes that FM to NM capacity ratio is 4:1. For NM memory, the configuration uses HBM Generation 2 and derived timing parameters from JEDEC 235 and 235A datasheet [6, 101] along with other publicly available sources [7, 24, 102, 103]. DDR3 technology is used as FM memory with latency parameters derived from JEDEC and vendor datasheet [104, 105]. The baseline is a system without NM. The bit vector history table is 72KB and the predictor is 1.5KB, both with an access latency of 1 cycle. The access latency of SRAM structures is derived from CACTI [93]. Experimentally for the NM, the FM counter threshold of 50 is found to work the best to determine the block hotness, so this value is used. The execution time is calculated using the time when all workloads in all cores terminate. The speedup (figure of merit) is calculated using the total execution of the baseline with no HBM memory over the execution time of a corresponding scheme, and therefore, higher speedup represents higher performance.

SILC-FM is compared against other five other designs: Random Static Placement (rand), HMA (hma), CAMEO (cam), CAMEOP (camp), and Part of Memory (pom). Random uses the entire NM and FM as OS visible address space and maps pages randomly. Thus, this scheme does not consider different bandwidth/latency characteristics of NM and FM, but rather, treats them

Category	Benchmark	MPKI (per core)	Footprint (GB)
Low MPKI	bwaves	10.12	6.82
	cactus	7.52	2.31
	dealII	4.46	0.69
	xalanc	5.98	2.87
Medium MPKI	gcc	31.23	1.34
	gems	15.95	10.59
	leslie	11.28	1.19
	omnet	27.22	2.06
	zeusmp	11.41	3.32
High MPKI	lbm	53.29	6.30
	lib	35.50	0.50
	mcf	88.95	18.46
	milc	34.13	9.05
	soplex	43.32	0.78

Table 5.3: SILC-FM Workload Descriptions

the same. HMA and CAMEO are described in Chapter 4. CAMEOP is an enhancement to the CAMEO scheme with a prefetcher that fetches an extra 3 lines along with the miss. Lastly, Part of Memory [21] migrates 2KB blocks based on block access counts.

5.2.2 Workloads

A representative region of SPEC CPU2006 benchmark suite [106] is run using 1 billion instruction slice Simpoint [107, 108]. Only a subset of the benchmarks, which exhibit high memory bandwidth usage, are used. Chosen benchmarks are categorized into three groups: low, medium, and high Misses Per Kilo Instructions (MPKI). Those workloads whose MPKI is lower than 11 are categorized as low MPKI benchmarks, those higher than 32 as high MPKI workloads, and those in between as medium MPKI workloads. Table 5.3

summarizes the workload composition and related characteristics. All reported MPKI are the LLC MPKIs computed per core, and the footprint is calculated by counting the total number of unique pages seen from LLC misses.

5.2.3 Performance

The breakdown of SILC-FM execution time improvement is shown in Figure 5.7. The stack bar begins with the Random scheme as it is the most naive scheme. Then, the performance improvement achieved through each technique is stacked on top. SILC-FM swap shows the performance improvement achieved with a direct-mapped small block scheme when any associativity, locking or bypassing technique is not applied. The system is able to achieve the speedup of 1.55 with only subblock granularity swapping between FM and NM. In high MPKI workloads where more bandwidth demand exists, the swapping alone can significantly alleviate the bandwidth bottleneck by swapping many hot blocks into NM. For that reason, Figure 5.7 shows that high MPKI workloads achieve the overall highest performance improvement. Workloads such as *milc* do not get much benefit from swapping as conflicts constantly swap out recently swapped in subblocks, which in turn shows the need for other features that are incorporated in SILC-FM. Then, the locking feature is added to SILC-FM, which can improve the hot page residency in NM. In this case, not all benchmarks benefit as some benchmarks do not have a significant amount of thrashing or conflicts from the baseline. However, a benchmark such as *xalancbmk* achieves an extra 14% performance improve-

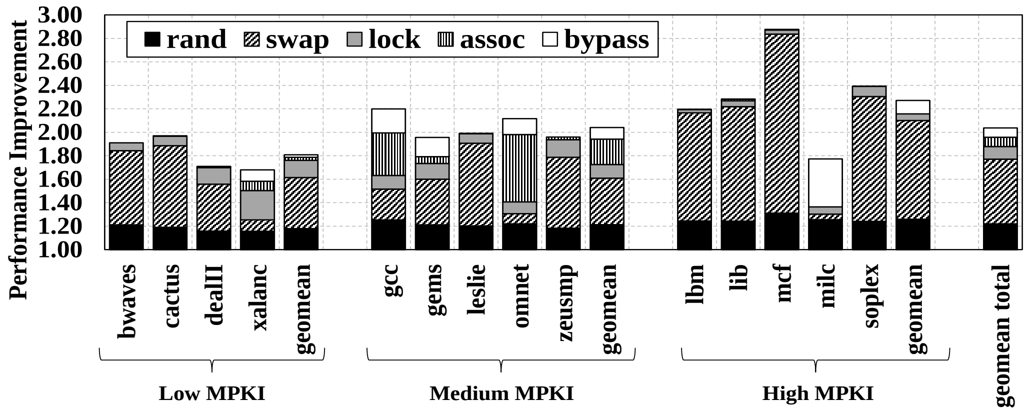


Figure 5.7: SILC-FM Performance Improvement Breakdown

ment just by locking hot pages. The problem of hot blocks being swapped out to FM, if they are not locked, arises due to the fact that address bits are used to place blocks, so not all hot pages are evenly spread out in the NM indexing. The *xalancbmk* benchmark is a good example and locking ensures that some hot blocks are locked, so at least those blocks can be serviced from NM even if not all hot blocks can be accommodated in the NM set.

Now for unlocked blocks, adding associativity achieves similar effects. In some benchmarks, the fraction of hot blocks that reach over the hotness threshold may not be a large portion of the entire working set, meaning many blocks are just lukewarm. In this case, the effects of associativity, which protects those unlocked pages from unwanted conflicts and thrashing, can be quite significant. For example, *gcc* achieves significant speedup of 36% with the addition of associativity while adding locking only improves performance by 11%. This is a good example of a case where the benchmark has many lukewarm blocks. As a result, locking, which only benefits hot blocks, provides negligible improvement, but associativity brings a huge performance improvement. Lastly, the bypassing feature is added in SILC-FM. Note that the bypassing feature is enabled only when the access rate exceeds 0.8. Benchmarks such as *bwaves* do not reach this point, so adding the bypassing feature does not provide additional performance. In contrast, *milc* exceeds the 0.8 access rate, and as a result, the bypassing feature enhances performance by utilizing the FM bandwidth, which otherwise would have been idle. Overall, SILC-FM is able to capture hot blocks and subblocks within lukewarm blocks in NM through features such as swapping, locking, and associativity while higher system-wide bandwidth utilization is achieved through bypassing.

Comparison with Other Schemes Figure 5.8 shows the performance improvement of SILC-FM against other schemes. First, the Random scheme does not see much significant performance improvement. The placement is done randomly without considering NM and FM characteristics, so pages are statically allocated. Although some pages may sit in NM, the access rate is low. Since there are no other overheads due to page migration during the execution, all workloads achieve similar performance improvement.

The HMA and PoM schemes improve upon the Random scheme by intelligently selecting hot pages and placing them in NM. The threshold based decision is able to select a subset of pages (mostly hot pages) and move them to NM. As seen in Figure 5.8, HMA achieves significantly higher performance than Random even though this scheme has additional software overheads such

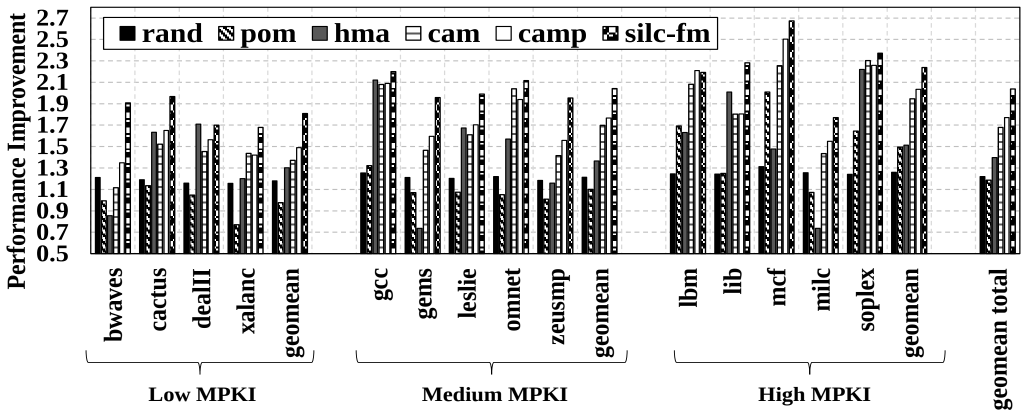


Figure 5.8: SILC-FM Performance Comparison with Other Schemes

as context switching and TLB shootdowns. This makes the majority of hot pages reside in NM. However, the migration only occurs at a very large time interval, so the selected pages may not be hot anymore by the time the decision is made, which is shown in *bwaves* and *milc*. PoM uses a similar scheme as HMA, yet in the evaluation, PoM uses much more FM bandwidth because it frequently transfers 2KB blocks. In high spatial workloads, most of 2KB blocks are used, yet most workloads have a rather low number of unique used subblocks within 2KB. Also, PoM has to accumulate a certain access count before the migration is triggered, so it achieves a lower performance.

The CAMEO scheme reacts quickly to any changes in the hot working set and it moves data at a small block granularity. In all workloads, this scheme effectively places most hot small blocks in NM. Yet, conflict misses are unavoidable since the NM is direct mapped. For example, *cactus* suffers from conflict misses, to the extent that schemes, such as HMA, that can withstand conflicts perform better than CAMEO. In addition, since only one small block is brought into NM at each time, CAMEO does not take advantage of abundant spatial locality within a page. However, the CAMEO scheme's data movement granularity, a small block granularity, uses FM bandwidth efficiently, so it achieves an overall higher performance improvement than other schemes. The improved CAMEO with prefetcher achieves a higher speedup as it enjoys some degree of spatial locality. However, naively prefetching subblocks also wastes bandwidth as those prefetched subblocks are not always useful.

The SILC-FM scheme effectively removes conflicts by offering associativity and locked blocks. Unlike HMA where pages are migrated and locked into NM at epoch boundaries, the blocks are locked in NM as soon as the access count reaches the threshold. This makes the hot block capturing ability of SILC-FM respond quicker to changes in the hot working set. The *gemsFDTD* workload shows performance degradation with HMA, but performance improvement with CAMEO. This benchmark has many short-lived hot pages, and as a result, the epoch length is too long to make smart migration decisions. SILC-FM, on the other hand, responds quickly, so performance benefits are seen. The associativity reduces conflicts among those pages that are not locked yet. For *libquantum*, HMA performs well since it offers fully associative placement at epoch boundaries. Here, CAMEO suffers from conflicts that SILC-FM can withstand by means of locking and associativity. Furthermore, the bit vector based fetching scheme migrates more useful subblocks than CAMEO with a prefetcher, and thus SILC-FM's benefits are greater. This ad-

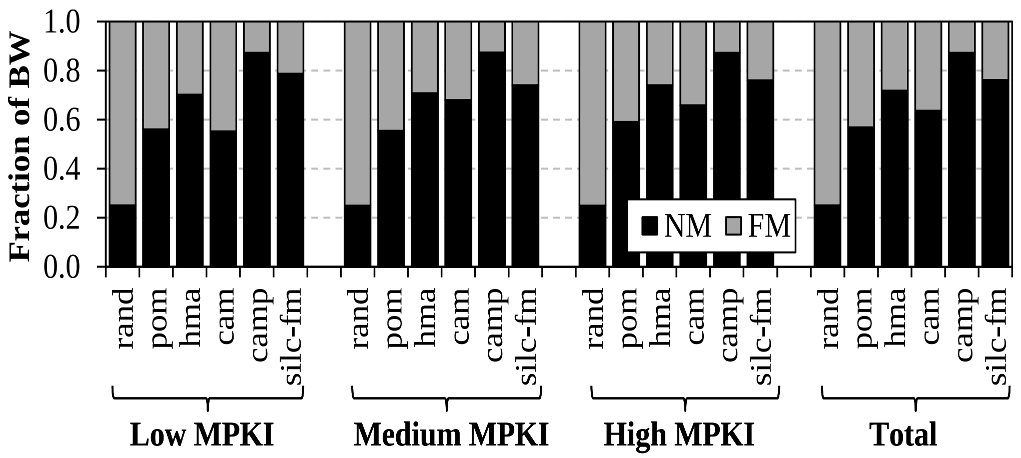


Figure 5.9: Fraction of FM and NM Bandwidth Usage

ditional performance gain makes SILC-FM achieve higher performance than the state-of-the-art scheme, CAMEO. Furthermore, the bypassing feature creates additional performance opportunities for certain workloads such as *milc* by using FM bandwidth, which would have been idle in other schemes due to its high access rate.

Figure 5.9 shows the fraction of the total demanded bandwidth usage broken down by either NM or FM. The ideal point here is 0.8 as discussed in Section 5.1.5. Note that only the bandwidth consumed by demand requests and not by migrations is shown, so the bulk page migration in the HMA scheme is not shown here. In HMA and PoM, 71% and 58% of the total demanded bandwidth usage, respectively, is consumed by NM on average, so the NM's high bandwidth is well utilized. Yet, CAMEO's low access rate makes it service more requests from FM when more of NM's bandwidth could be used. CAMEO with prefetcher adds additional traffic on NM bandwidth as

MPKI	Low	Medium	High	Total
Accuracy	0.59	0.69	0.64	0.64

Table 5.4: SILC-FM Predictor Accuracy

prefetched subblocks consume bandwidth, and thus, it creates an imbalance between FM and NM. Without bypassing, SILC-FM leaves FM bandwidth near idle, but by enabling the bypassing feature, we control the access rate. This makes SILC-FM to have 76% of the total bandwidth usage on NM bandwidth, which is only 4% below the ideal 80%. Therefore, it effectively utilizes available bandwidth in both NM and FM and improves performance.

Prediction Accuracy Schemes with NM metadata such as SILC-FM and CAMEO rely on predictors to reduce the serial latency associated with metadata. The correct predictor speculation on the miss path hides the NM access latency for fetching metadata. The SILC-FM predictor is unique from prior approaches in that it speculates the NM/FM location as well as the way. Since associativity significantly reduces conflicts and thrashing in large block schemes, it is desirable that the associativity is enforced in such designs. Table 5.4 shows the accuracy of our predictor. The speculation is considered correct if it correctly predicts both the NM/FM and the way. If only one is speculated correct, it is not considered as a correct speculation in this table. The average accuracy is 64% with the most accurate being 75% with *soplex*. SILC-FM predicts two outcomes at the same time, thus the accuracy is not as high as in prior work, which predicts either the NM/FM or the way. Yet, it allows the request to completely bypass NM upon a correct outcome. Therefore,

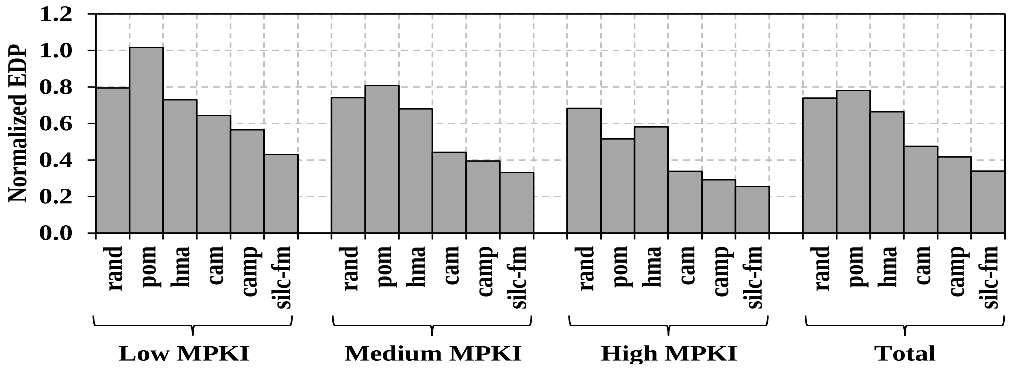


Figure 5.10: SILC-FM Normalized Energy Delay Product

64% of accesses retrieve data while the metadata is being fetching, and even in cases where the NM/FM location prediction is incorrect, the way prediction can save serialized associative remap table fetching latency.

