박 사 학 위 논 문
Ph.D. Dissertation

가상 메모리 시스템의 성능과 전력 효율성을 개선하기 위한 하드웨어 아키텍처와 운영체제 연구

Improving the performance and energy efficiency of the virtual memory system by skipping unnecessary translations and dynamically adjusting HW translation coverage

2019

박 창 현 (朴 昶 炫 Park, Chang Hyun)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

박 사 학 위 논 문

가상 메모리 시스템의 성능과 전력 효율성을 개선하기 위한 하드웨어 아키텍처와 운영체제 연구

2019

박 창 현

한 국 과 학 기 술 원

전산학부

가상 메모리 시스템의 성능과 전력 효율성을 개선하기 위한 하드웨어 아키텍처와 운영체제 연구

박 창 현

위 논문은 한국과학기술원 박사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2019년 05월 30일

심사위원장 허 재 혁 (인)

심 사 위 원 맹 승 렬 (인)

심 사 위 원 권 영 진 (인)

심 사 위 원 유 민 수 (인)

심 사 위 원 안 정 섭 (인)

Improving the performance and energy efficiency of the virtual memory system by skipping unnecessary translations and dynamically adjusting HW translation coverage

Chang Hyun Park

Advisor: Jaehyuk Huh

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

Daejeon, Korea
May 30, 2019

Approved by

Jaehyuk Huh
Professor of School of Computing

The study was conducted in accordance with Code of Research Ethics[1].

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

DCS
20155141

박창현. 가상 메모리 시스템의 성능과 전력 효율성을 개선하기 위한 하드웨어 아키텍처와 운영체제 연구. 전산학부 . 2019년. 75+iv 쪽. 지도교수: 허재혁. (영문 논문)

Chang Hyun Park. Improving the performance and energy efficiency of the virtual memory system by skipping unnecessary translations and dynamically adjusting HW translation coverage. School of Computing . 2019. 75+iv pages. Advisor: Jaehyuk Huh. (Text in English)

초 록

가상 메모리 시스템은 프로그램 개발자에게 다양한 개발의 편의성을 제공해준다. 하지만 가상 메모리 시스템을 사용하려면 모든 메모리 접근마다 주소 변환 과정이 필요하다. 최근 대용량 메모리 시스템들이 도입되고 이런 시스템을 활용하는 대용량 메모리 프로그램들이 도입되며 가상 메모리 주소 변환 시스템에서 심각한 성능 하락이 발생하기 시작하였다. 본 학위논문에서는 가상 메모리 주소 변환 시스템의 성능을 개선하는 두 가지 방법을 연구하였다. 첫째는 불필요한 주소 변환을 생략하는 것이며, 둘째는 주소 변환을 저장하는 하드웨어 구조의 주소 변환 표현 범위를 개선 시키는 방법이다.

본 학위 논문의 첫 연구를 위해 프로세서 캐시 계층을 가상 주소로 접근하는 기법을 활용하여 대부분의 주소변환을 캐시계층 이후로 늦추어 성능과 전력 효율성을 개선시켰다. 주소변환을 캐시계층 이후로 늦추게 될 경우 몇 개의 워크로드에서 성능 개선이 발생한다. 하지만 대용량 메모리 워크로드를 비롯하여 주소 변환에 문제가 되는 워크로드의 경우 여전히 주소 변환에서 성능 문제가 발생한다. 주소 변환 하드웨어 저장 구조의 (TLB) 확장성을 개선하기 위해 수천개의 세그멘트를 활용한 주소 변환 기법을 연구하였다. 본 기법의 평가를 통해 본 연구에서 제안한 기법들에 의해 TLB 로의 접근이 현저히 줄게 되며 전력 소모도 개선 하였다. 또한 세그먼트 기반 TLB 확장성 기법에 의해 성능 개선을 이룰 수 있었따.

다양한 실행 환경과 이종 메모리에 의해 메모리는 파편화되는 것이 관찰되는 가운데, 본 학위 논문의 두 번째 연구는 실행 중에 동적으로 TLB 의 표현력을 (Coverage) 조절할 수 있는 기법을 제안한다. 해당 기법은 하드웨어와 소프트웨어 공동 기법으로 운영체제의 메모리 할당에 의해 발생하는 메모리 할당의 연속성을 활용하는 기법이다. 운영체제는 메모리 할당의 연속성을 페이지 테이블에 기입하고, 하드웨어는 해당 정보를 활용하여 기존대비 개선된 표현력을 갖는 TLB 를 제공한다. 본 연구를 평가한 결과 다양한 메모리 할당 상황에서 본 기법은 대부분의 경우에서 가장 좋은 성능을 발휘하였으며, 최고 성능을 내지 못한 경우에서도 최고 성능을 발휘한 기존 기술과 유사한 성능을 낼 수 있었다.

핵 심 낱 말 가상 메모리, 주소 변환 저장 장치, 가상 주소 캐시, 페이지 테이블, 운영체제, 컴퓨터구조

Abstract

Virtual memory provides rich functionality for the program developer. However, address translation accompanies the virtual memory system, requiring an address translation for every memory access. Recently with the rise of big memory systems and big memory workloads the virtual memory address translation system has been suffering great misses, resulting in significant performance overhead. This dissertation focuses on improving the virtual memory translation system with two approaches. Firstly, the virtual memory system design is revisited, and the cache hierarchy is modified to allow skipping of unnecessary address translations. Secondly, the page table is extended and hardware support is added to improve translation coverage of the translation lookaside buffer (TLB).

Based on the virtual caching concept, the first part of this dissertation proposes a hybrid virtual memory architecture extending virtual caching to the entire cache hierarchy, aiming to improve both performance and energy consumption by delaying translation. For large memory applications, delayed translation alone cannot solve the address translation problem, as fixed-granularity delayed TLBs may not scale with the increasing memory requirements. To mitigate the translation scalability problem, this study proposes a delayed many segment translation designed for the hybrid virtual caching. The experimental results show that our approach effectively lowers accesses to the TLBs, leading to significant power savings. In addition, the approach provides performance improvement with scalable delayed translation with variable length segments.

Under fragmented and diverse memory allocations that occur due to diverse execution environments and memory heterogeneity, the second part of this dissertation proposes a novel HW-SW hybrid translation architecture, which can adapt to different memory mappings efficiently. The most important benefit of hybrid coalescing is its ability to change the coverage of the anchor entry dynamically, reflecting the current allocation contiguity status. By using the contiguity information directly set by the operating system, the technique can provide scalable translation coverage improvements with minor hardware changes, while allowing the flexibility of memory allocation. Our experimental results show that across diverse allocation scenarios with different distributions of contiguous memory chunks, the proposed scheme can effectively reap the potential translation coverage improvement from the existing contiguity.

Keywords Virtual memory, Translation lookaside buffer, Virtual Caching, Page Table, Operating System, Computer Architecture

Contents

List of Tables

List of Figures

Chapter 1. Introduction

Virtual memory provides rich functionality for the program developer. However, address translation accompanies the virtual memory system, requiring an address translation for every memory access. Recently with the rise of big memory systems and big memory workloads the virtual memory address translation system has been suffering great misses, resulting in significant performance overhead. This dissertation focuses on improving the virtual memory translation system with two approaches. Firstly, the virtual memory system design is revisited, and the cache hierarchy is modified to allow skipping of unnecessary address translations. Secondly, the page table is extended and hardware support is added to improve translation coverage of the translation lookaside buffer (TLB).

Conventional translation look-aside buffers (TLBs) are required to complete address translation with short latencies, as the address translation is on the critical path of all memory accesses even for L1 cache hits. Such strict TLB latency restrictions limit the TLB capacity, as the latency increase with large TLBs may lower the overall performance even with potential TLB miss reductions. Furthermore, TLBs consume a significant amount of energy as they are accessed for every instruction fetch and data access. To avoid the latency restriction and reduce the energy consumption, virtual caching techniques have been proposed to defer translation to after L1 cache misses. However, an efficient solution for the synonym problem has been a critical issue hindering the wide adoption of virtual caching.

Based on the virtual caching concept, the first part of this dissertation proposes a hybrid virtual memory architecture extending virtual caching to the entire cache hierarchy, aiming to improve both performance and energy consumption. The hybrid virtual caching uses virtual addresses augmented with address space identifiers (ASID) in the cache hierarchy for common non-synonym addresses. For such non-synonyms, the address translation occurs only after last-level cache (LLC) misses. For uncommon synonym addresses, the addresses are translated to physical addresses with conventional TLBs before L1 cache accesses. To support such hybrid translation, we propose an efficient synonym detection mechanism based on Bloom filters which can identify synonym candidates with few false positives. For large memory applications, delayed translation alone cannot solve the address translation problem, as fixed-granularity delayed TLBs may not scale with the increasing memory requirements. To mitigate the translation scalability problem, this study proposes a delayed many segment translation designed for the hybrid virtual caching. The experimental results show that our approach effectively lowers accesses to the TLBs, leading to significant power savings. In addition, the approach provides performance improvement with scalable delayed translation with variable length segments.

To mitigate excessive TLB misses in large memory applications, techniques such as large pages, variable length segments, and HW coalescing, increase the coverage of limited hardware translation entries by exploiting the contiguous memory allocation. However, recent studies show that in non-uniform memory systems, using large pages often leads to performance degradation, or allocating large chunks of memory becomes more difficult due to memory fragmentation. Although each of the prior techniques favors its own best chunk size, diverse contiguity of memory allocation in real systems cannot always provide the optimal chunk of each technique.

Under such fragmented and diverse memory allocations, the second part of this dissertation proposes a novel HW-SW hybrid translation architecture, which can adapt to different memory mappings efficiently. In the proposed *hybrid coalescing* technique, the operating system encodes memory conti-

guity information in a subset of page table entries, called *anchor entries*. During address translation through TLBs, an anchor entry provides translation for contiguous pages following the anchor entry. As a smaller number of anchor entries can cover a large portion of virtual address space, the efficiency of TLB can be significantly improved. The most important benefit of hybrid coalescing is its ability to change the coverage of the anchor entry dynamically, reflecting the current allocation contiguity status. By using the contiguity information directly set by the operating system, the technique can provide scalable translation coverage improvements with minor hardware changes, while allowing the flexibility of memory allocation. Our experimental results show that across diverse allocation scenarios with different distributions of contiguous memory chunks, the proposed scheme can effectively reap the potential translation coverage improvement from the existing contiguity.

1.1 Contributions

This dissertation provides the following contributions that come together to improve the virtual memory address translation system more efficient.