Energy Delay Product Analysis The Energy Delay Product (EDP) of SILC-FM is compared against other schemes. The results include both static and dynamic energy using publicly available data [7, 24, 102, 103]. For the processor side, the McPat [109] tool is used to compute the dynamic and static power of cores and on-chip caches. The Energy Delay Product (EDP) metric quantifies the amount of execution time reduction relative to the amount of increased energy. If EDP is reduced, then performance gain is higher than increased energy consumption. Therefore, a lower EDP value is desired and this section compares SILC-FM's EDP values with other schemes to show whether SILC-FM's performance improvement is worth the cost of increased energy consumption. Figure 5.10 shows the normalized EDP values to the baseline system. Since the execution time dominates the baseline scheme, it

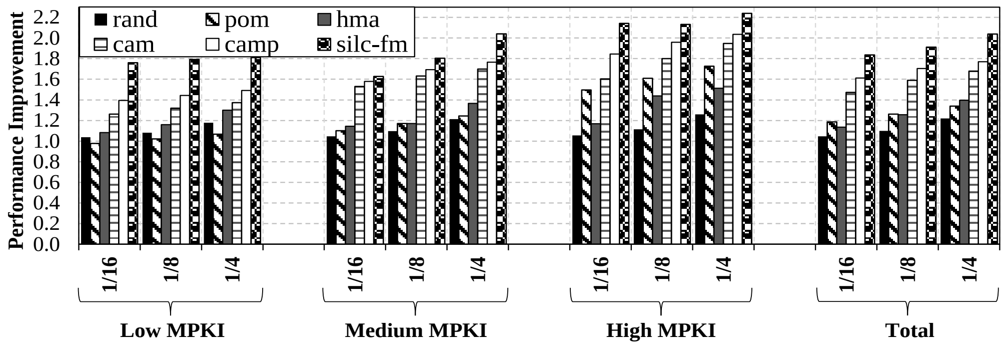


Figure 5.11: SILC-FM Performance Improvement with Various NM Capacities

has the overall highest EDP. Furthermore, SILC-FM is able to offset the energy consumption by having a significant speedup even in low MPKI cases where the processor energy is higher than other schemes (cores are more active since memory is not stalling them frequently). Similarly, CAMEO with prefetching outperforms other schemes such as Random and HMA, which show relatively high EDP values in comparison to SILC-FM as their increase in energy consumption is not compensated by performance improvement. Overall, SILC-FM is able to achieve 66% and 14% improvements in EDP over baseline and CAMEO respectively.

5.2.4 Sensitivity Studies

Capacity Analysis In this section, the performance improvement with different NM to FM capacity ratios is analyzed because the proportion of the NM capacity can range from a small to a large fraction of the overall memory capacity. This study varies the NM:FM capacity ratio from 1/16 to 1/4. In recent Intel Knights Landing processor [8], NM to FM ratio is approximately 1:24, so the capacity close to this ratio is evaluated to see performance effects. Figure 5.11 shows the performance improvement with various schemes and capacities. CAMEO and CAMEOP perform better with a larger capacity since it has larger number of sets, which reduces its inherent problem of conflicts. HMA and PoM's performance is not affected significantly by NM capacities. Although they can capture long-lasting hot pages, short-lived hot pages are not well captured. The set of benchmarks used in this paper does not have the varying number of long-lasting pages with NM capacities, and thus, their performance remains approximately constant with varying NM capacities. However, in other domains of benchmarks such as the cloud domain, it is possible that their performance improvement could be noticeable. SILC-FM, on the other hand, captures hot blocks/subblocks across various capacities. When the capacity is small, such as 1/16, the scheme also has a smaller number of sets like CAMEO; however, locking and associativity significantly prevent the problems with thrashing and conflicts that hurt CAMEO. This is apparent in low MPKI workloads, which does not cause significant performance degradation for SILC-FM even with a much reduced number of sets. Therefore, while conflicts and thrashing play a huge performance role in other schemes such as CAMEO, SILC-FM is able to minimize such an impact when the capacity is reduced. Overall, SILC-FM's average performance improves from 1.83 to 2.04 when the ratio grows from 1/16 to 1/4 while the best comparable scheme only improves from 1.47 to 1.67.

5.3 Summary

Die-stacked DRAM can help to overcome many challenges faced by current memory subsystems. Prior approaches focused on using block-based schemes, but naively adopting such schemes will benefit only a subset of different workloads. This chapter has presented an associative locking memory architecture called SILC-FM that locks hot pages in NM and intelligently remaps FM subblocks into NM. Unlike using NM as a cache, SILC-FM fully exposes the die-stacked DRAM capacity to the OS to take advantage of the additional capacity provided by NM. SILC-FM incorporates a predictor to reduce the access latency. In addition, some memory requests bypass NM and directly access FM to utilize the FM bandwidth available to the processor. In the end, SILC-FM is able to achieve, on average, 36% performance improvement over state-of-the-art die-stacked DRAM architecture. Not only does SILC-FM improve performance, but also it achieves energy efficiency by minimizing the number of accesses to energy-costly FM. In conclusion, SILC-FM is a novel memory architecture that takes the advantage of the large NM capacity by holding a significant amount of data in NM while effectively using the overall system bandwidth.

Chapter 6

Granularity-Aware Page Migration in Heterogeneous Memory System

In a heterogeneous memory system, the efficiency of the data migration mechanism between memories composed of different technologies is a key determinant of the overall system performance. Prior works on migration have focused on what to migrate and when to migrate. This chapter demonstrates the need to investigate another dimension in any migration scheme – *how much to migrate*. Specifically, the amount of data migrated together during a single migration attempt (called "migration granularity"), has a profound impact on the performance of an OS-driven page migration scheme. Through analysis performed on real hardware, it is shown that the best-performing migration granularity is different across applications, owing to their distinct memory access patterns. Since this best-performing migration granularity for a given application may not be known a priori, this dissertation proposes two schemes to dynamically infer the best migration granularity for an application. When these schemes were implemented by modifying the Linux OS running on real x86 hardware, the performance of a heterogeneous memory system improved by 66% on average across 13 benchmarks and up to 74%.

Detailed experiments on real hardware running Linux OS and across a wide range of applications were performed to show that *different applications perform best with different migration granularities*. Specifically, we studied three different migration granularities – 4KB, 64KB and 2MB. Applications like *xsbench* [110], which demonstrate near-random memory access patterns with little spatial locality across multiple pages, perform best with the smallest migration granularity (here, 4KB). Applications with a streaming-like behavior (e.g., *lulesh* [111]) prefer a large migration granularity (here, 2MB) as it helps amortize the costs of migration and implicitly prefetchs useful data to NM. Interestingly, there are applications like graph500 [75], which fall somewhere in the middle; the use of smallest migration granularity adds extra overheads, but if the granularity is too large, then the performance drops due to extraneous migration. This application has some spatial reuse across a few pages, but it is not to the degree of applications with streaming-like behavior.

Unfortunately, which application prefers which migration granularity may not be known a priori. Applications have also been known to show phase behaviors [112]. Therefore, two dynamic schemes are proposed to determine the preferred migration granularity for an application at runtime. In the first scheme, the OS monitors page-grain meta-data (e.g., access bits) to estimate application's spatial access locality. It then uses the largest migration granularity that it estimates will contain *enough* reusable data. This ensures that the overhead of migration is amortized using the largest migration granularity that is beneficial. At the same time, it ensures that the NM's capacity is not wasted due to useless data migration. Then, another independent scheme that uses sampling is proposed. Specifically, it breaks the execution into short sampling phases followed by normal execution phases. During the sampling phase, it performs migration with three different granularities (e.g., 4KB, 64KB and 2MB) and chooses the one that incurred the least overheads. The sampling phase repeats periodically to capture any phase change in application's behavior.

The aforementioned schemes are implemented in the Linux OS. While one possible page migration scheme is chosen to show the importance of selecting the *right* migration granularity, the lessons learned broadly apply to any other page migration scheme. In short, this chapter presents contributions as follows:

- The work demonstrates the need to consider the migration granularity in designing an effective page migration scheme for heterogeneous memory systems.
- By analyzing a wide range of applications on real hardware, it shows that different applications perform best under different page migration granularities.
- Two simple schemes are presented to dynamically select the preferred page migration granularity for a given application.
- The prototype implementation of the above mentioned schemes in Linux

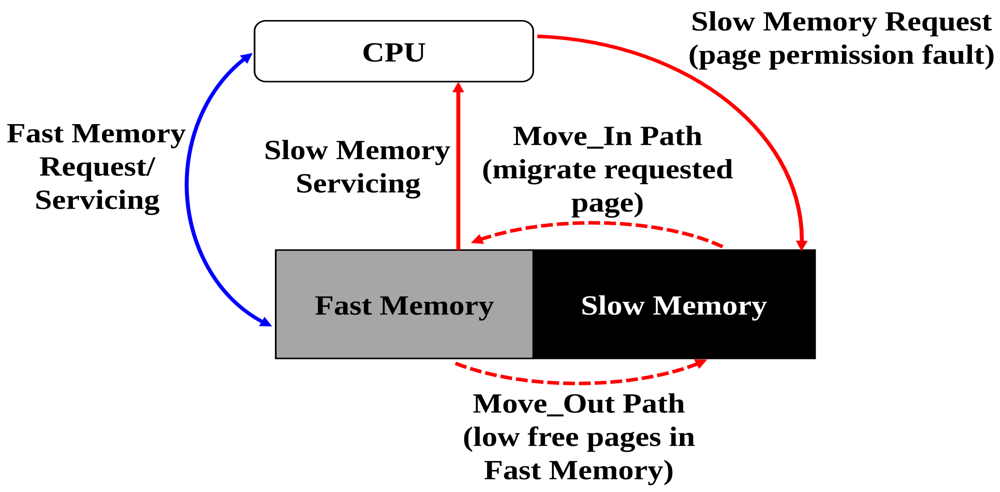


Figure 6.1: Baseline OS Driven Migration Scheme

OS shows that two schemes can improve performance by up to 74% over the state-of-the-art baseline page migration scheme.

6.1 Baseline: OS-Managed Page Migration

The goal is to demonstrate the importance of choosing the *right* migration granularity in an OS-managed page migration scheme. To make the contributions concrete, a state-of-the-art migration scheme is chosen as the baseline scheme. This scheme closely follows the one proposed by Oskin et al. [48] where the OS migrates pages between the FM and the NM, driven by the application's memory accesses. The baseline scheme is built in to the OS-based prototype that runs on a modern hardware and executes unmodified applications.

Figure 6.1 depicts a simplified depiction of the baseline. Both NM and

FM are part of the main memory and are accessible by the OS. An application can directly access data resident in the NM through load/store without OS intervention, but an access to the FM triggers a page permission fault (but no disk access). This helps OS monitor application's accesses to the slow-memory resident data. Initially data is allocated on the NM, and when it is full, then the data is migrated to the FM to make space. This process of evicting data from the fast to FM is called the *move_out* path. As mentioned before, the application's access to the FM triggers a page permission fault, which in turn, triggers the OS to migrate the page with the requested data to the NM. This migration into the NM is referred as the *move_in* path. A page is moved to the NM and the new mapping for this page is used in TLB. This means that no TLBs contain mappings for the pages in FM. During the steady state of an application whose memory footprint exceeds the NM capacity, every move_in will trigger a move_out as the NM would be full. In the baseline scheme, the page migrations happen at a fixed granularity of 4KB (x86-64's default page size). Note that while this migration scheme is chosen as the baseline, the contribution of this work equally applies to other OS-managed migration schemes.

Although it is possible to implement the migration driver in the background to reduce the overhead of invoking an OS when an access is made to FM, it is not considered in this work. First, this work is focused on studying the impact of a dynamic migration granularity, not a migration scheme. Therefore, the work does not modify the original migration scheme proposed in prior state-of-the-art work [48]. Second, when the migration effect is on the critical path as in the presented baseline, the impact of dynamically adjustable migration granularity can be easily shown. Therefore, the original scheme from prior state-of-the-art work is used.

6.2 Dynamically Selecting Migration Granularity

In this section, two independent schemes to select migration granularity at runtime are proposed. While there are a plethora of other alternative mechanisms to dynamically select migration granularity, the broader contribution is to quantitatively demonstrate the need to consider migration granularity as a key design parameter in any OS-directed page migration scheme.

6.2.1 History-Based Migration Scheme

The goal of any scheme for dynamically selecting migration granularity is to make use of largest granularity of migration as long as the migrated data is used *enough* by the application. This helps to amortize the overheads of migration as much as possible without flooding the capacity-constrained NM with useless data. One way to infer if a larger migration granularity would be helpful is to monitor application's spatial access locality across contiguous virtual memory regions (migration granularity). If there is substantial spatial locality within a larger virtual memory region, then the proposed scheme uses a larger migration granularity. For easier explanation of the scheme and how it is prototyped in the Linux OS, the description is organized into two parts –

move-in and *move-out*.

Algorithm 1 History Based Algorithm (Move_In Path)

```
// per application variables
VAR app.curr_granularity//current migration granularity
VAR app.2MB_region, app.64KB_region
procedure Move_In(VA addr)
    UPGRADE_GRANULARITY(addr)
    MIGRATE_TO_NM(addr)
procedure Upgrade_Granularity(VA addr)
    // Decide if larger granualrity is preferrable
    if app.curr_granularity $\neq$ 2MB then
        region := addr >> 21 // 2MB align
        NM_pages := NUM_4KB_IN_NM[region]++
        if NM_pages $\geq$ thr_NM_2MB then
            if (++app.2MB_regions)$\geq$ thr_movein_2MB then
                app.curr_granularity := 2MB
                CLEAR_MOVEIN_STATS()
                return
    if app.curr_granularity = 4KB then
        region := addr >> 16 // 64KB align
        NM_pages := NUM_4KB_IN_NM[region]++
        if NM_pages $\geq$ thr_NM_64KB then
            if (++app.64K_regions)$\geq$ thr_movein_64KB then
                app.curr_granularity := 64KB
                CLEAR_MOVEIN_STATS()
    return
procedure migrate_to_NM(VA addr)
    if app.curr_granularity = 2MB then
        Move 2MB virtually aligned contiguous pages to NM
    else if app.curr_granularity = 64KB then
        Move 64KB virtually aligned contiguous pages to NM
    else if app.curr_granularity = 4KB then
        Move 4KB page to NM
    Unset access_bit in PTE of each migrated pages
```

In the move-in path (Algorithm 1), the scheme decides if an application could have benefited by using a larger migration granularity than it is currently using. For this purpose, the scheme keeps counters (software counters in OS) for the 2MB (64KB) contiguous virtual memory regions of an application to track how many base pages (4KB) within its corresponding 2MB (64KB) region have been moved to the NM. A higher value signifies good spatial usage across the given virtual memory region. Instead of keeping such counters for every such virtual memory region in an application's address space, the scheme employs a hashmap of counters indexed by 2MB (64KB) aligned virtual memory address (*NUM_4KB_IN_NM*). These counters are software counters that are kept internally inside Linux kernel. When one such per-region counter crosses a pre-determined threshold, a *per-application* counter is incremented to track the number of such 2MB (64KB) regions in the application (Line 12-13 and line 20-21). When this per-application counter passes a threshold (*thr_movein_2MB*), then it indicates the existence of *enough* number of 2MB (64KB) regions, which could have benefited from using larger migration granularity (Algorithm 1: Line 13 and 21). It then sets a *per-application* variable that stores the current migration granularity for the application (*app.curr_granularity*) to 2MB (64KB). Although it is possible that the scheme can limit the migration granularity to change only from 2MB to 64KB and 64KB to 4KB, the proposed scheme does not enforce such constraints in order to make the scheme react quickly to changes in memory access patterns.

The next part of the move-in path performs actual data migration from the slow to NM. Here, the history based scheme simply checks the current migration granularity (*app.curr_granularity*). If this is set to migration granularity of 2MB (64KB), then we migrate any FM-resident 4KB pages within its corresponding 2MB (64KB) virtual memory region, at once.

The move-out path (Algorithm 2) is triggered by the need to evict data from the NM to make space for incoming data. This process starts by selecting a victim page (4KB) using Linux's already-existing LRU-based schemes.

Algorithm 2 History Based Algorithm (Move_Out Path)

```
 1: // per application variables
 2: VAR app.curr_granularity//current migration granularity
 3: VAR app.4KB_unuse, app.64KB_unuse
 4: procedure MOVE_OUT()
 5:     if low free pages in NM then
 6:         addr := find LRU page in NM
 7:         if (app.curr_granularity $\neq$ 4KB) then
 8:             DOWNGRADE_GRANULARITY(addr)
 9:         MIGRATE_TO_FM(addr)
10: procedure DOWNGRADE_GRANULARITY(VA addr)
11:      // Decide if smaller granualrity is preferrable
12:     not_accessed = FRAC_NOT_ACCESSED(addr)
13:     if not_accessed $\geq$ thr_NM_4KB then
14:         if (++app.4KB_unuse) $\geq$ thr1_moveout_4KB then
15:             app.curr_granularity := 4KB
16:             CLEAR_MOVEOUT_STATS()
17:     else if app.curr_granularity = 2MB &&
18:                     (not_accessed $\geq$ thr_NM_64KB) then
19:         if (++app.64KB_unuse) $\geq$ thr1_moveout_64KB then
20:             app.curr_granularity := 64KB
21:             CLEAR_MOVEOUT_STATS()
22: procedure MIGRATE_TO_SM(VA addr)
23:     if app.curr_granularity = 2MB then
24:         Move 2MB virtually aligned contiguous pages to FM
25:     else if app.curr_granularity = 64KB then
26:         Move 64KB virtually aligned contiguous pages to FM
27:     else if app.curr_granularity = 4KB then
28:         Move 4KB page to FM
29:     Shootdown TLBs
```

More importantly, the scheme then determines if it should have used a smaller migration granularity when it is currently using 2MB (64KB) migration granularity (Line 7 - 8). For this purpose, the scheme estimates if there are *many* neighboring 4KB (base) pages in the 2MB (64KB) contiguous virtual memory region where the victim pages map to. It does so by taking advantage of x86-64's *access bit* available in each PTE. Every x86-64 processor is guaranteed to set this bit in the PTE on the first access to the corresponding page [113]. This bit is first set to 0 in each PTE of a page that is being migrated from the slow to NM in the move-in path. Later, this bit is checked at the time of page's eviction from the NM. If the access bit is still zero, then it signifies that the content of the page was unused while in the NM. *FRAC_NOT_ACCESSED* function in Algorithm 2 performs the calculation and finds the fraction of the number of such 4KB pages in its 2MB (64KB) aligned virtual memory region. If it is above a pre-defined threshold, then migrating in a larger migration granularity was wasteful (Line 13 and 17). When a *enough* number of such occurrences are found, then the application's current migration granularity is altered (variable *app.curr_granularity*) to 64KB (4KB). Thereafter, both move-in (migration) and move-out (eviction) of data is performed in granularity of 4KB (or 64KB); until the move-in path upgrades the migration granularity again. Also note, irrespective of the data migration granularity, one TLB shootdown is performed for each migration.

6.2.2 Sampling-Based Migration Scheme

The primary contribution in this work is to demonstrate the need to dynamically select the migration granularity and not any specific migration scheme. The above claim is substantiated by proposing another orthogonal dynamic scheme. Furthermore, the history-based scheme is reactive and thus can alter migration granularity only after the *history* has been made. Below, a predictive scheme that does not depend on the past access history is proposed.

The key observation here is that the relative number of TLB shootdowns performed while employing different migration granularities is a good indicator of which granularity is preferred by application's access pattern. For example, if an application has perfect streaming behavior then using 2MB granularity instead of 4KB will decrease the number of TLB shootdowns by 512X. On the other hand, if an application has a complete random access pattern over large memory region, then it is possible that only one 4KB page in every 2MB region is touched while in the NM. In this case, the number of shootdowns with 2MB granularity could be equal to that with 4KB granularity. The 2MB granularity can incur more shootdowns by evicting useful data from the NM, which then needs to be re-migrated. In short, the ratio of the number of TLB shootdowns to different migration granularities is employed to select the preferred migration granularity.

Algorithm 3 Sampling Based Algorithm

```
// per application variables
VAR app.curr_granularity := {4KB, 64KB, 2MB}
procedure SAMPLING PHASE()
    if 4KB sampling phase then
        app.curr_granularity := 4KB
        num_shootdowns := 0
        Enter 64KB sampling phase after 10 ms
    else if 64KB sampling phase then
        4KB_# = num_shootdowns
        app.curr_granularity := 4KB
        num_shootdowns := 0
        Enter 2MB sampling phase after 10 ms
    else if 2MB sampling phase then
        64KB_# = num_shootdowns
        app.curr_granularity := 2MB
        num_shootdowns := 0
        Enter execution phase after 10 ms
    else if execution phase then
        2MB_# = num_shootdowns
        if 4KB_# / 2MB_# $\geq$ thr_prof_2MB then
            app.curr_granularity := 2MB
        else if 4KB_# / 2MB_# $\geq$ thr_prof_64KB then
            app.curr_granularity := 64KB
        else
            app.curr_granularity := 4KB
        Enter 4KB sampling phase phase after 200 ms
```

Algorithm 3 shows the simplified pseudo-code of a sampling based scheme for dynamically selecting the migration granularity that is inspired by the above observation. Specifically, it breaks the execution in two repeating phases – a sampling phase and a normal execution phase, each with a fixed time length. The sampling phase is further divided equally into three parts and three different migration granularities are tried out. At the same time, the number of TLB shootdowns are noted for each of these sub-phases during sampling. At the end of the sampling phase, the schemes calculate the shootdowns seen with different migration granularities and based on predefined thresholds, one of them is selected (Algorithm 3: Lines 20-25). In this work, the predefined threshold is experimentally found. Next, the normal execution period begins and uses the selected migration granularity. This cycle of sampling phase followed by normal execution phase repeats itself throughout the execution. One optimization that can be done is to only sample 2MB and calculate appropriate ratios to select the preferred granularity.

6.3 Prototype and Results

In this section, the evaluation methodology is first discussed and then the modifications to Linux OS for prototyping granularity-aware page migration are discussed. This is followed by a presentation of the detailed evaluation of the prototype.

Processor	Values
Number of Cores	4
Frequency	3.8 GHz
Caches	**Values**
L1 I-Cache	128 KB, 2 way
L1 D-Cache	64 KB, 4 way
L2 Cache	4 MB, 16 way
NM	**Values**
Type	DDR3-1866
Capacity	400 MB
FM	**Values**
Type	PCM (emulated)
Capacity	8 GB

Table 6.1: Real Machine System Parameters

6.3.1 Baseline Setup

The scheme is evaluated on AMD A10-7850K processor [78] running Linux OS (kernel version 3.16.36) [79]. Table 6.1 lists the configuration of the machine we used for evaluation. While systems with heterogeneous memory are imminent (e.g., Intel 3DXpoint [80]), they are not yet available commercially. Thus, the heterogeneous memory system is emulated using Linux's NUMA emulation feature [79]. This feature allows us to divide the aggregate physical memory available in the system (32GB) in equally sized contiguous physical memory zones. Pages can then be migrated across these zones. Specifically, four physical memory zones with 8GB each are created. This configuration allowed each physical memory zone to be mapped on to separate DIMMs since our test machine have 4 DIMMs. Of these, one of the zones acts as the NM and another as the FM. It is important that the memory footprint of an application surpasses the capacity of the NM in order to observe a meaningful number of page migrations. Another Linux feature called Memory HotPlug [81] allows part of physical memory to be *offlined* as if it does not exist. The effective (usable) capacity of the NM is reduced to 400MB and FM to 8GB in the baseline in order to emulate the real world scenario where the NM is a fraction of the application's memory footprint. Although a recent driver such as the one offered to Intel's Knights Landing [114] provides a way to migrate data, the experimental platform used in this dissertation does not use it as the modification is purely done inside official Linux kernel source code.

Next, the longer access latency of the FM is emulated by modifying the Linux kernel. In the baseline, accesses to the FM generate a page permission fault to trigger the migration to the NM. This is ensured by assigning *NULL* page permission to PTE entries for any page mapped to the FM. An additional latency is added in addition to the page permission fault latency for migrating data from the slow to NM such that an access latency to the FM is roughly 3X that of the NM [115, 116]. Since any migration uses the Linux page fault handler function, the delay is added inside the function call. Furthermore, the swapping functionality of Linux is disabled to avoid any interference. It ensured that the memory footprint of applications fits in the combined capacity of the fast and FM (400MB + 8GB). The scheme presented in this chapter does not exploit the bypassing feature presented in the previous chapter, so the performance shown in this chapter is an underestimate. Table 6.2 shows the description of workloads used in this chapter. Note that the Resident Set Size (RSS) is larger than the NM capacity (400MB). This indicates that having the NM itself is not sufficient to cover the memory footprint, so the migration is necessary in all workloads.

6.3.2 Granularity-Aware Page Migration

To model the proposed schemes, two tasks are required: ① enable migrations with varying granularity, ② implement dynamic schemes to select the preferred migration granularity for a given application during runtime. These goals are accomplished by modifying the Linux kernel and by adding a new OS driver. Specifically, a new functionality is added to the Linux kernel that can migrate a contiguous chunk of memory in an application's virtual address space between the fast and FM zones. This functionality is then invoked by the move-in and move-out path by specifying the desired migration granularity (implemented in the OS driver to minimize kernel changes). Currently, three migration granularities are supported – 4KB, 64KB and 2MB. However, the mechanism can be easily extended to other granularities that are any multiples of the base page size (4KB). Each migration issues one TLB shootdown, irrespective of its granularity. However, each shootdown routine is slightly different based on migration granularity. For 4KB, the shootdown routine executes x86-64's *invlpg* instruction to invalidate the entry for that given page in local TLBs of each (receiving) core. In x86, the *invlpg* instruction invalidates a single TLB entry across all cores. For 64KB, the shootdown routine at each core executes 16 invlpg instructions to invalidate all sixteen 4KB pages that that 64KB region maps to. For 2MB pages, instead of looping over 512 pages, it writes to local core's *cr3* register. Writing to *cr3* flushes all entries belonging to user applications from the local TLBs [113].

Next, a new OS driver is inserted to implement dynamic schemes to find the preferred migration granularity. The driver monitors the access patterns of applications, collects the number of TLB shootdowns, and alters the page migration granularity by setting an internal variable that encodes preferred migration granularity when needed (algorithm in previous Section). For the history-based scheme, the default threshold value of 0.85 for both 2MB and 64KB (e.g., 85% of 2MB region is accessed) is used to determine whether to upgrade/downgrade the migration granularity. For the profile-based scheme, the threshold of 2X is used as the default ratio of the number of TLB shootdowns between 4KB and 2MB to decide the migration granularity. The threshold values are experimentally found and the sensitivity study is conducted later in this chapter.

6.3.3 Workloads

The migration schemes are evaluated using a wide range of workloads drawn from PARSEC [74], NAS Parallel benchmarks [83], HPC Challenge [84], Mantevo [85], CORAL [110, 111], and graph500 [75]. Table 6.2 lists the individual applications used from these benchmark suites. The largest input is used to make sure it is run out of NM. The table also lists the working/resident

Notation	Suite	Workloads	Input	RSS
g500	-	graph500	-	761MB
gups	HPC	gups	-	2.00GB
str	HPC	stream	-	1.08GB
cg	NAS	cg	C	890MB
ft	NAS	ft	B	1.26GB
is	NAS	is	C	1.03GB
mg	NAS	mg	B	490MB
ua	NAS	ua	C	483MB
can	PARSEC	canneal	native	939MB
freq	PARSEC	freqmine	native	678MB
lul	CORAL	lulesh	-	696MB
mini	Mantevo	minife	-	642MB
xs	CORAL	xsbench	large	5.55GB

Table 6.2: Real Machine Experiment Workload Descriptions

set size (RSS) of each application as reported by Linux. This will help reader to understand the memory footprint of each application in relation to the emulated the NM capacity. All of the application are multi-threaded and made to use at least 8 threads. Workloads from PARSEC use pthread library while all others use OpenMP for multi-threading.

6.3.4 Performance

Figure 6.2 shows the normalized execution time (lower is better) of the baseline migration scheme under three different statically fixed migration granularities and also of history-based and sampling-based dynamic schemes that adjust the preferred migration granularity at runtime. The height of each bar is normalized to the execution time of a given application running on a

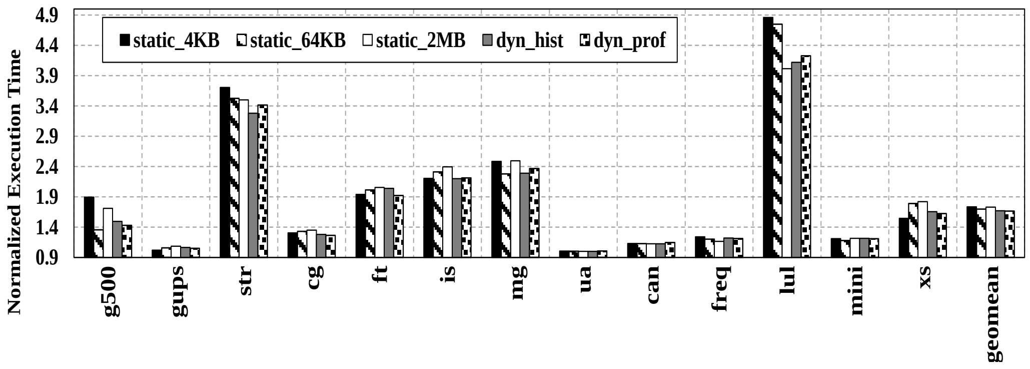


Figure 6.2: Dynamic Granularity Scheme Execution Time Normalized to Baseline (All NM)

system where the NM capacity exceeds application's memory footprint and thus, incurs *zero migrations*. Each application has five bars – the first three bars show the normalized execution time when the migration granularity is statically fixed to 4KB, 64KB and 2MB. The last two bars show the same normalized execution time when the proposed history-based and sampling based dynamic schemes are employed, respectively.

First, accessing the FM and migrating pages can significantly slow down an application, but its impact varies across workloads (e.g., *lulesh* is severely affected by migration while *ua* is barely impacted). Second, five out of thirteen applications studied (*gups, cg, ft, is*, and *xsbench*) perform better when migration granularity is statically set to 4KB. Three applications (*graph500, mg*, and *minife*) prefer 64KB static migration granularity. Three applications (*stream, luesh*, and *freqmine*) prefer 2MB static migration granularity. Applications *ua* and *canneal* are insensitive to migration granularity as they do not have a significant number of migrations. This quantitatively establishes that a migration granularity is an important aspect to be considered in designing any migration scheme.

The history-based (indicated in figures as dyn_hist) and sampling-based (indicated in figures as dyn_profiling) schemes for dynamically selecting migration granularity yield execution times close to that of the best-performing static migration granularity for almost all applications. For three applications (*stream, is*, and *cg*), the history-based dynamic scheme performs better than their respective best static migration granularity. The history-based scheme was able to achieve this by using different migration granularities at different times during an application execution. Furthermore, for *cg, is* and *ft*, the sampling-based dynamic scheme also performed better than any static migration granularity. This was possible since, unlike the static schemes, the sampling-based scheme was able to react to application's phase behavior. In some cases, like *lulesh* and *graph500*, dynamic schemes could not completely match the performance of the best static migration granularity. There, the history-based scheme is too slow to use a larger migration granularity while the sampling-based scheme's sampling period fell in non-representative phase of the applications.