- This dissertation shows that the virtual cache synonym problem, that have prevent virtual cache from widespread use, is infrequent or not observed at all in the various benchmarks that we have studied.
- This dissertation exploits the infrequency of accesses to synonyms and proposes a synonym filter, a filter that can quickly identify any synonym memory accesses.
- This dissertation proposes *hybrid virtual caching* using the synonym filter to enable virtual caching for most memory accesses, while falling back to the conventional physical caching for problematic synonym cases.
- This dissertation shows that delaying translation to after a last level cache miss can reduce TLB misses compared to conventional TLBs to some extent, but for problematic applications, even delayed translation suffers high TLB misses.
- This dissertation proposes *scalable delayed translation*, a translation scheme that provides an efficient many segment based translation scheme that is enabled by the relaxed time constraint provided by the delayed translation.
- This dissertation explores cache shootdowns, that are caused by page mapping changes as an effort to synchronize the mappings and any meta-data of the virtual cache that needs to be updated.
- This dissertation identifies that contiguity in memory mapping of the same application can be affected by running with other applications with different memory allocation behaviors.
- This dissertation proposes a HW-SW TLB and page table design that allows dynamically controlling the TLB coverage.
- This dissertation proposes a TLB scheme that supports dynamically adjusting the TLB coverage.
- This dissertation provides a mechanism to select the TLB coverage matching the application memory allocation property.
- This dissertation explores the effect of virtualization on TLB coalescing.

1.2 Thesis Statement

The computing environment is changing, with newer memory technologies being introduced [28], disaggregated/remote memory technology being enabled by fast, low latency and high bandwidth network fabrics [4, 46, 25, 41]. In such environments, the effective memory capacity of each system will become larger. To manage and use the memory, the virtual memory system will be used for the foreseeable future. To reap the benefits of virtual memory, one must pay the virtual memory tax: address translation. With memory migration for tiered memory and remote/disaggregated memory becoming a mechanism to make efficient use of the heterogeneous memory system [3, 41], the contiguity based address translation efforts that have been proposed by the industry [48, 53, 31] and academia [59, 58, 10, 37] will not suffice. To this end, **this dissertation aims to reduce TLB performance bottleneck in the wake of changing computing environments.**

1.3 Dissertation Organization

This dissertation is organized as follows:

- Firstly, we explore skipping unnecessary translations to minimize the stress on the TLB, relax the latency constraint on the TLB, and finally improve the efficiency of the TLB. We do so by using synonym filters and scalable delayed translation to offer *Hybrid Virtual Caching*, described in Chapter 2.
- Secondly, we explore a TLB with variable TLB coverage enabled by HW and SW co-design, to adapt to dynamic runtime memory allocation varabilities in *Hybrid TLB Coalescing*, described in Chapter 3
- Finally, Chapter 4 concludes this dissertation and discusses some future work and insights gathered during the course of this dissertation.

Chapter 2. Skipping Unnecessary Translations with *Hybrid Virtual Caching*

2.1 Introduction

With ever growing system memory capacity and increasing application requirements, traditional memory virtualization has become a performance bottleneck for big memory applications [10, 59]. For such applications, conventional translation lookaside buffers (TLBs) can no longer cover large working sets and generate excessive TLB misses. Furthermore, in the traditional memory hierarchy, the address translation through TLBs must be completed before L1 cache tag matching. Since the address translation is on the critical path of memory operations, it is difficult to increase the TLB capacity arbitrarily to cover more memory pages. In addition, the address translation consumes a significant amount of dynamic power for TLB accesses [11, 65].

An alternative approach to reduce the cost of address translation is virtual caching [11, 24, 68, 39, 32, 33, 34]. Virtual caching postpones the address translation until an L1 cache miss occurs. It eliminates the critical translation overhead between the core and L1 cache, reducing the energy consumption for address translation. However, a critical problem of virtual caching is the synonym problem where different virtual addresses can be mapped to the same physical address, allowing multiple copies of the same data in a cache. Prior studies address the synonym problem with solutions tuned for L1 virtual caches [11, 24, 68, 32, 33, 34]. They invalidate synonyms with a reverse translation to identify existing synonyms [11, 24, 68], or use self-invalidation with coherence protocol supports [39]. A software-oriented approach has also been proposed with software-based cache miss handling [32].

Inspired by the prior virtual caching techniques, this paper proposes a new hybrid virtual caching architecture supporting efficient synonym detection and scalable delayed translation. Unlike the prior virtual caching approaches, the proposed hybrid caching extends the scope of virtual addressing to the entire cache hierarchy for non-synonym addresses. Synonym addresses are detected by a synonym filter, and translated to physical addresses with conventional TLBs. Each physical memory block is addressed only by a single name, either virtual or physical address. As non-synonym private pages account for the majority of memory pages, most of the actual address translations are deferred to after last-level cache (LLC) misses. By extending the scope of virtual caching, the proposed hybrid translation aims not only to reduce the TLB access energy, but also to improve the translation performance, since large on-chip caches often contain cachelines which could have missed the conventional TLBs. Figure 2.1 describes the overall translation architecture of the proposed hybrid virtual caching. For synonym candidates detected by the synonym filter, TLB accesses occur prior to the L1 cache accesses. For non-synonym pages, delayed translation is applied only after LLC misses.

To allow such a hybrid translation, a key component is the *synonym filter*. Exploiting the fact that synonym pages are uncommon in real systems, the study proposes a synonym filter design based on Bloom filters with a low latency and low power consumption [15]. The operating system updates the bloom filter values, when it changes the page state to shared (synonym). The synonym filter guarantees the detection of synonym pages, while occasionally causing false-positives. With the synonym filter, any page in the virtual address space can be changed to a synonym page without restriction, unlike the prior approach [11].

Delaying translation can reduce TLB access overheads both in energy and latency significantly. However, for applications with large working sets, the delayed translation may not completely eliminate performance degradation due to a large number of delayed TLB misses. To reduce such translation costs, in addition to the traditional page granularity translation which will eventually reach the coverage limit, this paper proposes a many segment translation mechanism. It extends the prior variable-length segment translation [10, 37]. The prior segment translation proposed for physical caches, maps part of the virtual address space to one or more contiguous physical regions with variable length segments for each process. Unlike the prior segment approaches on the critical core-to-L1 path, providing 10s of concurrent segments [37], the proposed delayed segment translation can support 1000s of concurrent segments efficiently, supporting better OS memory allocation flexibility to mitigate internal and external fragmentation problems.

The proposed architecture differs from the prior approaches in four aspects. First, this study extends virtual caching to the entire cache hierarchy. For non-synonym cachelines, cache blocks are indexed and tagged by their address space identifier (ASID) and virtual address. The coherence mechanism also uses virtual addresses with ASID for non-synonym cachelines. Second, this study proposes an efficient HW and SW-combined synonym filtering technique based on Bloom filters. Unlike the prior pure hardware-oriented hash-based synonym detection [69], in this study, the operating system updates the Bloom filters, not requiring any HW synonym tracking. With the hash-based filtering technique, only the synonym candidates access TLBs before L1 accesses. Third, as the fixed page-granularity delayed translation after LLC misses will reach its coverage limit for large memory applications, this study investigates segment-based delayed translation supporting many concurrent segments. Finally, the proposed scheme mitigates the overhead of HW-based virtual machine supports, since the costly two-step translation is delayed until LLC misses. We propose to extend the synonym filter to detect synonym pages induced by both the guest OS and hypervisor.

The experimental results show that the proposed synonym filter is very effective with negligible false positives. The performance of memory intensive applications is improved by 10.7% compared to the physically addressed baseline system, and the power consumption of the translation components is reduced by 60%. For virtualized systems, the performance gain becomes larger with 31.7%, compared to a system with a state-of-the-art translation cache for two-dimensional address translation.

The rest of this paper is organized as follows. Section 2.2 presents virtual caching and its synonym problem with other prior work. Section 2.3 presents the proposed hybrid translation and synonym filtering. Section 2.4 describes the segment-based delayed translation. Section 2.5 presents how to support system virtualization efficiently. Section 2.6 presents the performance and energy improvements. Section 2.8 concludes the work.

2.2 Background & Motivation

2.2.1 Virtual Caching

Virtual caching postpones the address translation until a cache miss occurs, allowing caches to store cachelines with virtual address [11, 24, 68, 39, 32, 33, 34, 70]. With such deferred translation, virtual caches can reduce the dynamic power consumption for TLB accesses. A recent study has shown that using virtual L1 caches opportunistically reduces more than 94% of energy consumed by the data TLBs [11]. Although the prior virtual caching studies commonly aim to reduce power consumption

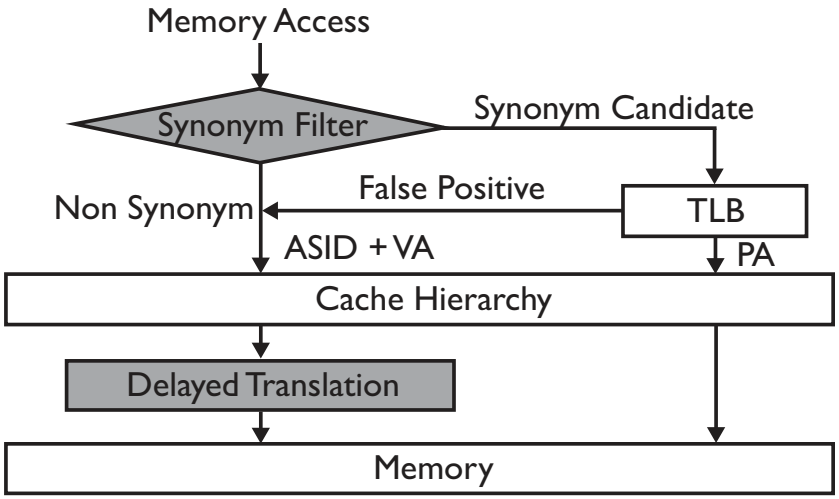


Figure 2.1: A synonym filter detects synonym candidates and allows non-synonym addresses to bypass translation until LLC miss. The components in gray are newly added by this work.

for address translation, virtual caching can improve performance too. The cached data do not require address translation and large on-chip caches can potentially have a larger data coverage than the limited TLBs. Such phenomenon was recently measured by Zhang et al. [74], and for several workloads, more than 95% of accesses that cause TLB misses were resident in the caches. For such cases, physically-tagged caches must wait for the TLB miss handling, even if the cachelines are in on-chip caches. Furthermore, by deferring address translation, the capacity of delayed TLBs can be increased as they are no longer restricted by the critical core-to-L1 path latency.