Figure 6.3 further analyzes the performance of dynamic schemes by showing the breakdown of migrations for each of the three possible granularities. For each application, there are two stacked bars – one for history-based scheme and one for the sampling-based scheme. Each stack bar shows

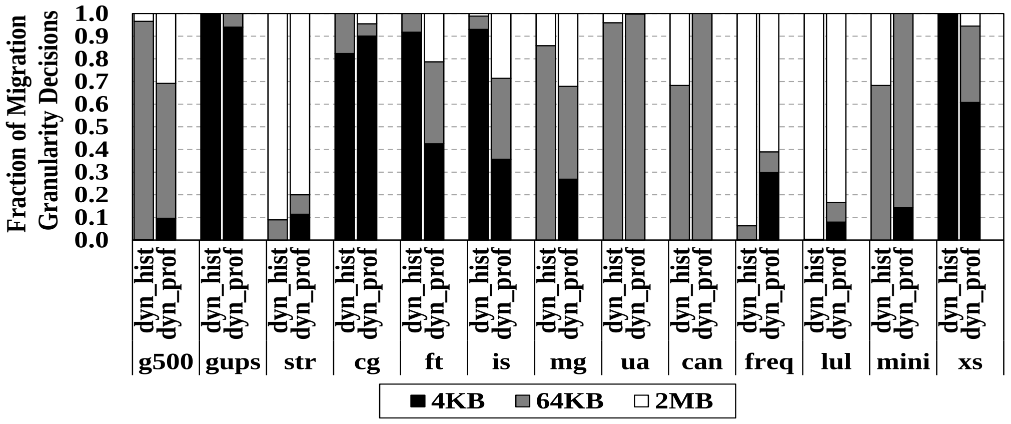


Figure 6.3: Dynamic Granularity Scheme Breakdown of Total Granularity Decisions

the fraction of migrations with 4KB, 64KB or 2MB granularity. Both dynamic schemes are able to pick the migration granularity that matches the best-performing static migration granularity for an application. For example, *graph500* is known to prefer 64KB migration granularity, and the breakdown in Figure 6.3 shows that both dynamic schemes picked that correctly. This explains why the dynamic schemes are able to keep up with best-performing static ones. Both dynamic schemes used a mix of migration granularities. The history-based scheme was able to use different migration granularities for different memory regions and at different times in the application's execution while the sampling-based scheme did so by using different granularities in different sampling-execution phases. Although the decisions were very accurate in some workloads such as *graph500*, *ua*, *xs*, and *gups*, their performance is not the same as the best static granularity case. Since the history based scheme incurs some overheads such as looping through virtual addresses and

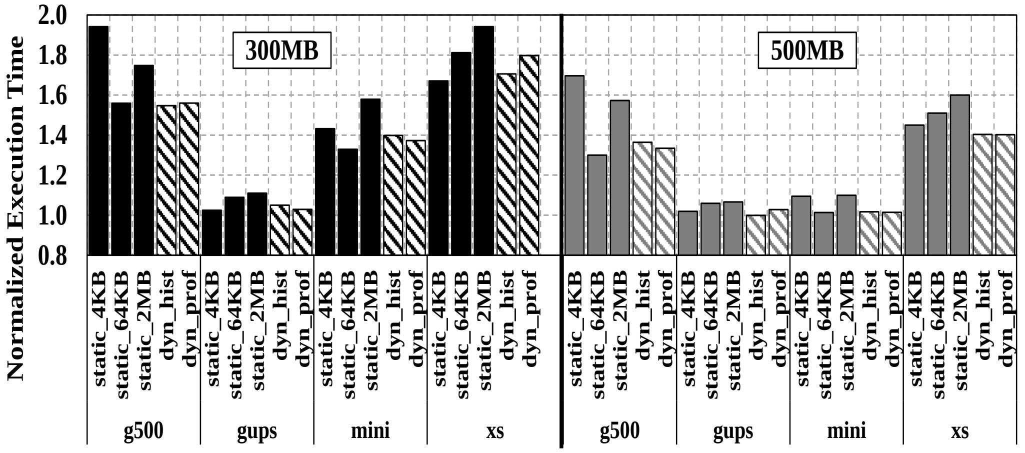


Figure 6.4: Dynamic Granularity Scheme Execution Time Normalized to Baseline (All NM) with Varying NM Capacities

translating them to appropriate physical addresses, the performance does not completely match the best static performance. One improvement to reduce such overhead is to sample a few pages rather than looping through the entire 2MB (64KB) region. Despite this overhead, the overall performance is higher than the performance of any single static granularity.

6.3.5 Sensitivity Studies

In this section, the sensitivity of the dynamic migration granularity schemes to various parameters is presented.

Capacity Sensitivity Results presented till now are measured with the NM capacity of 400MB. This size is chosen such that considerable parts of every applications' working set exceeds the NM capacity. Otherwise, it would not incur any meaningful number of migrations to study. However, to analyze the robustness of the analysis and proposals, this section evaluates both static migration granularities and dynamic schemes with the NM capacity of 300MB and 500MB. The capacity of the NM visible to the software is changed using Linux's Hotplug [81].

Figure 6.4 shows the normalized execution time of four representative applications with altered NM capacity. As earlier, the height of each bar is normalized to the execution time with zero migrations. The left half of the figure shows measurements with 300MB NM capacity and the right half shows that with 500MB NM capacity. For each representative application, there are five bars as in Figure 6.2. Three important observations are made from the figure. Firstly, the overhead of migration decreases across the board with larger (500MB) NM capacity since a larger part of an application's working set is held in the NM. Secondly, the best performing static migration granularity remains the same for a given application across varying NM capacities. For example, the migration granularity of 64KB is preferred by graph500 and 4KB is preferred by xsbench across all three NM capacities we evaluated. This is expected as the preference towards a particular migration granularity depends upon an application's memory access pattern. Thirdly, across different NM capacities, the dynamic schemes perform close to the best static granularity for a given application. This demonstrates the robustness of the proposed schemes.

Latency Sensitivity Until now, a particular heterogeneous memory configuration is assumed where the FM's access latency is 3X more than that of the

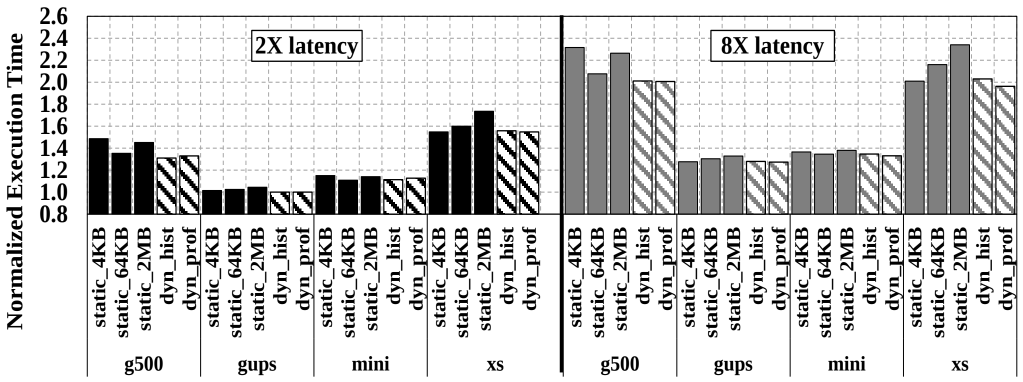


Figure 6.5: Dynamic Granularity Scheme Execution Time Normalized to Baseline (All NM) with Varying NM Latencies

NM. However, a heterogeneous memory can be formed with different memory technologies, which might have different NM and FM latency ratios. Here, this is emulated by varying the FM to NM latency to 2X and 8X (the bandwidth is not adjusted).

As in capacity sensitivity, Figure 6.5 shows the normalized execution time of four representative applications with different FM latencies. The left half presents results for the FM latency being 2X that of the NM, thereby emulating a faster FM scenario whereas 8X represents a case for a much slower FM scenario. Regardless of the latency ratios, dynamic schemes achieved the execution times close to or better than the best performing static cases. Intuitively, this result is expected as our dynamic migration schemes only depend on application's memory access patterns, which does not have any correlation with memory latencies. Finally, even though the latency quadruples between 2X and 8X, the efficacy of the dynamic schemes is not impacted significantly.

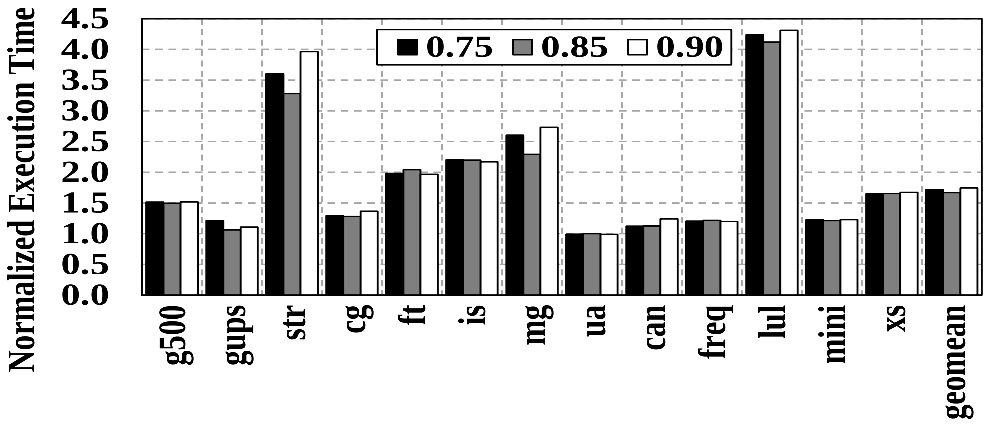


Figure 6.6: Dynamic Granularity Scheme Execution Time Normalized to Baseline (All NM) with Varying Threshold of Accessed Pages within 2MB/64KB Region

Again, the scheme amortizes such migration *overheads* (and not page copy latency) which are unrelated to memory latencies. Therefore, the improvements achieved by the proposed schemes remain effective across varying FM latencies.

Threshold Sensitivity Both dynamic schemes for selecting migration granularity employ several thresholds. In results presented this far, the best performing thresholds are empirically used. In this sensitivity study, however, the values of some of those thresholds are varied to show how these schemes react. For a history-based scheme, the threshold that decides if a larger migration granularity has enough spatial locality is changed. Specifically, the percentage of 4KB pages within a 2MB or within a 64KB migration granularity is set to be 75%, 85% and 90%, to be deemed to have *enough* spatial locality. The normalized execution times for each application with these thresholds are presented in Figure 6.6. The history-based scheme performs best with this threshold set to 85% and thus, this value is typically used in the evaluation. If the threshold is set to 75% then the scheme becomes too aggressive and wastes the NM capacity. The threshold at 95% makes it too conservative where it loses opportunities to use larger migration granularities. Workloads such as *streamcluster* and *lulesh* are sensitive to these thresholds as their streaming memory behaviors make 85% to be an ideal ratio. The software counter used for TLB shootdowns is the number of shootdown requests from remote nodes (near memory), so setting the threshold to 90% makes the scheme rarely change the migration granularity. The performance difference between 75% and 85% is due to the sampling that the sampled execution phases show a better correlation with 85%. If a better sampling mechanism is presented, then the performance gap between 75% and 85% in these workloads would be small. With the sampling period used in this work, the 85% TLB shootdown ratio between 4KB and 2MB migration granularity seems to be a good indicator of high and low spatial locality. If the sample period changes, this ratio might be different as high spatial locality workloads will even have much fewer number of TLB shootdowns with larger migration granularity.

Likewise, the sampling-based scheme also uses a threshold, which is the ratio of TLB shootdowns between different granularities to decide which granularity is likely to incur least overhead. The sensitivity to this threshold is presented in Figure 6.7 where different ratios of 4KB and 2MB TLB shootdowns are used to make the granularity decisions. In this figure, 4X means

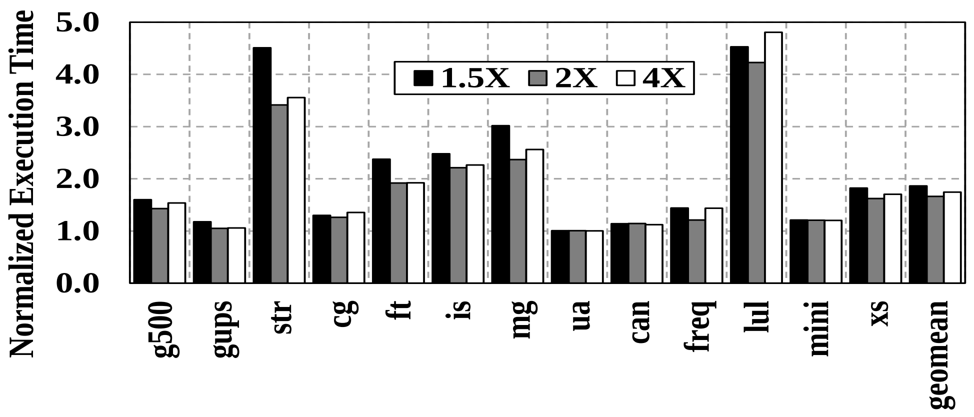


Figure 6.7: Dynamic Granularity Scheme Execution Time Normalized to Baseline (All NM) with Varying Latency Ratio

that the number of shootdowns between 4KB and 2MB must be greater than 4X in order to use a 2MB migration granularity. Making the ratio larger is a conservative approach that 4KB should cause significantly more TLB shootdowns than 2MB in order to use 2MB. Therefore, workloads such as *mg* do not achieve the best performance in case of 4X as its degree of spatial locality is somewhere between 64KB and 2MB (closer to 2MB), yet the conservative threshold always makes such workloads choose the minimum possible granularity, which is 64KB in this case. Therefore, carefully choosing the threshold in both dynamic schemes is also important.

6.4 Summary

This chapter has demonstrated that data migration schemes in heterogeneous memory systems must take another dimension into account, *migra-*

tion granularity. Two dynamic migration schemes are proposed that can detect spatial locality and dynamically change the migration granularity. These schemes achieve an overall system performance improvement up to 74% over the state-of-the-art baseline scheme that uses a single migration granularity. The implementation is only a prototype, and not only limited to the presented baseline scheme, but rather it can extend to any migration schemes. Therefore, the study has identified the migration granularity as an important design parameter.

Chapter 7

POM-TLB: Very Large Part-of-Memory TLB[4]

With increasing deployment of virtual machines for cloud services and server applications, memory address translation overheads in virtualized environments have received great attention. In the radix-4 type of page tables used in x86 architectures, a TLB-miss necessitates up to 24 memory references for one guest to host translation. While dedicated page walk caches and such recent enhancements eliminate many of these memory references, the measurements on the Intel Skylake processors indicate that many programs in virtualized mode of execution still spend hundreds of cycles for translations that do not hit in the TLBs.

This chapter presents an innovative scheme to reduce the cost of address translations by using a very large Translation Lookaside Buffer that is part of memory, the POM-TLB [117]. In the POM-TLB, only one access is required instead of up to 24 accesses as required in commonly used 2D walks with radix-4 page tables. Even if many of the 24 accesses may hit in the page walk caches,

[4]J. Ryoo, N. Gulur, S. Song, and L. K. John, “Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB”, in *Proceedings of the 44th International Symposium on Computer Architecture*, 2017. Jee Ho Ryoo and Nagendra Gulur equally contributed to the idea development and implementations. Shuang Song ran machine-measured experiments. Lizy K. John supervised the overall project.

the aggregated cost of the many hits plus the overhead of occasional misses from page walk caches still exceeds the cost of one access to the POM-TLB. Since the POM-TLB is part of the memory space, TLB entries (as opposed to multiple page table entries) can be cached in large L2 and L3 data caches, yielding significant benefits.

Through detailed evaluations running SPEC, PARSEC and graph workloads, the proposed POM-TLB improves performance by approximately 10% on average. The improvement is more than 16% for 5 of the benchmarks. It is further seen that a POM-TLB of 16MB size can eliminate nearly all TLB misses in 8-core systems. This chapter makes the following contributions:

- Slow memory structures like DRAM can be used to house a large capacity TLB that can hold nearly all required address translations. On average, the proposed POM-TLB can achieve 10% performance improvement over a baseline system. For 5 of the benchmarks, the speedup is 16% or higher.
- A mechanism is presented that makes it possible to cache TLB entries (not page table entries) into data caches. To the best of knowledge, no prior work proposed caching of TLB entries into general (non-dedicated) caching structures.
- Several solutions are presented to the challenges encountered while implementing a TLB in DRAM. Those include a low overhead TLB location predictor and other enhancements to make a DRAM-based POM-TLB a feasible option.

7.1 POM-TLB: A Very Large L3 TLB

In this section, the overall operation of POM-TLB is described where it is implemented in off-chip memory or die-stacked DRAM. Implementing in emerging die-stack DRAM gives some advantages although conceptually that is not a requirement.

7.1.1 System Level Organization

Most modern processors have private multi-level TLBs. This section uses an assumption that the TLB organization is similar to Intel Skylake architecture [28] where there are two levels of TLBs. A large shared L3 TLB is added after the private L2 TLBs. Conceptually, L2 TLB misses look up the large shared TLB and initiate a page walk if this shared TLB also suffered a miss. In practice, since DRAM look-up is slow, the POM-TLB is made addressable thereby enabling caching of TLB entries in data caches and faster translations.