Virtual caching, despite having such desirable properties, has critical problems which have hindered its wide commercial adoption. The main problem is the M:N mapping between the virtual address space and physical address space resulting in two types of problems. The first type of problem, called *homonym*, occurs when two different physical pages map to the same virtual page of different processes. To virtual caches, the virtual page number looks identical, however the underlying physical pages are different. The homonym problem has been fixed by adding the address space identifier (ASID) in each cache tag entry, to identify the cacheline owner for the given virtual page number. The second type of problem occurs when two different virtual addresses map to a single physical address. This problem, called *synonym* or *aliasing*, is the main crippling factor of widespread use of virtual caches. We discuss the synonym problem and prior work in the next section.

2.2.2 Prior Work on Synonym Problem

Synonyms can create multiple cachelines with different virtual addresses for the same physical address. The main problem of synonyms is that if a cacheline at one virtual address is updated, the other cachelines pointing to the same physical address need to see the update, otherwise the other cachelines will have stale data. There have been several hardware and/or software approaches to resolve the coherence problem [24, 68, 39, 32]. There are two common components to support coherence for synonyms. First, a *synonym tracking* mechanism finds all cached synonym copies of the same physical block. Second, a *synonym detection* mechanism identifies whether a given address is a synonym address. The synonym tracking can also be used as a detection mechanism. The two mechanisms are used together or in isolation to address the synonym problem.

HW-based synonym tracking: A key component to maintain the coherence of synonyms is to track virtually addressed cachelines that share common physical counterparts. Once the synonyms are identified, they can be invalidated or downgraded by checking all existing copies. To track synonyms, reverse mapping is used either by adding extra address tags in virtual caches [24], or by adding back-pointers in the L2 cache [68]. Goodman proposed to use dual-tag arrays with virtual and physical tags to support coherence through physical addresses for virtual caches [24]. Wang et al. proposed to keep back-pointers in the private physically addressed L2 cache, which track synonyms in the virtual L1 cache [68].

In parallel to our study, Yoon and Sohi proposed to map synonym pages of the same physical address to a leading virtual address. It uses a hash table, much like a Bloom filter used in our work, to identify synonyms and translate to the leading virtual address before accessing the L1 virtual cache [73].

Isolating synonym pages by OS: Instead of detecting synonyms, the OS can isolate synonym pages in certain known virtual addresses. OVC uses virtual address for private data, and physical address for shared data in L1 caches [11]. This approach restricts the OS to use a fixed region of virtual address space for synonym pages. In addition, unlike our approach, OVC limits virtual caching to the L1 cache, as it aims to reduce energy, and it requires reverse translation through extra physical tags in L1s for coherence.

SW-oriented approach: Single address space operating systems use one address space for all processes, and thus the synonym problem does not occur [42]. However, protection across different processes must be supported by an extra protection lookaside buffer. An alternative software-oriented approach is to use virtually addressed caches, but during cache miss handling, the address must be translated to a physical address by a SW-based cache miss handler [32]. Although it simplifies the HW complexity for the synonym problem greatly, a slow SW handler must intervene for cache misses.

Self-invalidation: Kaxiras and Ros proposed to use self-invalidation and downgrade from virtual L1 caches [39]. Using the support from coherence protocols, synonyms in L1 caches are self-invalidated to maintain their coherence. However, a limitation of this approach is that it requires a cache coherence protocol which does not send any coherence request directly to virtual L1 caches. To classify synonym and non-synonym addresses, they use a combined HW/SW approach to mark sharing status in page table entries.

Intermediate address space: Enigma makes use of the intermediate address space, available in the PowerPC architecture, that exists between virtual and physical address spaces[74][1]. Between the core and L1, a virtual address is translated to an intermediate address through a large fixed granularity segment. Memory sharing between processes is achieved by large fixed granularity segments. The cache hierarchy uses the intermediate address space, and the address translation occurs after LLC misses. Synonyms are addressed by the first-level translation with coarse-granularity mapping. The second-level translation after LLC misses uses conventional page-based translation with TLBs. Although Enigma exploits the large on-chip caches to reduce the burden of address translation for performance, its fixed page-based translation between the intermediate and physical address spaces limits translation scalability as shown in our results section.

Benchmark	Shared area	Shared access
ferret	0.004543%	0.849976%
postgres	66.753033%	15.788043%
SpecJBB	0.328539%	0.023681%
firefox	0.754580%	0.001840%
apache	9.402985%	0.444037%
SPECCPU	0%	0%
Remaining Parsec	0%	0%

Table 2.1: Ratio of r/w shared memory area and accesses to the r/w shared regions

2.2.3 Synonym Pages for Common Workloads

Synonym pages are used infrequently for many common workloads. Table 2.1 shows the ratio of r/w shared memory pages out of the total used memory pages, and the ratio of memory accesses to shared regions over all memory accesses. The presented shared memory ratio is an average of the ratios sampled per second. From the entire PARSEC suite, only `ferret` uses shared memory regions, but the actual memory area and access frequency are limited. Other than `ferret`, the rest of PARSEC and SPECCPU2006 applications do not exhibit any r/w memory sharing. The proposed scheme treats read only (r/o) shared pages equally to private pages, as r/o shared pages do not cause any coherence problem.

In addition to the SPECCPU and PARSEC applications, we examined four additional applications that have synonym pages. Only `postgres` exhibits a large number of shared memory pages, since it allows multiple processes to share data. However, the other applications have a relatively small amount of memory sharing. Even for `postgres`, the actual memory accesses to the shared region is 16% of the total accesses. In the next section, we will exploit the lack of synonym pages in applications to design an efficient synonym detection mechanism.

A key observation for addressing the synonym problem is that as long as a unique address is consistently used to identify each physical memory block, the coherence problem does not occur. Our hybrid caching uses either virtual or physical address, and guarantees that the same unique address is used for each memory block.

[1] PowerPC provides three address spaces in the order of logical, virtual, and physical; whilst Enigma renames the virtual address to intermediate address. We name these spaces in the order of virtual, intermediate, and physical for consistency with our work.

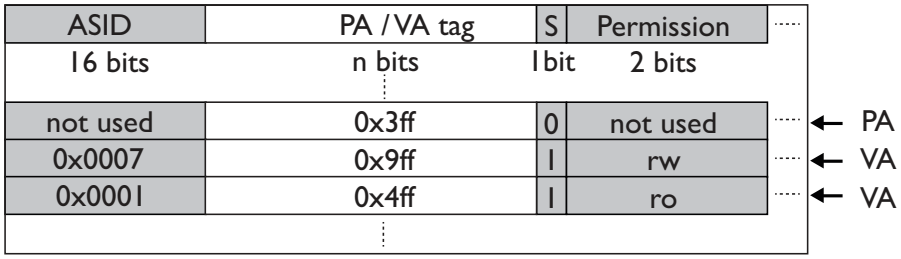


Figure 2.2: Cache tag extension for ASID and status/permission bits

2.3 Efficient Synonym Filtering

This section describes the proposed hybrid address translation architecture which allows delaying address translation for the majority of memory accesses.

2.3.1 Hybrid Address Translation

Cache Tag Extension: The hybrid virtual caching uses address space identifier (ASID) concatenated to virtual address (VA) for non-synonym pages, and physical address for synonym pages. The cache tags are illustrated in Figure 2.2. The cache tag entry contains a *synonym* bit which distinguishes a synonym (or physical address) from non-synonym (or ASID+VA). ASID bits are added to prevent the homonym problem of non-synonym cachelines. The ASID is configured to 16 bits which allow 65,536 address spaces, and such a large number of address spaces will be required for large systems and virtualized systems. For synonym cachelines, physical addressing is used, ignoring ASID. Note that the PA/VA tag portion is shared both by synonym and non-synonym cachelines, and the PA/VA tag width is determined by the maximum bits required for either virtual or physical address space. For example, AMD systems use a large 52 bit physical address space, which allows the virtual tag to fit in the physical tag. Permission bits are added for non-synonym cachelines to check access permission. The additional tag bits induce negligible overheads. The results from CACTI [52] estimates 1.7-3.7% static and 0.16-0.76% dynamic power increase for all three levels of the cache.

Address Translation: The first step of the address translation is to access the synonym filter. The synonym filter checks whether the address is a synonym candidate. An important property of the synonym filter is that it must be able to detect all synonym addresses correctly, although it can falsely report a non-synonym address as a synonym page (*false-positive*).

If an address is determined to be a non-synonym address, the ASID and virtual address are concatenated to access the L1 cache (`ASID+VA`). `ASID+VA` is used throughout the entire cache hierarchy for non-synonym addresses. Even the coherence protocol uses the `ASID+VA` address. When the external memory needs to be accessed, the `ASID+VA` is translated to the actual physical address through *delayed translation*. The delayed translation can use conventional TLBs with fixed page sizes, or variable-length segments as we will discuss in Section 2.4. Since the majority of memory accesses are to non-synonym addresses, the accesses to the synonym filter and the L1 cache with `ASID+VA` can be overlapped, hiding the synonym filter latency.

If an address is predicted to be a synonym address by the filter, a normal TLB access occurs. For a TLB hit, the address is translated to a physical address, if it is truly a synonym. If the detection

was false positive, the TLB entry for a non-synonym page will report the false positive status[2]. For the false-positive case, the address is not translated to a physical address, and the L1 cache block accessed with `ASID+VA` is used. If a TLB miss occurs, the HW page walker will translate the address and fill the TLB entry. For a false-positive caused by the synonym filter, the non-synonym TLB entry is added to the TLBs to quickly correct potential false-positives for future accesses to the address.

The proposed hybrid design does not require any reverse mapping to track synonyms. It is guaranteed that a single address (either `ASID+VA` or `PA`) is used for a physical cacheline in the entire cache hierarchy. With such a single address for each memory block, cachelines are kept coherent by the HW coherence mechanism either with `ASID+VA` or `PA`, eliminating the synonym problem entirely. With the hybrid address design, as long as the synonym filter can identify all synonym pages, its correctness is guaranteed. The pages used for direct memory access (DMA) by I/O devices are also marked as synonym pages, and they are cached in physical address.

Page Deallocation and Remap: Virtual page mapping or status changes may require selective flushing of cachelines addressed in `ASID+VA`. If the synonym status changes from non-synonym (private) to synonym (shared), the cachelines in the affected page must be flushed from the cache hierarchy, although such changes will be rare. The deallocation of a physical page from a virtual page or remapping of a virtual page also requires the invalidation or flushing of cachelines in the page.

On a modification of a virtual mapping, current systems issue a TLB shootdown which is broadcasted via inter-processor interrupts to all cores. In the hybrid virtual caching, depending on the previous state of the mapping (synonym or non-synonym), the shootdown can be directed to the per core TLB structure (synonym), or the per core TLBs and shared delayed TLB (non-synonym).