7.1.1.1 POM-TLB Organization

While conceptually not a requirement, implementing POM-TLB in emerging die-stacked DRAMs integrated onto the processor gives bandwidth and possibly small latency advantages, but is challenging. Figure 7.1 shows a single channel of a typical die-stacked DRAM with multiple banks. The detailed layout of a single row in a bank is shown. Each row can house multiple TLB entries as the row size is 2KB. Each entry has a valid bit, process ID,

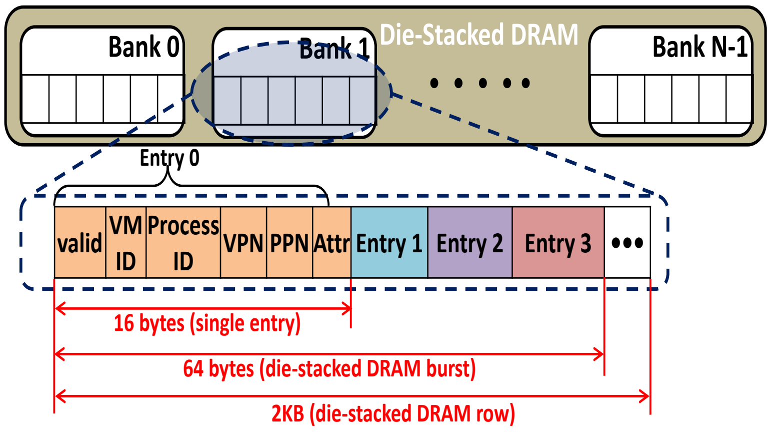


Figure 7.1: POM-TLB Die-Stacked DRAM Row Organization

Virtual Address (VA), and Physical Address (PA) as in on-chip TLBs. To facilitate the translation in virtualized platforms, Virtual Machine (VM) ID is supplied to distinguish addresses coming from different virtual machines as in Intel's Virtual Process ID (VPID) [26]. The attributes include information such as replacement and protection bits. Each entry is 16B and four entries make 64B. The TLB uses a four way associative structure since 1) the associativity lower than four invokes significantly higher conflict misses and 2) 64B is the common die-stacked DRAM burst length where no memory controller design modifications are necessary. Upon a request, four entries are fetched from a single die-stacked DRAM row. Each row can incorporate 128 TLB entries and with 4 way associativity, a row can hold 32 sets of TLB entries.

7.1.1.2 Support for Two Page Sizes

In order to support both small page (4KB) and large page (2MB) TLB entries, and to avoid complexity in addressing a single TLB structure with two page sizes, the TLBs are partitioned into two; one dedicated to hold 4KB page entries (denoted POM_TLB_{Small}) and the other dedicated to hold 2MB page entries (denoted POM_TLB_{Large}). In this implementation, their sizes are statically set and remain fixed. As they are DRAM-based and can afford large capacities, it is observed that their exact sizes do not matter much.

The implementation incorporates a page size predictor to minimize having to perform two DRAM look-ups for the two page sizes. Based on the predicted page size, the corresponding TLB is accessed first. If it is a miss, the other TLB is accessed next. The page size predictor is highly accurate, so the address lookup almost always requires just a single DRAM access.

7.1.1.3 Caching TLB Entries

While L1 and L2 TLBs are designed for fast look-up, the POM-TLB is designed for very large reach, and consequently, its DRAM-based implementation incurs higher access latency. In order to alleviate this, the POM-TLB is mapped into the physical address space. Making the TLB addressable achieves the important benefit of enabling the *caching* of TLB entries in data caches. Although today's system caches PTEs, making the TLB cacheable significantly reduces the space occupied as only single TLB entry is needed instead of 24 in the worst case. Both POM_TLB_{Small} and POM_TLB_{Large} are assigned address ranges. A POM-TLB comprising N sets is assigned an address range of $64 \times N$ bytes as each set holds four 16-byte TLB entries. The virtual address (VA) of the L2 TLB miss is converted to a POM-TLB set index by extracting $log_2(N)$ bits of the VA (after XOR-ing them with the VM ID bits to distribute the set-mapping evenly). For the POM_TLB_{Small}, the memory address of the set that the VA maps to is given by:

$$\begin{aligned} \text{Addr}_{\text{POM\_TLB\_Small}}(VA) = & \\ & ((VA \oplus VM\_ID) >> 6) \\ & (1 << log_2(N) - 1)) * 64 + \\ & \text{Base\_Addr}_{\text{POM\_TLB\_Small}} \end{aligned} \tag{7.1}$$

where $Base\_Addr_{POM\_TLB\_Small}$ is the starting address of the POM_TLB_{Small}. POM_TLB_{Large} addresses are computed similarly.

In this scheme, L2 TLB misses do not initiate page walks. Instead, for each L2 TLB miss, the MMU computes the POM-TLB (say POM_TLB_{Large}) set address where the TLB entry for the virtual address of the miss may be found. The MMU then issues a load request to the L2D$ with this address. At this point, this becomes a normal cache access. If the entry is found in the L2D$, then the MMU reads the L2D$ cache block (64B) to access all the 4 translation entries stored in it. It performs associative search of the 4 entries to find a match for the incoming virtual address. If a match is found, then the corresponding entry provides the translation for this address. Being a normal read access, if the L2D$ does not contain the POM_TLB_{Large} address, then the request is issued to the L3D\$. If no match was found in the L3D\$, then the physical memory (in this case a POM_TLB_{Large} location) is accessed. Associative search of the set stored in the POM_TLB_{Large} is used to identify if a translation of the virtual address is present or not. Like data misses, TLB entries that are misses in data caches are filled into the caches after resolving them at the POM-TLB or via page walks.

Since the POM-TLB provides two potential set locations where the translation for a given VA may be found (POM_TLB_{Small} and POM_TLB_{Large}), it has to perform two cache look-ups starting with the L2D\$. Assuming an equal number of accesses to 4KB and 2MB pages, this results in 50% additional TLB look-up accesses into the L2D\$. This has both latency and power implications. In order to address this, a simple yet highly accurate *Page Size Predictor* is designed whose implementation is described next.

7.1.1.4 Page Size Prediction

A simple yet highly effective page size predictor comprises 512 2-bit entries, with one of the bits used to predict the page size and the other bit used to predict whether to bypass the caches. The predictor is indexed using 9 bits of the virtual address of the L2 TLB miss (ignoring the lower order 12 bits). If the predicted page size is incorrect (0 means 4KB, 1 means 2MB), then the prediction entry for the index is updated. While consuming very little SRAM storage (128 bytes per core), it achieves very high accuracy.

7.1.1.5 Cache Bypass Prediction

In workloads where the data load/store access rate to the data caches far exceeds the rate of L2 TLB misses, the caches tend to contain very few POM-TLB entries since they get evicted to make room to fill in data misses. In such a scenario, looking up the data caches before reaching the POM-TLB is wasteful in terms of both power and latency. Thus, a 1-bit bypass predictor is incorporated to bypass the caches. The predictor implementation is shared with the page size predictor described before.

7.1.1.6 Putting It All Together

POM-TLB relies on making the large TLB addressable to achieve both a high hit rate in data caches as well as lower the access latency. Figure 7.2 shows an overall flow of POM-TLB. L2 TLB misses start out by consulting the page size predictor. If the predictor indicates a cache bypass, then the MMU directly accesses the predicted POM-TLB depending on the predicted page size. If a translation entry is found, the PFN (Physical Page Frame) is returned. If it is predicted not bypass, the MMU checks the POM-TLB entries in data caches. In the common case, the predictor does not predict cache-bypassing. The L2D$ is first probed with the address of the predicted POM-TLB set location. If a match is found with correct (VA, VM_ID), then the translation is done with a single cache access by returning the corresponding PFN. If no match is found, then the MMU probes the L3D$ similarly. If the probed POM-TLB address misses in both L2D$ and L3D$, then the MMU

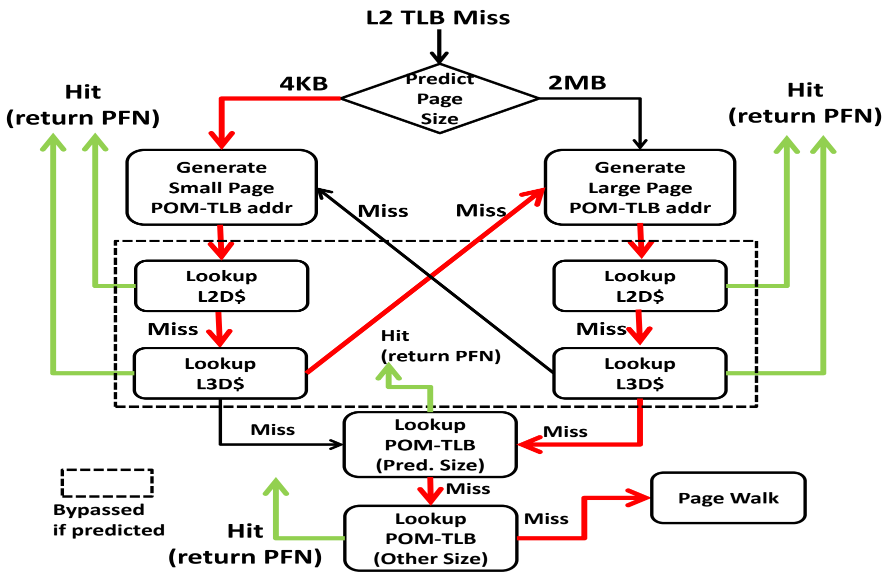


Figure 7.2: POM-TLB Architecture Overview

computes a new POM-TLB address (corresponding to the size that was not predicted) and initiates cache look-up. If the new POM-TLB address misses, then a page walk is initiated. In practice, it is observed that a vast majority of POM-TLB entries hit in the L2D$ and L3D$, resulting in very few DRAM accesses.

7.1.2 Implementation Considerations

This subsection explains a few key design decisions of the POM-TLB.

Consistency: Since POM-TLB is shared across cores, the consistency requirement between entries in POM-TLB and underlying L1/L2 TLB has to be met. Although a strictly inclusive POM-TLB is desirable, it adds significant hardware complexity. Since the TLB operates at DRAM latency, which is already much slower than on-chip SRAM TLBs, adding a structure supporting a strictly inclusive design is not a practical option. Similar to prior work [31], POM-TLB adopts the mostly inclusive implementation, which is adopted in x86 caches [118]. In this design, each TLB can make independent replacement decisions, which makes it possible that some entries in L1/L2 TLBs are missing from POM-TLB. However, this significantly reduces the hardware overheads associated with keeping strictly inclusive. Therefore, POM-TLB is designed to be aware of TLB-shootdowns. TLB-shootdowns require that all corresponding TLBs are locked until the consistency issue is resolved. Yet, TLB-shootdowns are rare occurrences and recent work [48] has shown a shootdown mechanism that can significantly reduce the overhead. Thus, the benefits of having simpler consistency check hardware outweighs the shootdown overheads, and hence, such a design can be adopted. Since the infrastructure does not model TLB shootdowns, the detailed shootdown overhead is not included in this dissertation.

In addition, the consistency across different virtual machines is already handled by modern virtual machine managers such as KVM hypervisor [119]. Upon a change in the TLB, a memory notifier is called to let the host system know that a guest TLB has been updated. Then, the host OS invalidates all related TLBs in other VMs. Therefore, issues such as dirty page handling, process ID recycling, etc. are already incorporated in KVM and host OS. The recent adoption of VM ID facilitates this process, and thus, POM-TLB can maintain consistency in the presence of multiple virtual machines. Although not all virtual machine managers have such feature, since such a feature is implemented in software, future virtual machines can incorporate it.

Channel Contention: Memory systems share a common command/data bus to exchange data between controllers and multiple banks. Many of today's applications experience memory contention as the bandwidth is either saturated or near saturation [120]. Implementing the POM-TLB in an integrated die-stacked DRAM offers advantages from this perspective. The proposed scheme adds additional traffic only to the integrated DRAM to retrieve translation entries and not to the off-chip DRAM. Also, this additional traffic is minor and only incurred when the L2D$ and L3D$ return cache misses when probed for cached POM-TLB entries. The path from last level caches to die-stacked DRAM architecture is different from one to off-chip DRAM as it has its own dedicated high-speed bus to communicate with processors. Hence, additional traffic due to POM-TLB does not interfere with existing main memory traffic. In fact, POM-TLB's high hit rate reduces a significant amount of page table walks that result in main memory accesses, so it is likely that the main memory traffic sees considerable performance benefits as well.

Entry Replacement: Since POM-TLB is four way associative, the attribute metadata (annotated as *attr* in Figure 7.1) contains 2 LRU bits. These bits are updated upon each POM-TLB access and the appropriate eviction candidate is chosen using these bits. Since LRU bits of four entries are fetched in a DRAM burst, the replacement decision can be made without incurring additional die-stacked DRAM accesses. This replacement policy requires a read-write-modify (RMW) operation as the LRU bit has to be updated on every access. Since page walks are not bandwidth bound, the additional bandwidth is not modeled as the increased bandwidth due to page walks is negligible. However, the latency is modeled in the simulation. An optimization can be made where a random entry replacement is performed to eliminate the RMW operation.

Other Die-Stacked DRAM Use: Die-stacked DRAM capacity is growing to multi-gigabytes, and POM-TLB achieves good performance at capacities like 16MB. The remaining die-stacked DRAM capacity can be used as a large last level data cache or a part of memory as proposed by prior work [16–18, 20, 22, 39, 41, 42, 44, 45, 121–123]. Since the HBM JEDEC standard incorporates multiple channels [6], this dissertation assumes using only one dedicated channel to service the POM-TLB requests. When the large die-stacked DRAM is used as both a large TLB and a large last level cache, the performance improvement will be even higher than the results shown in this dissertation, which only presents performance improvement from address translation.

Assuming 16MB of POM-TLB, there can be a tradeoff between using this additional capacity as L4 data cache vs POM-TLB. In a cache design, a hit saves one memory access. However, in the case of an POM-TLB, especially in virtualized environment, the POM-TLB hit can save up to 24 accesses. This significantly reduces the total number of overall memory accesses. Furthermore, data accesses are non-blocking accesses where multiple requests can be on the fly. The access latency can be hidden by means of memory level parallelism such as bank level parallelism, which is common in today's DRAM. On the other hand, an address translation is a critical blocking request where upon a TLB miss, the processor execution stalls. Therefore, the impact of serving the translation request is much higher. Consequently, using the same capacity as a large TLB is likely to save more cycles than using it as L4 data cache. Note that 16MB is a small fraction of a die-stacked DRAM, and as previously mentioned, the rest of die-stacked DRAM can be used as a large data cache via a separate channel without translation traffic contention.

7.2 Results

7.2.1 Experimental Setup

The performance of POM-TLB is evaluated using a combination of real system measurement, PIN-based and Ramulator-like [72] simulation, and performance models. The virtualization platform is QEMU 2.0 with KVM support and the host system is Ubuntu 14.04 running on Intel Skylake [28] with Transparent Huge Pages (THP) [32] turned on. The host system has Intel VT-x with support for Extended Page Tables while the guest OS is Ubuntu 14.04 installed on QEMU also with THP turned on. The system has separate L1 TLBs for each page size (4KB, 2MB, and 1GB in the system) though the applications do not use 1GB size.

The L2 TLB is a unified TLB for both 4KB and 2MB pages. Finally, the specific performance counters (e.g., 0x0108, 0x1008, 0x0149, 0x1049) include page walk cycles taking MMU cache hits into account, so the page walk cycles

	astar	bwaves	canneal	ccomponent	gcc
Overhead Native (%)	13.89	0.73	3.19	0.73	0.30
Overhead Virtual (%)	16.08	7.70	6.34	7.40	12.12
Avg Cycles/L2TLB-miss Native	98	128	53	44	46
Avg Cycles/L2TLB-miss Virtual	114	151	61	1158	88
Frac Large Pages (%)	41.7	0.8	16.0	50.0	29.0
	GemsFDTD	graph500	gups	lbm	libquantum
Overhead Native (%)	10.58	1.03	12.20	0.05	0.02
Overhead Virtual (%)	16.01	7.66	17.20	12.02	7.37
Avg Cycles/L2TLB-miss Native	129	79	43	110	70
Avg Cycles/L2TLB-miss Virtual	133	80	70	290	75
Frac Large Pages (%)	71.0	7.0	2.59	57.4	32.9
	mcf	pagerank	soplex	streamcluster	zeusmp
Overhead Native (%)	10.32	4.07	4.16	0.07	0.01
Overhead Virtual (%)	19.01	6.96	17.07	2.11	10.22
Avg Cycles/L2TLB-miss Native	66	51	144	74	136
Avg Cycles/L2TLB-miss Virtual	169	61	145	76	137
Frac Large Pages (%)	60.7	60.0	12.3	87.2	72.1

Table 7.1: Benchmark Characteristics Related to TLB misses

used in this chapter are the average cycles spent after a translation request misses in L2 TLB.

7.2.2 Workloads

A subset of SPEC CPU and PARSEC applications, which are known to be memory intensive, are chosen. In addition, graph workloads such as the *graph500* and big data workloads such as *connected components* and *pagerank* are also run. The benchmark characteristics are collected from the Intel Skylake platform and are presented in Table 7.1. Applications whose page walk cycles, walk overheads, etc are in a wide range of spectrum (low to high) are used. Since SPEC CPU applications are single threaded, multiple copies of SPEC CPU applications (as in the SPECrate mode) are run to evaluate performance on the multicore simulator. It is ensured that applications do not share the physical memory space through proper virtual-to-physical address translation. For multithreaded workloads, benchmarks are profiled with 8 threads.

7.2.3 Evaluation Methodology

A combination of measurement (on real hardware), simulation and performance modeling is used to estimate the performance of the proposed scheme. First, the workloads listed in Table 7.1 are executed to completion and the Linux *perf* utility is used to measure the total instructions (I_{total}), cycles (C_{total}), number of L2 TLB misses (M_{total}) and total L2 TLB miss penalty cycles (P_{total}) in a manner similar to the methodology in prior work [29–31, 60, 82]. The baseline IPC is obtained as: $IPC_{baseline} = I_{total}/C_{total}$. Also, the ideal cycles C_{ideal} and average translation penalty cycles per L2 TLB miss $P_{Avg}^{Baseline}$ are computed as:

$$C_{ideal} = C_{total} - P_{total} \tag{7.2}$$

$$P_{Avg}^{Baseline} = P_{total}/M_{total} \tag{7.3}$$

Note that the effects of various caching techniques like page walk caches, caching of PTEs in data caches, Intel extended page tables and nested TLBs are already included in the performance measurement since they are part of the base commodity processor. Since the experimental platform has only a single page walker, this result is reasonably accurate. Although the overlap of page walks and instructions is not modeled, it is not a significant amount of the total cycles. The average translation costs per L2 TLB miss, as computed above, are also listed in the workloads table.

Next, PIN and the Linux pagemap are used to generate memory traces for the workloads. For each workload, all load and store requests are recorded. The Linux pagemap is used to extend the PIN tool to include page size and other OS related metadata. The trace contains virtual address, physical address, instruction count, read/write flag, thread ID and page size information of each reference. Memory instructions are traced in detail while the non-memory instructions are abstracted. The memory traces are collected for 20 billion instructions.

Furthermore, a detailed memory hierarchy simulator is used to simulate two levels of private TLBs, two levels of private data caches, a 3rd level shared data cache, and finally, the proposed 3rd level shared memory-based TLB. The simulator also models the size predictor and the cache bypass predictor, which are indexed using memory addresses. Although POM-TLB misses can be handled by existing page walk caches, it is not modeled in the simulation. Instead, the machine measured page walk cycle numbers, which have page walk hits into account, are used. The simulator executes memory references from multiple traces while scheduling them at the proper issue cadence by using their instruction order. Information on the number of instructions in between the memory instructions are captured in the traces and thus memory level parallelism and overlap/lack of overlap between memory instructions are simulated. Note that the simulator is simulating both address translation traffic as well as data request traffic that go into underlying data caches. Finally, the simulator reports the L2 TLB miss cycles and detailed statistics such as hits and misses in the L1, L2 TLBs, data caches, the POM-TLB and predictor performance.

7.2.4 Performance Simulation of POM-TLB

Since the baseline performance (obtained from real system measurements) already includes the benefits of hardware structures such as large pages, EPT and Page Structure Caches, these are not modeled in the simulator, and instead, the baseline ideal cycles together with the estimated cost incurred by the POM-TLB are used when the DRAM-based TLB incurs a miss.

Total cycles taken by the POM-TLB and the resulting IPC for each core are obtained as:

$$C_{total}^{POM\_TLB} = C_{ideal} + M_{total} * P_{Avg}^{POM\_TLB} \tag{7.4}$$

$$IPC_{POM\_TLB} = I_{total}/C_{total}^{POM\_TLB} \tag{7.5}$$

$P_{Avg}^{POM\_TLB}$ denotes the average L2 TLB miss cycles in POM-TLB obtained from simulation. Having obtained the baseline and POM-TLB IPCs for each core, the overall performance improvement of the POM-TLB is obtained. It may be observed that the linear additive formula is used to add the L2 TLB miss cycles to the ideal cycles. This linear performance model ignores potential overlap of TLB processing cycles with execution cycles, but is similar to models used in previous research [29–31, 60, 82]. Such effects not only exist in POM-TLB, but also in the baseline, so the performance impacts from this exists equally in all schemes.

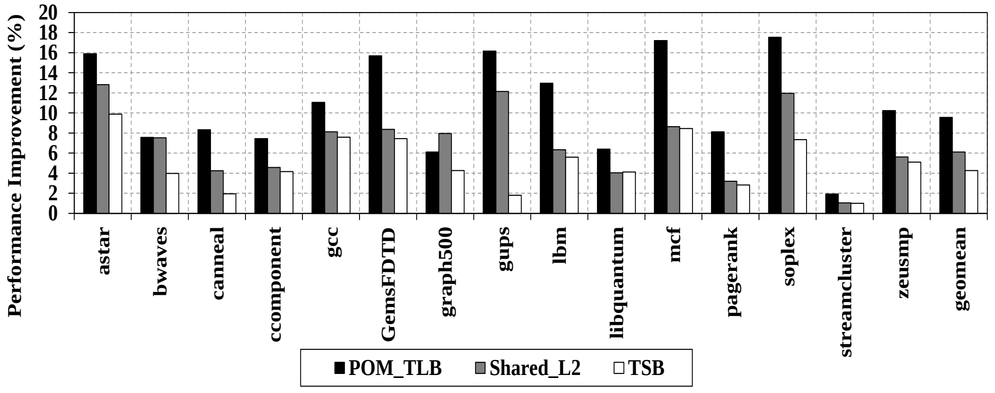


Figure 7.3: Performance Improvement of POM-TLB (8 Core)

In addition, POM-TLB is compared against other prior work, which is annotated as Shared_L2 in the rest of this chapter. This scheme is implemented similar to [31]. Shared_L2 combines private SRAM-based L2 TLBs into a single shared TLB, so when a request misses in L1 TLB, the large SRAM-based shared TLB is looked up. Finally, a comparison is made between POM-TLB and the Translation Storage Buffer (TSB) that exists in SPARC processors. Since TLB misses are handled by OS in SPARC processors, TSB is managed by OS although indexing and address calculation is done by dedicated hardware. Unlike x86 architecture, upon a TLB miss in the SPARC architecture, an OS trap is called and appropriate TSB lookup or a software page table walk is initiated. TSB can be considered as a large MMU cache implemented in a large buffer, but is dedicated and cannot be cached. POM-TLB is compared against such TSB design to show the benefits of POM-TLB.

7.2.5 Performance

Figure 7.3 plots the performance improvements of 16MB POM-TLB on 8 core configuration. The baseline is the execution time gathered from the experimental runs on SkyLake processors. Note that the improvement is shown in percentage (%) and 2 different comparable schemes are presented in addition to POM-TLB. The improvement ranges from 1% in *streamcluster* to 17% in *soplex*. The workloads with high page walk overheads in virtualized platforms have the highest improvement (such as *mcf, soplex, GemsFDTD, astar* and *gups*), which indicates that POM-TLB is effective in reducing costly page walks. The *streamcluster* benchmark does not contain significant page walk overheads to begin with (2.11%). Therefore, this benchmark does not possess a lot of headroom for improvement from POM-TLB. On average, POM-TLB is able to achieve the performance improvement of 9.57%. It may also be noted that these performance gains are obtained with the use of large pages. Even where a large fraction of pages are 2MB pages (for example, *mcf* has 70% and *astar* has 40% large pages), the workloads exhibit considerable performance improvements.

Shared_L2 is able to achieve 6.10% performance improvement on average. This scheme does benefit from sharing of the combined L2 TLB capacities, yet it is still limited in terms of capturing the hot set of TLB entries. Thus, it encounters high shared L2 TLB miss penalty. On the other hand, POM-TLB is able to capture many more TLB entries. First, the physical capacity of POM-TLB is 16MB, which is a few orders of magnitude higher than TLBs in existing systems or L2 TLB capacity in Shared_L2 design. Upon a TLB miss in Shared_L2, when the page walk is initiated,the data caches are also searched since intermediate PTEs are stored in data caches. However, in order to get a complete translation, many of these intermediate entries must be searched until the last level PTE is found. Only then is the translation done. Even though the access's latency only comes from SRAM latency, multiple accesses are made in order to complete the page walk. MMU caches, such as PSC, help to reduce the number of such intermediate entry walks, yet their capacity is very limited, so they only cache a small amount of the TLB misses. In addition, even though POM-TLB is located in die-stacked DRAM, which incurs an access latency similar to DRAM, many of TLB entries are cached in data caches. This enables the POM-TLB to achieve much lower access latency as many of these entries are cached in data caches. Exploiting L2D$ and L3D$ captures a lot of TLB entries in caches. An additional advantage is that single virtual address only requires single entry in data caches whereas Shared_L2 has multiple intermediate PTEs stored in caches, which consumes much more capacity.

TSB achieves an average performance improvement of 4.27% across the workloads. The improvement is surprising low considering that it uses 16MB capacity as in POM-TLB. However, the performance of this scheme is limited as each TLB miss incurs a trap operation, which is required in the operation of TSB as it is software managed. Unlike POM-TLB, which has an associativity of 4, TSB is a direct mapped organization, so it sees more conflict

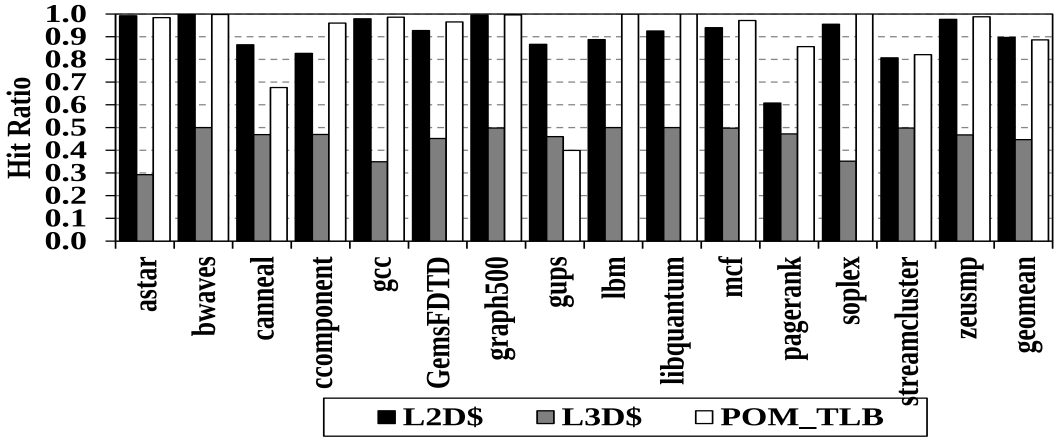


Figure 7.4: Hit Ratio of POM-TLB (8 Core)

misses. POM-TLB uses the 64B cacheline size, so each cacheline has 4 TLB entries. Since the data transfer granularity between die-stacked DRAM and on-chip caches is done at 64B, it allows POM-TLB to have an associativity of 4. Furthermore, TSB entries are not cached as with the POM-TLB, so an access to TSB incurs a higher access latency than POM-TLB. An interesting observation is made for the *gups* benchmark. This benchmark is known to have low locality in page tables, so an ability to achieve high performance for such low spatial locality workloads can show how well each scheme retains translation entries. In the case of TSB, it is not able to capture many of these entries even with 16MB as it only achieves 1.80% improvement. However, POM-TLB achieves performance improvement of 16%, approximately an order of magnitude of difference in performance. Therefore, POM-TLB makes much better use of the 16MB space that is located in die-stacked DRAM.

7.2.6 Hit Ratio

The effectiveness of POM-TLB can be shown using the hit ratio. This can indicate how well POM-TLB can capture L2 TLB misses, thereby reducing costly page walks. Figure 7.4 shows the hit ratio perceived at the different levels of the memory subsystem where TLB entries are stored by the proposed scheme. First, the L2D$ has a very high hit rate of 89.7% on average. Since L2 data caches are private, it is not affected by interference from other cores. Since the L2 capacity is much larger than other private TLB structures in the processor, it keeps a lot of translation traffic from performing page walks. Note that caching of TLB entries is only feasible since POM-TLB is addressable. In conventional systems, the TLB entries are not visible as they are entirely managed by MMUs, yet the novel idea of making the large TLB addressable has enabled a larger number of TLB requests to be cached in rather large on-chip caches.

When a request misses in L2D$, then it is looked up in shared L3D$. The hit ratio here is not as good as L2D$. First, it is a shared data structure, so interference starts degrading performance. Also, a majority of TLB requests are filtered by L2D$, so only requests with a low degree of locality are passed down to the L3D$. However, POM-TLB in die-stacked DRAM again picks up a lot of these requests as shown by the higher hit ratio of 88% on average. POM-TLB can achieve this as the capacity is rather large, so it can recapture many translation requests that missed in a smaller L3D$. It may also be noted that the data caches are also caching the normal data accesses made by the cores and are not being used solely for TLB entries.

7.2.7 Predictor Accuracy

POM-TLB incorporates two predictors, which are size and bypass predictors. The size predictor speculates whether the incoming translation address is going to be a request for a large or small pages. Although there are proposals [124] that enable simultaneous accesses to TLB structures to check both small and large pages, it requires sophisticated design/verification efforts as well as consumes more power. Rather, POM-TLB uses a simple predictor. As seen in Figure 7.5, the size predictor is highly accurate as it achieves an average accuracy of 95%. The accuracy is calculated by dividing the total number of correct speculations by the total number of speculations. In such cases, the second TLB accesses to look for the other size are reduced 95%. Yet, in comparison to performing a serialized access, the predictor can fetch the correct TLB entry in a single POM-TLB access.

The implementation adds a miss penalty if translations miss in data caches as additional on-chip cache lookups are performed prior to accessing POM-TLB. The bypass predictor effectively eliminates such latency and forwards the request directly to POM-TLB upon L2 TLB miss. The predictor achieves a low accuracy of 45.8% on average. Although some workloads such as *bwaves, lbm* and *libquantum* are able to achieve close to perfect accuracy, others such as *soplex* and *pagerank* have a low hit rate. Although the data cache access latencies are an order of magnitude lower than page walk cycles, the

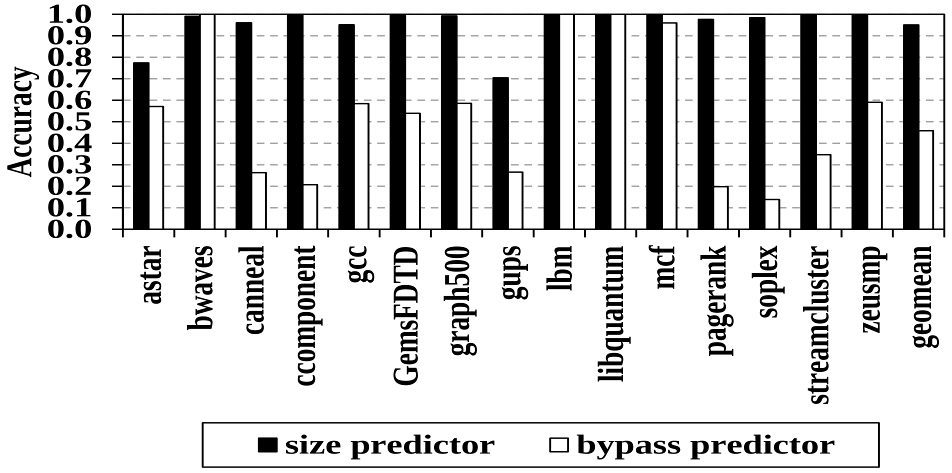


Figure 7.5: POM-TLB Predictor Accuracy (8 core)

misprediction penalty keeps POM-TLB from achieving the best performance.