Permission Support: For permission enforcement, each cacheline holds the permission bits which are set through the delayed translation for non-synonym pages. For synonym pages, the TLB translation will check the permission status before cache accesses. For non-synonym pages, permission violation such as writes to a read-only (`r/o`) page will raise an exception. When the permission of a non-synonym page changes, the permission bits in cached copies must be updated along with the flush of the delayed translation TLB entry for the page.

2.3.2 Hash-based Sharing Detection

The proposed synonym detection uses Bloom filters [15] to detect synonym pages efficiently with a short access latency and low power consumption. Each address space has a set of two Bloom filters maintained by the operating system. The Bloom filters are stored in the OS memory region, and for each context switch, the hardware registers for the starting addresses of the Bloom filters must be set by the OS, along with the conventional page table pointer. Setting the filter registers will invoke the core to read the two Bloom filters from the memory and store them in the on-chip filter storage of the core. Woo et al. proposed using Bloom Filters to filter out synonyms to reduce extra cache lookups to find synonyms, with a pure hardware-oriented mechanism [69]. This study uses a HW and SW-combined approach.

The proposed scheme uses two filters for each address space to reduce false positives as shown in Figure 2.3. The first coarse-grained filter checks the synonym region at 16MB granularity, and the second fine-grained filter checks at 32KB granularity. The synonym filter reports a synonym candidate

[2] The page table entries need to add a single sharing bit for page mappings to mark a page sharing or non-sharing. Such information is easily accessible by the kernel and reserved bits are available for use.

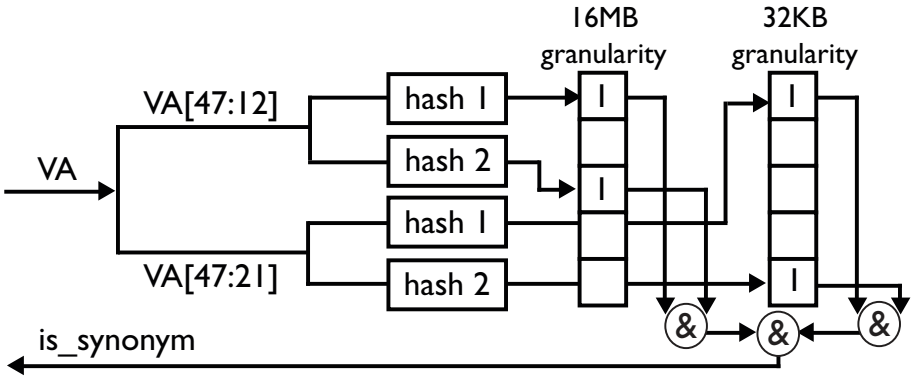


Figure 2.3: A synonym filter is composed of two Bloom filters each with different granularities. Each filter uses two hash functions. The synonym filter only returns true, thus a synonym candidate when all four bits are set to one.

only when both filters are set to true for the corresponding entry for an address. To further reduce false-positive, we used two hash functions for each Bloom filter as shown in the figure.

The Bloom filters are cleared during the creation of an address space (process). When the operating system changes the status of a virtual page to a shared one (synonym), it must add the page to the coarse and fine-grained Bloom filters. Updating the Bloom filter pair will require synchronizing all cores running the same ASID. Such a status change is rare, and it uses the same mechanism as TLB shootdowns. Note that conventional systems also require TLB shootdowns to synchronize the TLB entries on page table updates. Changing the status of a page from the synonym state to the non-synonym state does not clear the Bloom filters, since multiple pages may share the same bit. However, such changes are rare, and if such changes exceed a certain threshold and generate too many false positives, the OS can reconstruct the filter set for the process.

In this paper, we use a 1K-bit Bloom filter for both 32KB and 16MB filters. We chose the 32KB granularity, since shared pages are commonly allocated in 8 consecutive 4KB pages. The hash functions partition the address bits (trimmed by 15 bits or 24 bits according to the Bloom filter granularity) into two parts. One function partitions them by 1:1 ratio and the other by 1:2 ratio. For each hash function, 5 bits hash result is generated from a partition by exclusive-oring the bits in the partition. The two 5-bits from the partitions are concatenated into 10 bits, which is the index bit for the filter.

2.3.3 Performance of Synonym Filter

Methodology: In this section, we use a trace-based model to evaluate a long execution of applications with a large amount of synonym pages. We used the Pin tool [50] and implemented our synonym filter, TLBs, and delayed translation TLBs (delayed TLBs). We also modeled the cache hierarchy to feed only cache misses into the delayed translation. In the model, false positives are also inserted into the TLBs. We experimented on a system with Linux kernel 3.12 and glibc 2.19.

In addition to the Pin tool, we identified workloads with read-write sharing by extracting and analyzing system call traces. We investigated the five applications with synonyms shown in Table 2.1. The baseline conventional TLBs have a 64-entry L1 TLB backed by a 1024-entry 8-way L2 TLB. Compared to the baseline, the synonym TLB is a 64-entry 4-way associative single level TLB. Our delayed TLB is a 1024-entry 8-way TLB for the results in this section, to have the same overall TLB area as

	false positive rates	TLB access reduction	total TLB miss reduction
ferret	0.000756%	99.1%	20.4%
postgres	0.427410%	83.7%	-6.1%
SpecJBB	0.000019%	99.9%	42.6%
firefox	0.521944%	99.4%	63.2%
apache	0.000000%	99.5%	69.7%

Table 2.2: False positive rates, TLB access and miss reduction

the conventional system. The cache is an 8MB shared cache, and the simulations of the workloads are multi-programmed or multithreaded (depending on each workload).

Results: Table 2.2 presents the effectiveness of the proposed synonym filter with two 1K-entry Bloom filters. The second column shows false-positive access rates, due to the hash conflicts in the Bloom filters. Among all accesses, such false-positive accesses are very small, less than 0.5%. The third column shows the reduction of TLB accesses by bypassing TLBs for non-synonym pages. Except for `postgres` which has a significant amount of shared pages, all the other applications can reduce TLB accesses by 99%, reducing the power consumption significantly. Even for `postgres` with 66% of shared memory, the TLB access reduction is significant with an 84% decrease.

The fourth column shows the TLB miss reduction achieved by the synonym TLB and delayed TLB, compared to the baseline system with two levels of TLBs. Even though the total size of TLBs is equal in the proposed system and the baseline, the proposed system can significantly reduce the TLB misses by up-to 70% (`apache`). This is due to the large underlying last-level cache which filters out unnecessary translation requests for resident cachelines. The reason for the miss increase in `postgres` is due to the false positives and smaller 64-entry TLB for the synonym candidates compared to the conventional TLBs.

In this section, we investigated only the applications with shared pages, and even for such adverse applications to the proposed scheme, the results show that the synonym filtering and delayed translation work effectively. More diverse application results will be presented in Section 6.

2.3.4 Efficient Support for Read-only Sharing

As shown in the previous section, synonym pages are not common. One possible source of synonyms not evaluated in the previous section is content-based memory sharing. If a large number of memory pages become synonym pages due to content-based memory sharing, the synonym filters will become less effective, resulting in a significant increase of physically-addressed pages in caches.

However, exploiting the read-only property of such content-based memory sharing, we propose a mechanism to eliminate the need for physical addresses for the read-only (r/o) shared pages. Even if r/o synonyms exist in caches, r/o permission prevents coherence problems of r/o synonyms. As discussed in the previous section, cache tags are extended with permission bits. When the hypervisor or OS sets a page to be a content-shared page, it changes the status to read-only. The r/o permission bit is carried in all copies of the cacheline, and if a write operation occurs, a permission fault is raised. Upon receiving a permission fault, the hypervisor or OS assigns a new physical memory page, copies the content of the

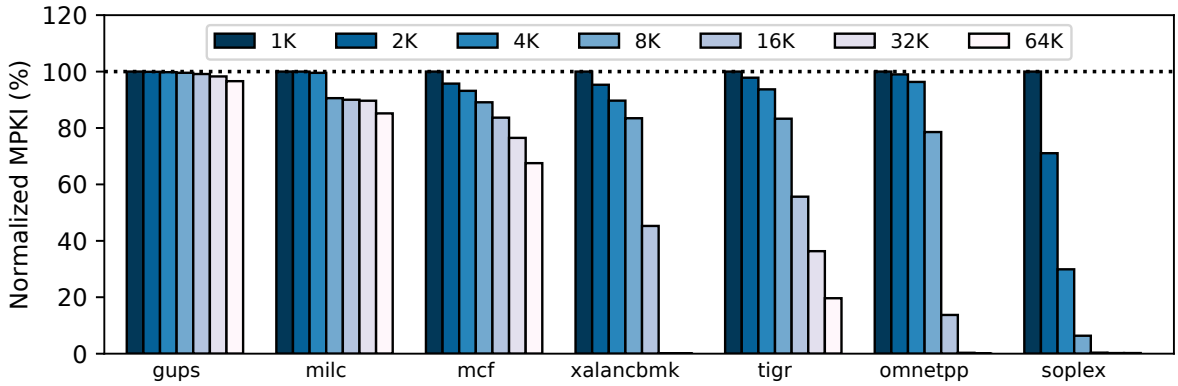


Figure 2.4: Normalized TLB miss rates (MPKI) for different TLB sizes

r/o shared page to the newly assigned page, and changes the permission of the new page to r/w, making it a private r/w page.

When the status of a page changes from non-synonym to r/o shared, all the cachelines of the page must be either invalidated or their cacheline status must be changed to read-only. However, such status changes do not trigger updates of synonym filters, and do not degrade the filtering capability.

2.4 Scalable Delayed Translation

This section presents the limitation of page-based delayed TLBs, and proposes many segment translation architecture to support many variable length segments.

2.4.1 Variable Length Segment Translation

Delayed TLB translation

For non-synonym pages, address translation occurs on LLC misses. Delayed address translation can use conventional page-based translation with a TLB lookup on each LLC miss. With large on-chip caches, cache-resident data will not cause TLB misses, even if their entries are not in the TLBs. However, such fixed granularity page-based translation, *delayed TLBs*, will eventually reach its limit as the memory working sets of applications increase. Figure 2.4 shows the limitation of fixed granularity pages for delayed translation. TLB requests are filtered by a 2MB LLC. Only LLC misses access the delayed TLB ranging from 1K to 32K entries. For `GUPS`, `mcf`, and `milc`, the increase in TLB size does not reduce the number of misses effectively, as their page working sets are much larger than the delayed TLB capacity. Even with a large 32K-entry TLB, 32 times larger than the current 1K-entry L2 TLBs, there are significant TLB misses per 1K instructions.