7.2.8 Row Buffer Hits (RBH) in POM-TLB

The spatial locality of TLB accesses can lead to a high Row Buffer Hits (RBH) in the stacked DRAM. Figure 7.6 plots the RBH values showing that the stacked DRAM achieves a high average RBH of 71%, thereby ensuring a low latency POM-TLB lookup. Each row contains 128 TLB entries, which is similar in capacity as an on-chip L2 TLB. Since they are located in the same row, these TLB accesses are likely to hit in the row buffer. As expected, applications with high spatial locality show high RBH values. For example, *streamcluster* has streaming behaviors, which thus have high spatial locality, explaining its high RBH value in Figure 7.6.

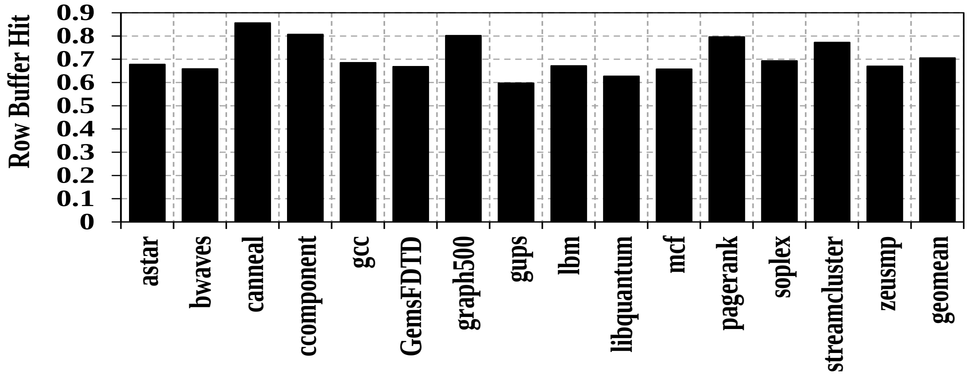


Figure 7.6: Row Buffer Hits in POM-TLB (8 core)

7.2.9 POM-TLB without Data Caches

In this analysis, the performance gain of POM-TLB from storing TLB entries in data caches is presented. Figure 7.7 shows the performance improvement when TLB entries are cached in data caches and performance degradation when not cached. As shown, caching significantly helps performance as it provides an additional performance improvement of 5%. Caching does not help by reducing the number of page table walks. Whether data caches are used or not does not affect the number of page walks as this reduction is performed by the large capacity of POM-TLB. Instead, what caching enables is hiding the long latency of die-stacked DRAM accesses, bridging the latency gap between on-chip TLBs and die-stacked DRAM-based POM-TLB.

7.3 Summary

In this section, the feasibility of building very large L3 TLBs in die-stacked DRAM is presented. A TLB which is part of the memory space enables

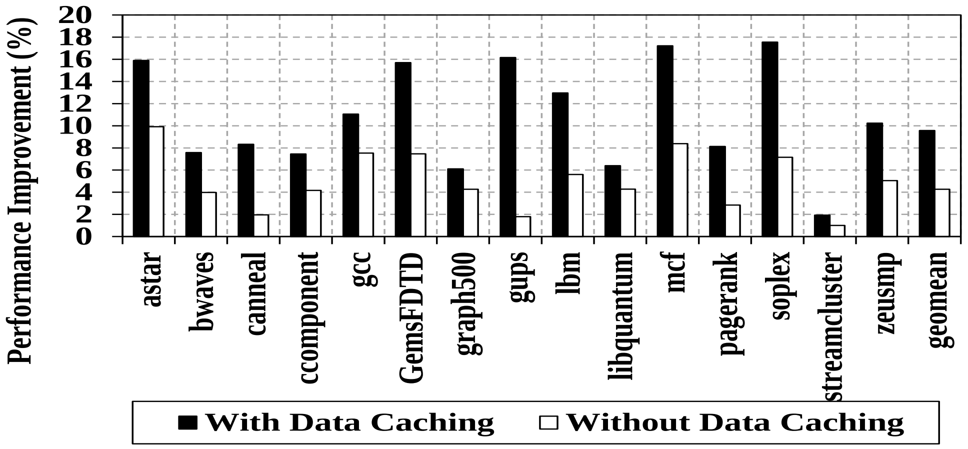


Figure 7.7: POM-TLB With and Without Data Caching (8 core)

the possibility of caching TLB entries in conventional L2 and L3 data caches. A thorough analysis using measurements on state-of-the-art Skylake processors, the simulation and an additive model show that the proposed POM-TLB can practically eliminate the page walk overhead in most memory intensive workloads. In addition, the proposed POM-TLB provides higher performance improvement than previously proposed shared TLB or prefetching techniques at the L1/L2 TLB level. Simulation studies with various number of cores running SPEC, PARSEC and graph workloads demonstrated that more than 16% performance improvement can be obtained in a third of the experimented benchmarks (with an average of 10% over all benchmarks). In most configurations, 99% of the page walks can be eliminated by a very large TLB of size 16 MB.

Chapter 8

Conclusion

As emerging memory technologies are proposed in place of today's off-chip DRAMs, the needs for an innovative architecture to achieve higher performance are introduced. One particular memory technology, die-stacked DRAM, offers much higher bandwidth than off-chip DRAM and higher capacity compared to on-chip SRAM. These unique characteristics create the need for new designs that can exploit die-stacked DRAM in non-traditional ways. This work presents a set of proposals that can utilize die-stacked DRAM as data as well as address translation entry storage.

8.1 Summary

This work describes three major contributions of using die-stacked DRAM for multicore systems. While the research is primarily done using die-stacked DRAM technology, the concepts can be applied to many other heterogeneous memory systems consisting of two or more disparate memory devices.

SILC-FM data management scheme optimizes the data migration between two memories by orchestrating various features of die-stacked DRAM without incurring OS overheads. Migrating subblock in an interleaved fashion optimizes both the bandwidth and the hit rate of the memory system by keeping the current and past history of the memory accesses. SILC-FM also incorporates additional die-stacked DRAM features that extract extra performance. Bypassing, locking and associativity can hold identified hot data much longer than conventional data management schemes.

Granularity-aware migration scheme takes a different perspective on optimizing the data movement by reducing overheads associated with an OS driven migration scheme. An OS driven scheme is advantageous as it does not need to wait for next generation processes that have custom hardware modifications tailored to die-stacked DRAM. Kernel modifications presented in this dissertation can easily detect the degree of spatial locality in either memory regions and execution phases at run time. Then, the OS can adjust its migration granularity to amortize the overheads of migration by migrating as much useful data as possible simultaneously. Such mechanisms are robust to various heterogeneous memory configurations as the optimized problem space does not rely on memory latency or capacity.

Die-stacked DRAM can also allocate a small portion of its capacity for address translation to address the problem that emerging virtualized platforms suffer from scarce on-chip TLB capacity. Even though today's most recent processors have a larger L2 TLB capacity and MMU caches, the capacity is still not enough. Using a small fraction of die-stacked DRAM (e.g., 16MB) is a significant TLB capacity enhancement that eliminates virtually all page walks that are very expensive due to radix-4 nested page tables in virtualized systems. Even though die-stacked DRAM TLB is much slower than SRAM TLBs, the latency issue can be addressed by making the TLB a part of memory space. This enables the TLB entries in die-stacked DRAM to be stored in much faster SRAM data caches. Therefore, highly used TLB entries are serviced from SRAM caches, solving the issues associated with the slow access latency of die-stacked DRAM.

8.2 Future Work

While this dissertation makes significant contributions in using emerging memory technologies, there are still opportunities for future work. This section list possible future work in both data management schemes and address translation schemes. Lastly, a general future work direction is also presented.

8.2.1 Enhancement to Data Management in Emerging Memories

An OS granularity-aware scheme presented in this thesis performs the data migration on the critical path as every access to the slow memory triggers a page permission fault. This adds considerable overhead since the migration does not have to be on the critical path. The work can be extended to have an OS driver that performs the data migration in the background. In fact, whether the data is migrated or not does not affect the correctness of the program execution. Therefore, moving the migration to the background will provide a considerable amount of performance improvement.

8.2.2 Enhancement to POM-TLB

Although PTEs are currently stored in data caches, introducing cacheable TLB entries makes the translation and data request traffic compete for capacity in the data cache. Conventional systems will not distinguish the TLB entries from data entries in making replacement decisions, so the least recently used entries will blindly be chosen and replaced. It is more detrimental to evict TLB entries than data cache entries. First, each 64B cacheline contains 4 TLB entries, which covers 16KB of address range with the 4KB page size. On the other hand, evicting a data entry only affects a single 64B address range. Therefore, conventional caching policies will not achieve the optimal performance by obliviously evicting some important TLB entries. Second, an address translation is often on the critical path of the execution, so servicing the address translation is more likely to be critical than data requests. Giving priority to TLB entries will thus be more beneficial in terms of performance. In sum, the future enhancement to POM-TLB could be devising a new cache replacement policy considering different types of entries in the data caches.

Bibliography

[1] P. J. Nair, D.-H. Kim, and M. K. Qureshi, "ArchShield: Architectural framework for assisting DRAM scaling by tolerating high error rates," in *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 72–83, ACM, 2013.

[2] Z. Li, R. Zhou, and T. Li, "Exploring high-performance and energy proportional interface for phase change memory systems," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013)*, pp. 210–221, IEEE, 2013.

[3] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 2–13, ACM, 2009.

[4] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nature Nanotechnology*, vol. 8, no. 1, pp. 13–24, 2013.

[5] S. Tehrani, J. Slaughter, E. Chen, M. Durlam, J. Shi, and M. DeHerren, "Progress and outlook for MRAM technology," *IEEE Transactions on Magnetics*, vol. 35, no. 5, pp. 2814–2819, 1999.

[6] JEDEC, "High Bandwidth Memory (HBM) DRAM Gen 2 (JESD235A)." `https://www.jedec.org`, 2016.

[7] J. T. Pawlowski, "Hybrid memory cube: breakthrough DRAM performance with a fundamentally re-architected DRAM subsystem," in *Hot Chips*, vol. 23, 2011.

[8] A. Sodani, "Knights Landing (KNL): 2nd Generation Intel® Xeon Phi processor," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, pp. 1–24, IEEE, 2015.

[9] NVIDIA, "NVIDIA Pascal." `http://www.nvidia.com`.

[10] J. Macri, "AMD's next generation GPU and high bandwidth memory architecture: FURY," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, pp. 1–26, IEEE, 2015.

[11] G. B. Berriman, G. Juve, E. Deelman, M. Regelson, and P. Plavchan, "The application of cloud computing to astronomy: A study of cost and performance," in *e-Science Workshops, 2010 Sixth IEEE International Conference on*, pp. 1–7, IEEE, 2010.

[12] Y. Chen, S. Alspaugh, and R. Katz, "Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1802–1813, 2012.

[13] C. Luo, J. Zhan, Z. Jia, L. Wang, G. Lu, L. Zhang, C.-Z. Xu, and N. Sun, "Cloudrank-d: benchmarking and ranking cloud computing systems for data processing applications," *Frontiers of Computer Science*, vol. 6, no. 4, pp. 347–362, 2012.

[14] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, “CHOP: Adaptive filter-based DRAM caching for CMP server platforms,” in *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pp. 1–12, IEEE, 2010.

[15] S. Kumar and C. Wilkerson, “Exploiting spatial locality in data caches using spatial footprints,” in *ACM SIGARCH Computer Architecture News*, vol. 26, pp. 357–368, IEEE Computer Society, 1998.

[16] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, “Simple but effective heterogeneous main memory with on-chip memory controller support,” in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, IEEE Computer Society, 2010.

[17] G. H. Loh, N. Jayasena, K. Mcgrath, M. O’Connor, S. Reinhardt, and J. Chung, “Challenges in Heterogeneous Die-Stacked and Off-Chip Memory Systems,” in *the 3rd Workshop on SoCs, Heterogeneous Architectures, and Workloads (SHAW)*, 2012.

[18] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, “Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories,” in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 126–136, IEEE, 2015.

[19] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proceedings of the international conference on Supercomputing*, pp. 85–95, ACM, 2011.

[20] C. Chou, A. Jaleel, and M. K. Qureshi, "Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1–12, IEEE Computer Society, 2014.

[21] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent hardware management of stacked DRAM as part of memory," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 13–24, IEEE, 2014.

[22] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison cache: A scalable and effective die-stacked DRAM cache," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 25–37, IEEE, 2014.

[23] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the bandwidth wall: challenges in and avenues for CMP scaling," in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 371–382, ACM, 2009.

[24] HMC Consortium, "HMC Specification." http://www.hybridmemorycube.org, 2012.

[25] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating Two-dimensional Page Walks for Virtualized Systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 26–35, ACM, 2008.

[26] Intel, "Intel(R) Virtualization Technology." `http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html`.

[27] T. W. Barr, A. L. Cox, and S. Rixner, "Translation Caching: Skip, Don't Walk (the Page Table)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pp. 48–59, ACM, 2010.

[28] Intel, "6th Generation Intel Core i7-6700K and i5-6600K Processors." `http://www.intel.com/content/www/us/en/processors/core`.

[29] J. Gandhi, M. D. Hill, and M. M. Swift, "Agile Paging: Exceeding the Best of Nested and Shadow Paging," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 707–718, IEEE, 2016.

[30] A. Bhattacharjee and M. Martonosi, "Inter-core Cooperative TLB for Chip Multiprocessors," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, pp. 359–370, ACM, 2010.

[31] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level TLBs for chip multiprocessors," pp. 62–63, 2011.

[32] A. Arcangeli, "Transparent hugepage support," in *KVM Forum*, vol. 9, 2010.

[33] Oracle, "Translation Storage Buffer." `https://blogs.oracle.com/elowe/entry/translation_storage_buffers`.

[34] T. W. Barr, A. L. Cox, and S. Rixner, "SpecTLB: A Mechanism for Speculative Address Translation," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, pp. 307–318, ACM, 2011.

[35] B. Pham, J. Vesely, G. H. Loh, and A. Bhattacharjee, "Using TLB Speculation to Overcome Page Splintering in Virtual Machines," tech. rep., Technical Report DCS-TR-7132015. Rutgers University.

[36] J. H. Ryoo, K. Ganesan, Y.-M. Chen, and L. K. John, "i-MIRROR: A Software Managed Die-Stacked DRAM-Based Memory Subsystem," in *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 82–89, IEEE, 2015.

[37] M. El-Nacouzi, I. Atta, M. Papadopoulou, J. Zebchuk, N. E. Jerger, and A. Moshovos, "A dual grain hit-miss detector for large Die-Stacked DRAM caches," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pp. 89–92, 2013.

[38] S. Franey and M. Lipasti, "Tag tables," in *2015 IEEE 21st International*

Symposium on High Performance Computer Architecture (HPCA), pp. 514–525, IEEE, 2015.

[39] N. D. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan, "Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 38–50, IEEE, 2014.

[40] F. Hameed, L. Bauer, and J. Henkel, "Simultaneously optimizing DRAM cache hit latency and miss rate via novel set mapping policies," in *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pp. 1–10, IEEE, 2013.

[41] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pp. 404–415, ACM, 2013.

[42] G. H. Loh and M. D. Hill, "Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 454–464, ACM, 2011.

[43] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management," *Computer Architecture Letters*, vol. 11, no. 2, pp. 61–64, 2012.

[44] M. K. Qureshi and G. H. Loh, "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 235–246, 2012.

[45] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi, "A mostly-clean DRAM cache for effective hit speculation and self-balancing dispatch," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 247–257, IEEE Computer Society, 2012.

[46] A. Sembrant, E. Hagersten, and D. Black-Shaffer, "TLC: A Tag-less Cache for Reducing Dynamic First Level Cache Energy," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pp. 49–61, ACM, 2013.

[47] B. Goglin and N. Furmento, "Memory migration on next-touch," in *Linux Symposium*, 2009.

[48] M. Oskin and G. H. Loh, "A Software-managed Approach to Die-stacked DRAM," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 188–200, IEEE, 2015.

[49] J. Laudon and D. Lenoski, "The SGI Origin: a ccNUMA highly scalable server," in *ACM SIGARCH Computer Architecture News*, vol. 25, pp. 241–251, ACM, 1997.