As the memory requirements for big memory applications increase, the delayed translation must also scale with the requirements. On-chip caches can filter out some translation requests with the hybrid virtual caching, but eventually, the translation efficiency must be improved to support big memory applications. Traditional fixed page-based translation cannot provide such scalability of address translation.

Prior Segment Translation Approaches

This section discusses the prior segment translation approaches for conventional physically-addressed caches. To mitigate the limitation of the current TLB-based translation, direct segment was proposed as an alternative to page-based translation [10]. In the direct segment, each process has a set of segment-supporting registers: `base`, `limit` and `offset`. Using the registers, a segment maps a variable length virtual memory partition to a contiguous physical memory region. With such a low complexity, a single segment support can be easily added to the existing page-based systems to allow static memory allocation of large contiguous memory. If the virtual address lies outside the segment region, traditional paging is used.

Redundant memory mapping (RMM) extends on the limitation of a single direct segment, supporting multiple concurrent segments for each process [37]. The operating system can allocate multiple contiguous memory regions with variable lengths for each process, allowing flexible memory management. Traditional paging is used redundantly to RMM. Since the address translation is still on the critical core-to-L1 path, RMM limits the number of segments to 32 operating at the latency of seven cycles, equivalent to the L2 TLB latency.

2.4.2 Many Segments for Delayed Translation

Delayed translation can improve segment-based translation by supporting 1000s of segments efficiently. Such many segment translation can provide the operating system with the improved flexibility and efficiency of memory management.

Potential benefits of many segments: Table 2.3 presents the segment counts for each application, including the segment counts produced in RMM [37]. Our analysis uses a different memory allocator library, `glibc` instead of customized `TCmalloc` used by RMM, which is the main cause of the difference in segment counts. In the table, some applications use few segments while some other applications use a very large number of segments. `Memcached` for example, requests for more memory on demand (in 64MB requests) instead of provisioning large chunks of memory.

Using a limited 32 segments may not be able to provide efficient translation when the segments are thrashing in RMM. Workloads such as `tigr, xalancmbmk` and `memcached` caused considerable MPKIs (segment misses per 1K instructions) for 32 segments in our experiments. If a segment miss occurs, either a SW or HW segment walker must fill the segment in RMM.

Another inherent issue of segment-based translation is the low utilization of eagerly allocated memory regions. Instead of the widely used demand paging, segment translation uses *eager allocation*, which allocates contiguous memory segments immediately on application request. Eager allocation increases the contiguity of allocated memory to reduce the number of resulting segments, but may cause internal fragmentation. The final column of Table 2.3 shows the utilization of segmented memory regions. Although many workloads use most of the allocated regions, four applications do not utilize 17-75% of their allocated memory. Reservation-based allocation can be used to handle such cases by reserving a large contiguous segment, but internally dividing the segment into smaller segments [67]. Only on actual accesses, the smaller sub-segments are actually allocated to the process. Adjacent sub-segments can be merged as they are promoted from reserved to allocated. However, reservation-based allocation requires more segments to support the reservation functionality.

Benchmark	RMM [37]	Reproduced	MPKI	Usage(%)
astar	33	16	0	96.5
mcf	28	4	0	72.5
omnetpp	27	106	0.02	99
cactus.	70	56	0	83.2
GemsFD.	61	143	0	100
xalancbmk	N/A	244	0.13	96.5
canneal	46	22	0	84.7
stream.	32	16	0	24.7
mummer	61	3	0	100
tigr	167	90	22.93	99.5
memcached	86	839	0.08	100
NPB:CG	95	5	0	100
gups	62	7	0	99.9

Table 2.3: Maximum number of segments in use by each application, RMM MPKI, and memory utilization

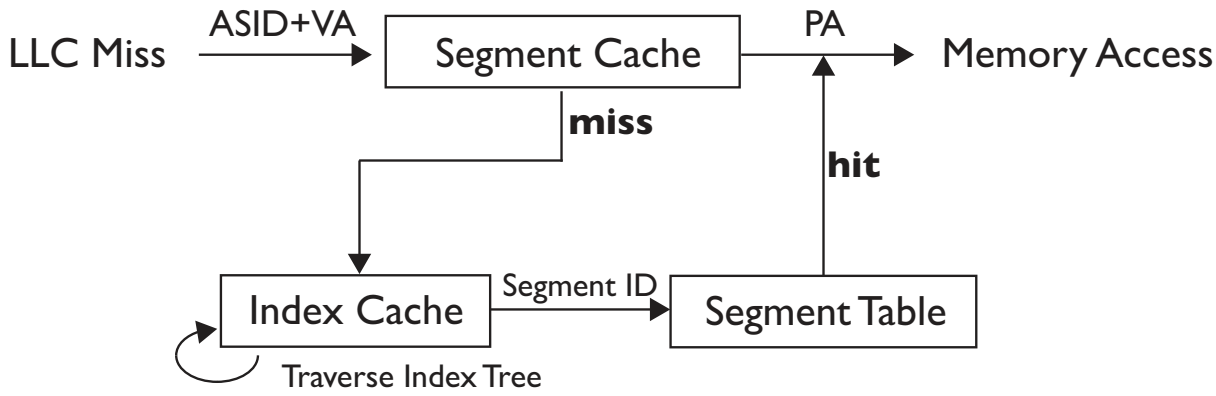


Figure 2.5: Scalable delayed translation: The overall translation flow from ASID+VA to PA

2.4.3 Many Segment Architecture

Figure 2.5 presents the overall translation flow of the proposed many segment architecture after an LLC miss occurs. The translation mechanism consists of *segment table*, *index cache*, and *segment cache*. Figure 2.6 shows the internal organization of segment table and index cache.

Segment Table: The OS maintains a system-wide *in-memory segment table* that holds all the segments allocated by the OS. Each segment entry has the starting `ASID+VA` address(`base`), `limit`, and `offset`. The table is indexed by the *segment-ID* and holds 2K segment entries, as shown in Figure 2.6. A hardware structure, *segment table*, mirrors the in-memory segment table. If an incoming `ASID+VA` address is not covered by the segments in the segment table, an interrupt occurs and the operating system fills the table entry. However, segment misses occur only for cold misses, as the size of HW table is equal to the in-memory segment table size to simplify implementation.

Index Cache: One of the key design issues is to find the corresponding segment for an incoming `ASID+VA` address from the segment table efficiently. Unlike TLBs with fixed mapping granularity, segments have varying sizes which complicate the lookup mechanism. A naive way of searching for a segment is to serially search all entries, which look ups all table entries in the worst case. Since the translation latency is important, we propose a hardware-based efficient search mechanism backed by the operating system.

The operating system maintains a B-tree indexed and sorted by the `ASID+VA` for all segments in the segment table, called an *index tree*. Each B-tree node has a key to compare the incoming address, and pointers to the nodes of the next level (the value). The resulting value, which may exist on the leaf or on intermediate nodes, is an index to the segment table, or the *segment-ID*. Thus, a traversal through the index tree yields the segment-ID of the segment that the incoming address belongs to.

For every LLC miss, the index tree must be accessed to find the segment-ID which points to the corresponding segment in the segment table. Since accessing the in-memory index tree for every LLC miss is not feasible with long latencies, a hardware cache for the index tree, *index cache*, stores recently accessed entries from the index tree. For each LLC miss, in the worst case, the index cache must be accessed by the number of the tree depth to reach the leaf node containing the requested segment-ID.

The index cache is a regular cache of 64 byte blocks addressed by physical address. The index tree nodes are cache block aligned. Each node contains six keys (starting address of a segment) and seven values (pointer to the next level node, or the segment-ID). 1024 and 2048 segment entries can fit in an index tree with depth of four, when spanned by a factor of seven (seven values). A hardware walker

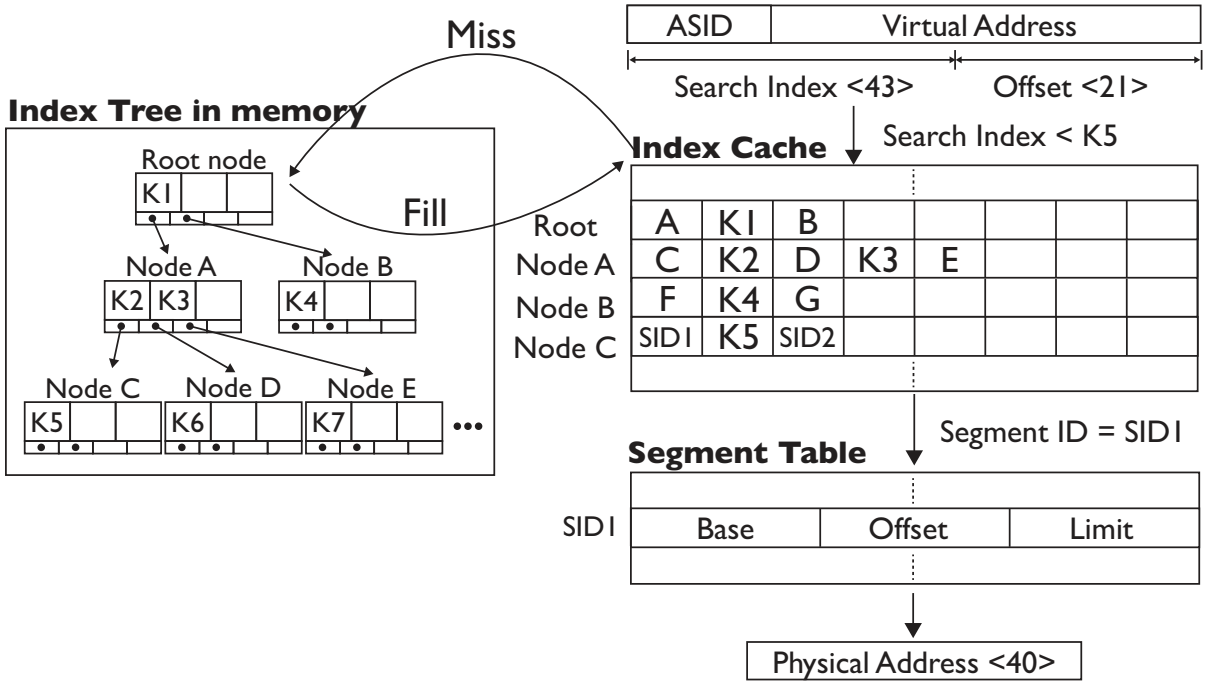


Figure 2.6: Organization of segment table and index cache

will bring in a cache block from the memory and will compare the incoming address to all six keys in parallel. The leftmost pointer which satisfies the comparison (address>key) will be used for the next node lookup.