[50] E. Hagersten, A. Saulsbury, and A. Landin, "Simple COMA node implementations," in *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, vol. 1, pp. 522–533, IEEE, 1994.

[51] B. Falsafi and D. A. Wood, "Reactive NUMA: a design for unifying S-COMA and CC-NUMA," in *ACM SIGARCH Computer Architecture News*, vol. 25, pp. 229–240, ACM, 1997.

[52] F. X. Lin and X. Liu, "Memif: Towards programming heterogeneous memory asynchronously," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 369–383, ACM, 2016.

[53] M. R. Meswani, G. H. Loh, S. Blagodurov, D. Roberts, J. Slice, and M. Ignatowski, "Toward efficient programmer-managed two-level memory hierarchies in exascale computers," in *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*, pp. 9–16, IEEE, 2014.

[54] C. Cantalupo, V. Venkatesan, J. R. Hammond, K. Czurylo, and S. Hammond, "User extensible heap manager for heterogeneous memory platforms and mixed memory policies," *Architecture document*, 2015.

[55] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and efficient huge page management with ingens," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 705–721, USENIX Association, 2016.

[56] K. J. Nesbit and J. E. Smith, “Data Cache Prefetching Using a Global History Buffer,” *IEEE Micro*, vol. 25, no. 1, pp. 90–97, 2005.

[57] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, “Spatio-temporal memory streaming,” in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 69–80, ACM, 2009.

[58] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, “Temporal streaming of shared memory,” in *ACM SIGARCH Computer Architecture News*, vol. 33, pp. 222–233, IEEE Computer Society, 2005.

[59] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, “Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks,” in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 178–189, IEEE Computer Society, 2014.

[60] A. Bhattacharjee, “Large-reach Memory Management Unit Caches,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 383–394, ACM, 2013.

[61] A. S. Tanenbaum and H. Bos, *Modern operating systems.* Prentice Hall Press, 2014.

[62] I. Yaniv and D. Tsafrir, “Hash, don’t cache (the page table).,” in *SIGMETRICS*, pp. 337–350, 2016.

[63] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal, "Energy-efficient address translation," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 631–643, March 2016.

[64] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient Virtual Memory for Big Memory Servers," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 237–248, 2013.

[65] G. B. KANDIRAJU and A. SIVASUBRAMANIAM, "Going the distance for TLB prefetching: An application-driven study," in *Proceedings-International Symposium on Computer Architecture*, pp. 195–206, 2002.

[66] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-based TLB Preloading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 117–127, ACM, 2000.

[67] L. Zhang, E. Speight, R. Rajamony, and J. Lin, "Enigma: architectural and operating system support for reducing the impact of address translation," in *Proceedings of the 24th ACM International Conference on Supercomputing*, pp. 159–168, ACM, 2010.

[68] S. Srikantaiah and M. Kandemir, "Synergistic TLBs for high performance address translation in chip multiprocessors," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 313–324, IEEE Computer Society, 2010.

[69] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach TLBs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 258–269, IEEE, 2012.

[70] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB reach by exploiting clustering in page translations," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 558–567, IEEE, 2014.

[71] T. E. Carlson, W. Heirmant, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–12, IEEE, 2011.

[72] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE Computer Architecture Letters*, pp. 45–49, 2016.

[73] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[74] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pp. 72–81, ACM, 2008.

[75] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, “Introducing the graph 500,” *Cray Users Group (CUG)*, 2010.

[76] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, “PIN: a binary instrumentation tool for computer architecture research and education,” in *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, ACM, 2004.

[77] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, “DRAMSim2: A cycle accurate memory system simulator,” *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[78] AMD, “AMD A-Series Desktop APUs.” `http://www.amd.com/en-us/products/processors/desktop/a-series-apu`.

[79] Linux Kernel Organization, “The Linux Kernel Archives.” `http://www.kernel.org`.

[80] R. Crooke and F. Al, “Intel Non-Volatile Memory Inside. The Speed of Possibility Outside,” *Intel Developer Foreum (IDF)*, 2015.

[81] Y. Ishimatsu, “Memory Hotplug.” `https://events.linuxfoundation.org`.

[82] I. Yaniv and D. Tsafrir, “Hash, Don’T Cache (the Page Table),” in *Proceedings of the 2016 ACM SIGMETRICS International Conference*

on Measurement and Modeling of Computer Science, pp. 337–350, ACM, 2016.

[83] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pp. 158–165, ACM, 1991.

[84] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi, "The HPC Challenge (HPCC) benchmark suite," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, p. 213, ACM, 2006.

[85] S. N. Laboratories, "Mantevo." `https://software.sandia.gov/mantevo/`.

[86] X. Wu and V. Taylor, "Power and performance characteristics of CORAL Scalable Science Benchmarks on BlueGene/Q Mira," in *2015 Sixth International Green Computing Conference and Sustainable Computing Conference (IGSC)*, pp. 1–6, IEEE, 2015.

[87] J. H. Ryoo, M. R. Meswani, R. Panda, and L. K. John, "POSTER: SILC-FM: Subblocked interleaved Cache-Like Flat Memory Organization," in *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 435–437, IEEE, 2016.

[88] J. H. Ryoo, A. Prodromou, M. R. Meswani, and L. K. John, "SILC-FM: Subblocked interleaved Cache-Like Flat Memory Organization," in *2017 IEEE International Symposium on High Performance Computer Architecture*.

[89] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "DiDi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory," in *2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 340–349, IEEE, 2011.

[90] A. C. de Melo, "The new linux perf tools," in *Slides from Linux Kongress*, vol. 18, 2010.

[91] J. Buell, D. Hecht, J. Heo, K. Saladi, and R. Taheri, "Methodology for performance analysis of VMware vSphere under Tier-1 applications," *VMware Technical Journal*, vol. 2, no. 1, 2013.

[92] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee, "Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways?," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 1–12, ACM, 2015.

[93] S. J. E. Wilton and N. P. Jouppi, "CACTI: an enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, 1996.

[94] S. M. Khan, D. A. Jiménez, D. Burger, and B. Falsafi, "Using dead blocks as a virtual victim cache," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pp. 489–500, ACM, 2010.

[95] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A modified approach to data cache management," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 93–103, IEEE Computer Society Press, 1995.

[96] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 430–441, ACM, 2011.

[97] M. Talluri and M. D. Hill, "Surpassing the tlb performance of superpages with less operating system support," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 171–182, ACM, 1994.

[98] C. Chou, A. Jaleel, and M. K. Qureshi, "BEAR: techniques for mitigating bandwidth bloat in gigascale DRAM caches," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 198–210, ACM, 2015.

[99] C.-C. Chou, A. Jaleel, and M. Qureshi, "BATMAN: Maximizing Bandwidth Utilization for Hybrid Memory Systems," tech. rep., Technical

Report TR-CARET-2015-01. Technical Report for Computer ARchitecture and Emerging Technologies (CARET) Lab.

[100] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking bandwidth for GPUs in CC-NUMA systems," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 354–365, IEEE, 2015.

[101] JEDEC, "High Bandwidth Memory (HBM) DRAM (JESD235)." `https://www.jedec.org`, 2013.

[102] "AMD Working With Hynix For Development of High-Bandwidth 3D Stacked Memory." `http://wccftech.com`, 2013.

[103] Micron, "HMC Generation 2." `http://www.hybridmemorycube.org`, 2013.

[104] JEDEC, "DDR SDRAM (JESD79-3C)." `https://www.jedec.org`, 2008.

[105] Micron Technology Inc., "TN-46-03 Calculating Memory System Power for DDR," 2001.

[106] SPEC, "SPEC CPU 2006." `http://www.spec.org`, 2006.

[107] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large Intel® Itanium® programs with dynamic instrumentation," in *37th International Symposium on Microarchitecture*, pp. 81–92, IEEE, 2004.

[108] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, “Using SimPoint for Accurate and Efficient Simulation,” *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, 2003.

[109] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures,” in *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 469–480, IEEE, 2009.

[110] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, “XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis,” in *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, (Kyoto).

[111] I. Karlin, J. Keasler, and R. Neely, “Lulesh 2.0 updates and changes,” *Tech. Rep. LLNL-TR-641973*, 2013.

[112] A. Sembrant, D. Black-Schaffer, and E. Hagersten, “Phase behavior in serial and parallel applications,” in *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 47–58, 2012.

[113] AMD, “AMD64 Architecture Programmer’s Manual.” `http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf`.

[114] Intel, “Intel Xeon Phi Processor Software.” `https://software.intel.com/en-us/articles/xeon-phi-software`.

[115] F. Bedeschi, C. Resta, O. Khouri, E. Buda, L. Costa, M. Ferraro, F. Pellizzer, F. Ottogalli, A. Pirovano, M. Tosi, R. Bez, R. Gastaldi, and G. Casagrande, "An 8Mb demonstrator for high-density 1.8V Phase-Change Memories," in *2004 Symposium on VLSI Circuits, 2004*, pp. 442–445, IEEE, 2004.

[116] C. T. Office, "Phase Change Memory." `http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf`.

[117] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, "Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB," in *2017 IEEE International Symposium on Computer Architecture*.

[118] G. Hinton, D. Sager, M. Upton, D. Boggs, *et al.*, "The microarchitecture of the Pentium® 4 processor," in *Intel Technology Journal*, 2001.

[119] A. Arcangeli, "Linux KVM Forum," 2008. `http://www.linux-kvm.org/page/KVM_Forum_2008`.

[120] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, 1995.

[121] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb, "Die Stacking (3D) Microarchitecture," in *Proceedings of the 39th Annual IEEE/ACM International*

Symposium on Microarchitecture, pp. 469–479, IEEE Computer Society, 2006.

[122] C.-C. Huang and V. Nagarajan, "ATCache: Reducing DRAM Cache Latency via a Small SRAM Tag Cache," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pp. 51–60, ACM, 2014.

[123] L. Zhao, R. Iyer, R. Illikkal, and D. Newell, "Exploring DRAM cache architectures for CMP server platforms," in *25th International Conference on Computer Design*, pp. 55–62, 2007.

[124] A. Seznec, "Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB," *IEEE Trans. Computers*, vol. 53, no. 7, pp. 924–927, 2004.

[125] M. Wu and W. Zwaenepoel, "Improving TLB miss handling with page table pointer caches," tech. rep.

[126] R. Love, *Linux Kernel Development.* 3rd ed., 2010.

[127] ITRS, "The International Technology Roadmap for Semiconductors, Process Integration, Device and Structures." `http://www.itrs.net`, 2013.

[128] Amazon, "Amazon EC2 - Virtual Server Hosting." `https://aws.amazon.com/ec2/`.

[129] Rackspace, "OPENSTACK - The Open Alternative To Cloud Lock-In." `https://www.rackspace.com/en-us/cloud/openstack`.

[130] "NVIDIA Volta Next-Generation GPU Unveiled Features 1TB/S Bandwidth and Stacked DRAM." `http://wccftech.com`.

[131] SUN, "The SPARC Architecture Manual." `http://www.sparc.org/standards/SPARCV9.pdf`.

[132] JEDEC, "Wide I/O Single Data Rate (Wide I/O SDR)." `https://www.jedec.org/standards-documents/docs/jesd229`, 2011.

[133] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, vol. 3, 2009.

[134] E. Bolotin, D. Nellans, O. Villa, M. O'Connor, A. Ramirez, and S. W. Keckler, "Designing efficient heterogeneous memory architectures," *IEEE Micro*, vol. 35, no. 4, pp. 60–68, 2015.

[135] F. Gaud, B. Lepers, J. Funston, M. Dashti, A. Fedorova, V. Quéma, R. Lachaize, and M. Roth, "Challenges of memory management on modern NUMA systems," *Communications of the ACM*, vol. 58, no. 12, pp. 59–66, 2015.

[136] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal, "Range Translations for Fast Virtual Memory," *IEEE Micro*, pp. 118–126, May 2016.

[137] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor," *IEEE Micro*, vol. 30, no. 2, pp. 16–29, 2010.

[138] D. Gove, "CPU2006 Working Set Size," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, 2007.

[139] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, 2002.

[140] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM System Journal*, vol. 9, no. 2, 1970.

[141] H.-C. Shih, P.-W. Luo, J.-C. Yeh, S.-Y. Lin, D.-M. Kwai, S.-L. Lu, A. Schaefer, and C.-W. Wu, "DArT: A Component-Based DRAM Area, Power, and Timing Modeling Tool," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 9, 2014.

[142] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008.

[143] Y. Huai, M. Pakala, F. Albert, T. Valet, and P. Nguyen, "Observation of spin-transfer switching in deep submicron-sized and low-resistance mag-

netic tunnel junctions," *Appl. Phys. Lett.*, vol. 84, no. cond-mat/0504486, pp. 3118–3120, 2005.

[144] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, pp. 41–46, 2005.

[145] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *31st Annual International Symposium on Computer Architecture*, pp. 338–349, IEEE, 2004.

[146] Intel, "Intel 64 and IA-32 Architectures Software Developers Manual Volume 3A: System Programming Guide Part 1,"

[147] Intel, "Intel Virtualization Technology: Processor Virtualization Extensions and Intel Trusted execution Technology,"

[148] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 423–432, IEEE Computer Society, 2006.

[149] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013)*, pp. 639–650, IEEE, 2013.

[150] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *2009 International Symposium on Computer Architecture*, pp. 14–23, ACM, 2009.

[151] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System Using Phase-change Memory Technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pp. 24–33, ACM, 2009.

[152] H. Jang, Y. Lee, J. Kim, Y. Kim, J. Kim, J. Jeong, and J. W. Lee, "Efficient footprint caching for Tagless DRAM Caches," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 237–248, IEEE, 2016.

[153] J. Sim, J. Lee, M. K. Qureshi, and H. Kim, "FLEXclusion: balancing cache capacity and on-chip bandwidth via flexible exclusion," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pp. 321–332, IEEE, 2012.

[154] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee, "A fully associative, tagless DRAM cache," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 211–222, IEEE, 2015.

[155] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the

Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 37–48, ACM, 2012.

[156] B. Lepers, V. Quéma, and A. Fedorova, "Thread and memory placement on NUMA systems: asymmetry matters," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pp. 277–289, 2015.

[157] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma, "Large pages may be harmful on NUMA systems," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pp. 231–242, 2014.

[158] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 381–394, ACM, 2013.

[159] Xilinx, "Xilinx SSI Technology," 2012.

[160] J. Gaur, M. Chaudhuri, P. Ramachandran, and S. Subramoney, "Near-optimal Access Partitioning for Memory Hierarchies with Multiple Heterogeneous Bandwidth Sources," in *Proceedings of the 2017 23rd IEEE/ACM International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[161] D. Chiou, P. Jain, L. Rudolph, and S. Devadas, "Application-specific memory management for embedded systems using software-controlled caches," in *Proceedings of the 37th Annual Design Automation Conference*, pp. 416–419, ACM, 2000.

[162] Z. Wang, D. A. Jiménez, C. Xu, G. Sun, and Y. Xie, "Adaptive placement and migration policy for an STT-RAM-based hybrid cache," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 13–24, IEEE, 2014.

[163] A. Hartstein, V. Srinivasan, T. R. Puzak, and P. G. Emma, "Cache miss behavior: is it 2?," in *Proceedings of the 3rd conference on Computing frontiers*, pp. 313–320, ACM, 2006.

[164] IBM, "Power8 Processor User's Manual for the Single-Chip Module,"

[165] D. Kaseridis, J. Stuecheli, J. Chen, and L. K. John, "A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large CMP systems," in *2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1–11, IEEE, 2010.

[166] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," in *ACM SIGOPS Operating Systems Review*, vol. 38, pp. 177–188, ACM, 2004.

[167] Intel, "Intel Xeon Processor 5500 Series Datasheet, Vol. 1,"

[168] SanDisk, "CloudSpeed Ultra Gen. II SATA SSD,"

[169] M. Saxena and M. M. Swift, "FlashVM: Revisiting the Virtual Memory Hierarchy," in *HotOS*, pp. 13–13, 2009.

[170] Intel, "Intel 64 and IA-32 Architectures Software Developers Manual Volume 3B: System Programming Guide, Part 2,"

[171] E. Seo, S. Y. Park, and B. Urgaonkar, "Empirical Analysis on Energy Efficiency of Flash-based SSDs," in *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, HotPower'08, 2008.

[172] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *35th International Symposium on Computer Architecture*, pp. 39–50, IEEE, 2008.