Segment Cache (SC): To hide the latency of accessing the index cache and segment table, the delayed translation can be done simultaneously with LLC accesses. Although the parallel accesses to the delayed translation and LLCs can improve the performance, such parallel accesses can increase the energy consumption for delayed translation unnecessarily. To reduce the energy overhead, an alternative way is to access delayed translation serially only after LLC misses. However, to further reduce the latency overhead of serial delayed translation, this architecture adds a small 128-entry segment cache. The segment cache is a simple TLB-like component with 2MB granularity. For an SC miss, the translation results from the segment table will be used to fill the fixed granularity SC entry for next accesses.

Translation Flow: As shown in Figure 2.5, the incoming address from an LLC miss checks the 2MB granularity segment cache (SC) first. If it hits SC, the translation is completed. For a miss, the translation traverses the index tree by looking up tree nodes. The segment-ID resulting from the index tree traversal is used to index the HW segment table to find the segment information. Using the segment entry, the address is checked against the base and limit values of the segment. Finally, the offset value is used to translate the incoming `ASID+VA` to PA.

2.4.4 Index Cache Efficiency

We conducted three sensitivity studies on the index cache size. The first study investigates single-threaded applications, and the second one for four threads of applications modeling a multi-programmed quad-core system. To stress the index cache, we artificially broke a segment into 10 segments, adding the effect of external fragmentation. Without the added external fragmentation effect, a smaller size

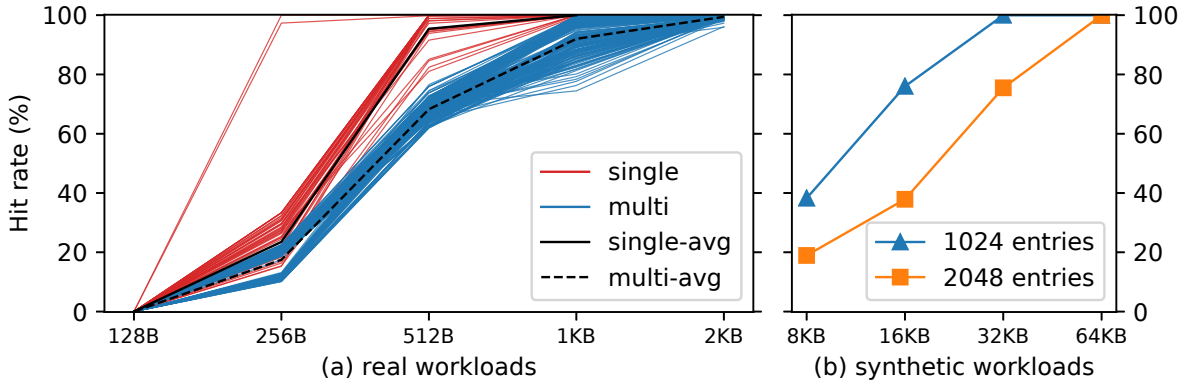


Figure 2.7: Index cache size sensitivity study. (a) shows index cache hit rate for actual workloads. (b) shows synthetic worst case benchmark.

of index cache will be good enough for many real applications. The last study investigates the worst case scenario, where all accesses are randomly issued. The results from the three studies are shown in Figure 2.7. The index cache is an 8-way associative cache ranging in size as labeled in the x-axis.

The single and multi-core application simulations were conducted using the Pin-based simulation of workloads described in Section 2.6.1. The multi-core evaluation was conducted by interleaving four Pin traces (quad core system). Ten applications causing most misses were chosen, and a total of 210 quad-core mixes were generated and executed. The lighter lines show the miss curves of single thread applications, and the darker lines show multi-threaded applications. The multi-threaded workloads cause more conflict misses compared to single applications, thus the curves show slightly more misses for the same index cache size. The access patterns of real applications exhibit locality, thus the index cache does not suffer misses even with a modestly sized index cache of 8KB.

For the worst case, we distributed the physical address space of 40 bits to 1024 or 2048 segment entries equally. We inserted all entries into the index tree, and simulated one million random accesses to the entire physical address space. This workload shows no spatial locality, and is thus the absolute worst case. Even in this worst case, 32KB index cache can almost eliminate index cache misses for 1024 segment entries, and provide 75.5% hit rates for 2048 segment entries.

Using CACTI 6.5 [52], we estimated the access latency of the index cache. For a 3.4GHz machine, both 32KB and 64KB 8-way index cache have the latency of 3 cycles. Also the access latency of the segment table of 2048 entries is seven cycles [3]. Thus we can estimate that four accesses or less to the index cache (four level index-tree) followed by an access to the segment table is about 19 cycles. Based on the analysis, we assume that the delayed many-segment translation takes 20 cycle.

To support 2048 segments, the segment table size is about 48KB (base, offset, length), and the index cache size is 32KB. Compared to the LLC size, the extra 80KB structure does not add a significant area overhead. Furthermore, a multi-core processor needs to have only one index cache and segment table shared by multiple cores.

[3]The segment table is configured to use low standby power configuration. All others use the high performance configuration

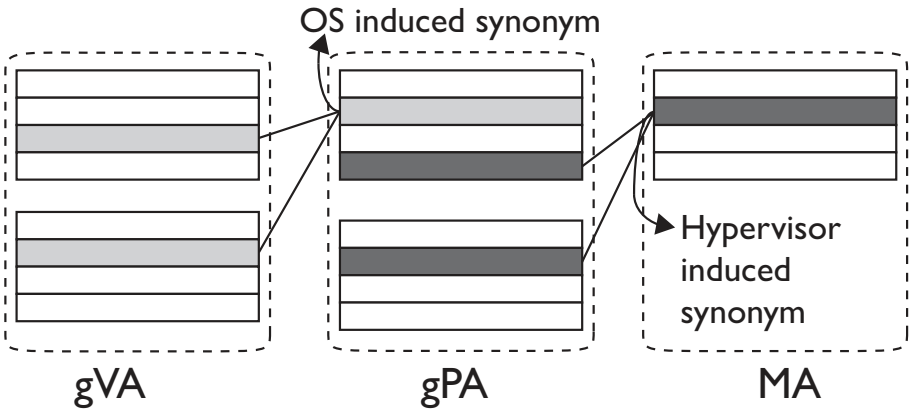


Figure 2.8: Difference of synonyms incurred by the OS and hypervisor

2.5 Virtualization Support

The overall translation architecture for virtualized systems is similar to non-virtualized systems. For non-synonym pages, the ASID must include the virtual machine identifier (VMID) in addition to the process ID within a VM. With the ASID extension, a VM cannot access virtually-addressed cachelines of another VM, since their ASIDs do not match.

2.5.1 Extending Synonym Detection

In virtualized systems, a guest virtual address (gVA) is translated to a guest physical address (gPA), and the guest physical address is translated to a machine address (MA). The guest operating system maintains the mapping between the guest virtual and guest physical address spaces, and the hypervisor maintains the mapping between the guest physical and machine address spaces. In non-virtualized systems, the synonym filters are maintained by the operating system. However, for virtualized systems, both the OS and hypervisor can cause synonyms for memory pages. In addition to OS-induced synonyms, the hypervisor can make two different pages in the same VM or different VMs share a physical page. Figure 2.8 illustrates how two types of synonyms differ.

To address the additional hypervisor-induced synonyms, the synonym filters are updated both by the OS and hypervisor. Similarly to the two dimensional page walks supported by recent x86 architectures, the OS and hypervisor maintain different filters, the `guest` and `host` filters. For a guest process context switch, the guest OS switches the guest filter as in a native system. For a VM context switch, the hypervisor switches the host filter. When looking up the synonym filter, both filters are looked up. If either one of the filters reports a filter hit, the accessed page is identified as a synonym candidate. The subsequent flow for synonym and non-synonym pages are identical to native systems.

One of the challenges for the synonym detection is that the filters must be set in gVA for both the guest and host Bloom filters, since both filters are looked up using the guest virtual address. When a guest virtual page becomes a synonym due to a hypervisor-induced sharing, the hypervisor is responsible for setting the host filters by the guest virtual address.

When the hypervisor changes a guest physical page to a synonym page (shared page), it traces the corresponding guest virtual addresses in the virtual machine, and updates the host filter content. To facilitate the process, the hypervisor may maintain an inverse mapping from gPA to gVA for each virtual

machine.

On a special case where the guest OS changes the mapping of a guest virtual page from a private guest physical frame to a hypervisor-induced shared physical frame, the guest is not aware of the guest physical frame being shared by the hypervisor. Thus, it is the responsibility of the hypervisor to mark the newly mapped gVA as a synonym. Our solution is simple for the case. For such remapping of a guest virtual page, the guest OS must flush the corresponding cachelines anyways, regardless of the hypervisor-induced synonym problem. Access to the newly mapped page causes an LLC miss, causing delayed translation. The address will be identified as a synonym address during the two dimensional walk, and an exception will be raised for the hypervisor to handle the newly identified synonym.

2.5.2 Segment-based 2D Address Translation

The current full virtualization of the x86 architecture uses two separate page tables in the guest OS and hypervisor to map the virtual memory to the actual system memory [12]. A hardware two-dimensional page walker walks both guest and hypervisor page tables and fills the TLB with the translation from gVA to MA. One of the benefits of the hybrid virtual caching for virtualized systems is that it hides the cost of the two-dimensional translation. By removing the translation from the critical core-to-L1 path, much of the delayed translation can be filtered by LLC hits. For the proposed hybrid virtual caching, a similar 2D page walker will be used for the page-based delayed translation, which will walk the guest and host page tables for translation cache misses.

To support full virtualization for many segment translation, segment translation must be supported for guest and host segments, which are governed by the guest OS and hypervisor respectively. The guest OS can update only the guest segments. Full virtualization incurs two overheads if it is done before L1 cache accesses as in RMM [37]. First, segment-based translation needs to be done twice for guest and host segments. The two steps are required because the hypervisor cannot guarantee to allocate equally sized physical segments for every guest OS segment allocation requests, and may serve a guest OS segment allocation with multiple host segments. Second, the per-core segments limited in number are further divided into guest and host segments, reducing the number of segments available for each user process.

The proposed many segment translation does not suffer from the limited number of segments for guest and host segments. The additional latency incurred by the two dimensional translation can be overlapped with LLC accesses to mitigate additional latency at the expense of minor extra energy consumption. However, in this study, to reduce energy consumption, we employ serial accesses to the LLC and segment translation. To further reduce the additional latency, a 128-entry segment cache (SC) is used to store direct translation from gVA to MA for 2MB regions, skipping the gPA.

2.6 Experimental Results

2.6.1 Methodology

To validate the performance of the proposed system, we used MARSSx86 [56] + DRAMSim2 [62] which is a cycle accurate full system simulator running a linux image, with an accurate DRAM simulator. Table 2.4 shows the simulated system configuration. For the baseline system, we modeled the TLBs after the Haswell architecture of Intel [30]. The evaluated experiments run SPECCPU2006, Graph500 (sized 22), NPB benchmarks (C sizes, and B size for `NPB_DC`), and `tigr` of the BioBenchmark suite [6].

Parameter	Value
Processor	Out-of-order x86 ISA, 3.4GHz 128-entry ROB, 80-entry LSQ 5-issue width, 4-commit width 36-issue queue, 6-ALU, 6FPU
Branch Predictor	4K entry BTB, 1K entry RAS Two-level branch predictor
L1 I/D Cache	2/4-cycles, 32KB, 4-way, 64B block
L2 Cache	6-cycles, 256KB, 8-way, 64B block
L3 Cache	27-cycles, 2MB, 16-way, 64B block
TLB L1	1-cycles 64 entry, 4-way, non-blocking
TLB L2	7-cycles 1024 entry, 8-way, non-blocking
Memory	4GB DDR3-1600, 800MHz, 1 memory controller

Table 2.4: Simulated baseline system configurations

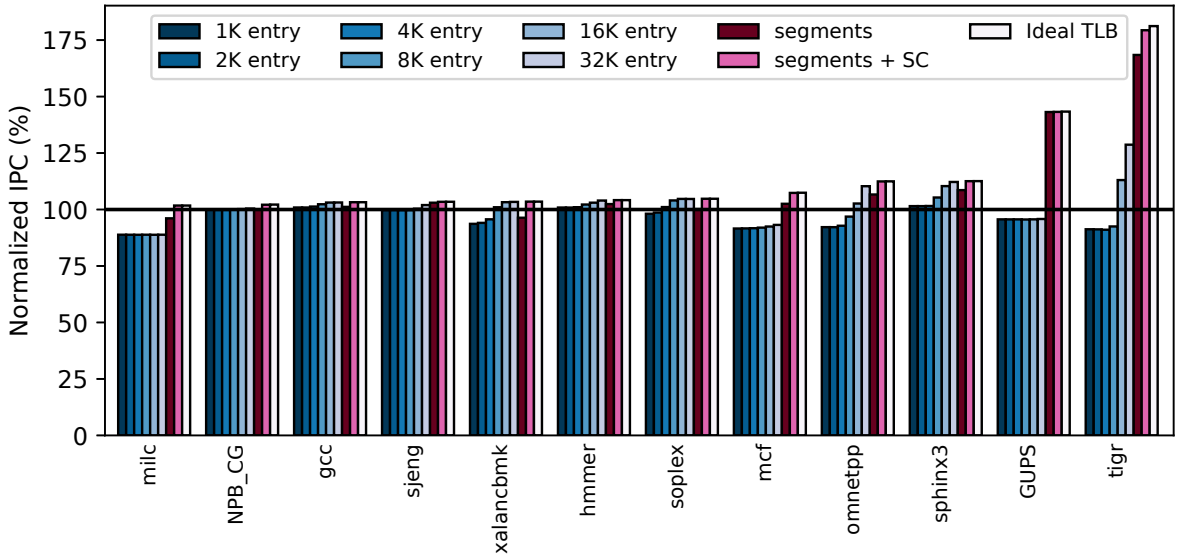
Additionally we conducted experiments of GUPS (with size 30), a random access benchmark. One billion instructions were simulated for evaluation. However, for `tigr` we had to reduce the number of instructions executed to 500 million, due to the elongated simulation time resulting from a very low IPC.

2.6.2 Performance

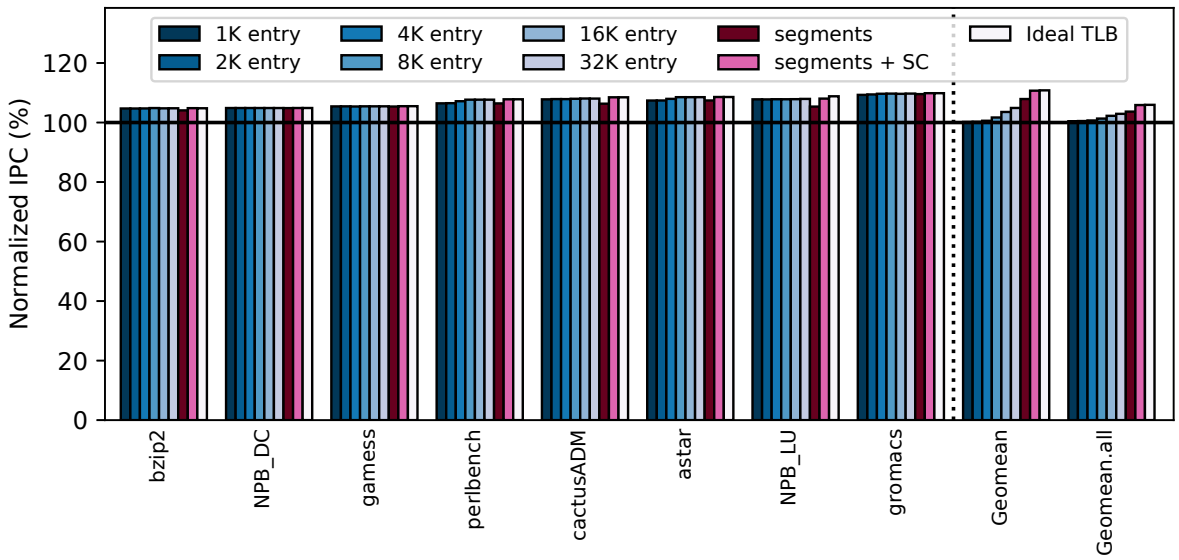
Native Systems: In Figure 2.9, the first experimental results show the performance improvement of delayed translation in non-virtualized systems. The results were normalized to the baseline system. The evaluated configurations are the baseline, fixed granularity delayed TLB translations varying the TLB size from 1K to 32K entries (henceforth labeled as delayed TLB), the delayed many-segment translation (without and with 128-entry segment cache), and finally the ideal TLB performance. The ideal TLB depicts the potential performance of a system without TLB misses. The graph is divided into three parts by the dotted vertical line. For brevity, only the workloads sensitive to the different configurations are plotted. The remaining workloads are aggregated in the `Geomean.all` plot.

The Figure 2.9b shows the workloads which benefit directly from delayed translation of virtual caching. These workloads efficiently utilize the cache hierarchy, and delayed translation effectively removes unnecessary translation requests which occurred in the baseline system. The performance of these workloads is higher than the baseline, and consistent for different sizes of delayed TLB, as the 1K TLB can absorb most of the delayed translation requests.

For the workloads in Figure 2.9a, a naive TLB-based delayed translation causes performance re-



(a) Workloads that favor the scalable delayed translation



(b) Workloads that benefit from delaying translation

Figure 2.9: Normalized performance of our proposed system to the baseline system. Workloads on the left are workloads which benefit from delayed translation. The results in the center show workloads in which fixed granularity delayed translation causes overhead but improves as more delayed TLB entries are provided.

duction. These applications exhibit significant LLC misses, and delayed translation adds extra latencies for each miss. Most of the applications in the center part suffer from a certain performance reduction with the delayed TLBs, compared to the baseline. The reason for performance degradation is that the delayed TLB miss handling can take longer latencies than the conventional TLB miss handling in our conservative model of delayed TLBs. In conventional TLBs, a TLB miss handler (page table walker) can access L1, L2, and L3 caches to fetch the required page table entries during page walks. Among the accesses, a significant number of accesses are L1 or L2 cache hits. However, in the delayed TLBs,

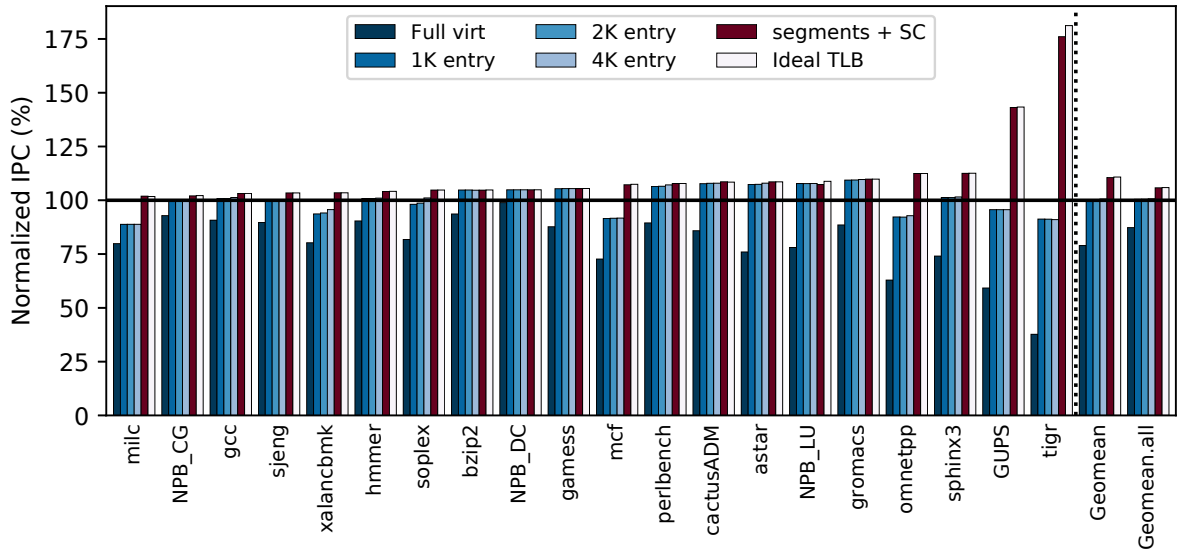


Figure 2.10: Performance of full-virtualized baseline and delayed translation normalized to the baseline native system

we conservatively assume that the page table walker can access only the LLC (L3), since the TLB miss handler is located along with the LLC miss handler. Therefore, even if the TLB miss rates are equal in the baseline and delayed 1KB TLBs, the performance degradation can occur with the delayed TLBs. A simple remedy for this problem is to use a translation caching scheme which caches non-leaf tree entries of multi-level page table for delayed translation. For the performance comparison in this section favoring the baseline, we did not evaluate the translation caching support for the delayed TLBs.

For a subset of workloads such as `sjeng`, `xalancbmk`, `hmmer`, `soplex`, `omnetpp`, and `sphinx`, the performance is improved for larger delayed translation TLBs. However, `milc`, `mcf`, and `GUPS` still suffer from performance drops, as the larger TLBs cannot cover the working sets. The results are consistent to the Pin-based study presented in Figure 2.4. For the workloads with high performance drops even with large TLBs, our proposed many segment translation shows its potential. Many segment translation with a small segment cache (SC) can almost eliminate the cost of delayed translation. Across all the workloads evaluated, the performance with many segment translation with SC matches the performance of ideal translation. For `GUPS` and `tigr`, the performance improvements are 43% and 79% compared to the baseline. On average, the many segment translation scheme improves the performance of selected workloads by 7.9% and many-segment with SC shows an average of 10.7% performance gain. Although not shown in the figure, parallel accesses to the many segment translation and LLCs can provide near ideal performance for all the applications, 0.1% less than the ideal runs. For the selected workloads, parallel accesses achieve an average of 10.8% performance gain compared to the baseline.

Delayed TLB exploits the LLC to filter away unnecessary TLB accesses, improving TLB caching efficiency. TLB miss reduction from the baseline 1K-entry TLB to the delayed TLB of 1K-entry is 99.3% on average. In other words, 99.3% of L2 TLB misses on conventional systems can be filtered away if the translation is delayed to after an LLC miss. `GUPS`, `tigr`, and `mcf` have lower TLB reductions of 45.5%, 61%, and 76.4%, respectively. The workloads above show random access patterns that are not effectively cached in the LLC, resulting in lower TLB reductions.

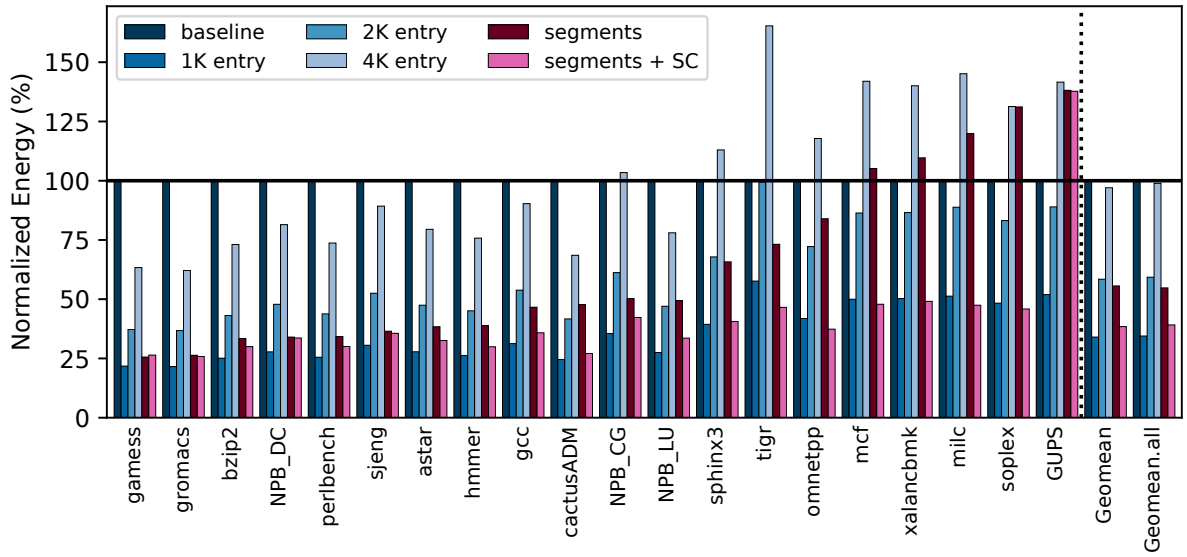


Figure 2.11: Normalized power consumption of delayed translation over baseline system (power consumption by translation components)

Full Virtualization: The second experimentation is to evaluate the performance of fully virtualized systems. We modeled an x86-like fully virtualized system, denoted as `full virt`, and delayed TLB translation which does two dimensional page walks on a miss with TLB sizes ranging from 1K to 4K entries. The 2D many segment translation is denoted as `segments + SC`. The ideal TLB performance is also shown. The results are normalized to the native non-virtualized system performance.

Full virtualization incurs page walk overheads as up-to 24 memory accesses are required, instead of four in native systems. To shorten the page walk latency, commercial architectures support translation caches to cache non-leaf translation entries of multi-level page tables [31, 12]. For the experiment in this work, we have added a page walk cache (PWC) to speed up the page table walks for the `full virt` system. However, with the conservative assumption on our own technique, PWC is not used for delayed TLB translation.

Figure 2.10 shows the performance result of virtualized systems. The fully virtualized system shows significant performance drops due to the overhead of two-dimensional page walks. Delayed TLB improves performance, and our many segment translation (with 128-entry segment cache) shows the best performance, close to the ideal system. As with the native results, the delayed TLB shows a slight performance improvement with more TLB entries for some workloads, while other workloads such as `milc`, `GUPS` and `mcf` do not benefit from larger TLBs. Many segment translation overcomes such limitations. On average, many segment translation performs 18.5% better than the current fully virtualized architecture for all workloads, and 31.6% better for the selected workloads.

2.6.3 Energy Consumption

Figure 2.11 presents the dynamic and static power consumption of translation components of proposed mechanisms normalized to the baseline TLB system. The power models are generated from CACTI 6.5 [52]. Delayed translation with TLBs and many segments can achieve reduction in overall power con-

sumption, because the power consumed for accessing the synonym filters[4] is lower than that for accessing L1 TLBs, and the number of accesses to the delayed TLB is smaller than that to conventional TLBs, because most of accesses are filtered by the LLC.

Delayed many segment translation generally has lower power consumption compared to the delayed TLB configurations due to the higher static power consumption of larger delayed TLBs. The delayed TLB structures that are larger than 2K entries consume more static power compared to the index cache and segment table, resulting in higher overall power consumption over the delayed many segment translation. However, for the workloads with high LLC misses, delayed many segment without SC consumes more power than even the baseline system.

Using a segment cache reduces the power consumption significantly, as it reduces accesses to the index cache and segment table, and also achieves modest performance gains as observed in Figure 2.9. The segment cache miss rate is 0.05%, effectively buffering multiple lookups to the index cache and segment table. One exception is `GUPS`, which incurs a high segment cache miss rate (of 96.5%), as the workload has inherently random access patterns. Other noticeably high miss rates were observed in `tigr`, `NPB_LU`, and `NPB_CG` with miss rates of 31.3%, 12.4%, and 10.8%, respectively.

Summary: For many common workloads with low LLC miss rates, hybrid virtual caching can reduce the translation power consumption significantly, as it can eliminate most of the conventional TLB accesses. It can improve the performance too, if the LLC can contain cachelines which could have missed the conventional TLBs. However, for the applications with high LLC misses, the many segment translation can increase the power consumption due to increased accesses to the index cache and segment table. However, although translation power consumption increases for such cases, the performance of the applications can potentially improve significantly with a much more effective translation mechanism than the conventional system, if the traditional TLBs cannot provide a sufficient translation coverage for the applications. The aforementioned two scenarios show the main advantage of the proposed techniques. The hybrid virtual caching can bring either power, performance, or both benefits by virtual caching and delayed translation depending on application TLB and LLC miss behaviors.

2.7 Discussion: Avoiding cache shootdowns

Aside from synonyms, one of the difficulties associated with employing virtual caching for the entire cache hierarchy is the cost that is accompanied with page mapping changes. Page mapping changes are where the operating system needs to change the mapping from virtual memory page to physical frame. In current systems, page mapping changes require merely flushing affected translation entries cached in the TLB and MMU page walking caches. This is because, the TLB is looked up before cache accesses in conventional physical address caching systems. Thus, with conventional systems it is sufficient synchronize the mapping change by just flushing the TLB.

However, with Hybrid Virtual Caching the entire cache hierarchy holds virtual address cache lines, and to provide the permission checking offered by the traditional TLB, the virtual cache lines need to hold access permission bits. This is illustrated in Figure 2.2. Whenever the page mapping is modified, whether it be mapping location changes (i.e. remapping the page from one physical frame to another, or even freeing pages) or page permission changes; the affected cache lines along with the TLB entries

[4] Due to restrictions of the CACTI system, the modeled synonym filter reads in byte granularity instead of bits granularity which causes over estimation of dynamic power per filter access. We expect optimized hardware implementation to be able to conserve more power per access by accessing the synonym filter by bits.

at the delayed TLB, all need to be synchronized. Henceforth, we call this act of flushing cache lines to synchronize with the mapping change as *cache shootdowns*. In this section we first identify the what causes such mapping modifications. Then we look at how we can support mapping modifications for each of the modification triggers.

2.7.1 Causes for Page Table Modifications

In this work we address three main types of mapping modifications that require synchronization for the sake of correctness.

- **Freeing back memory to the OS:** user space applications voluntarily return pages back to the operating systems.
- **Protection Changes:** permission changes are frequent when loading programs, and also utilized as guard pages, and even just in time compilations.
- **Memory Migration:** with heterogeneous memory, NUMA systems, and even compaction for large pages, memory migration is frequently invoked.

There are some other mapping modification operations. Two examples are access bit clearing, and dirty bit clearing. These operations are currently used by the OS to manage idle pages to distinguish active and inactive pages, and dirty bits are cleared after successful write backs of the dirty page to the backing media. Both access and dirty bit clearing need the corresponding cache lines to be flushed for precise use of the access/dirty bits. This would be especially important for the dirty bit clearing, as after a page write back, the dirty bit must be reset when an consecutive store is performed on the page. However, if a cache line with a dirty bit is not flushed, writes to the cache line will not cause the appropriate dirtying of the bit in the page table.

Thus, for the bits of the page table (access, dirty) cache shootdowns are required. However, If we do implement system with virtual cache hierarchies, the placement of access/dirty bits will need to be rethought. Will these bits need to be replicated in the cache hierarchy? Or will all stores need to propagate to the TLB in the background? These questions open up opportunities for future research, and we leave it to future work. We instead focus on the three types of mapping modifications iterated above. As cache flushing in addition to TLB flushing is undesirable, we explore possibilities of forgoing cache shootdowns.

2.7.2 Handling returning pages to the OS

During the lifetime of a process, the process will request for memory, and return memory that it no longer needs. Finally when the process finishes, all the memory that the process held onto is returned back to the OS. Even during the execution of a process, the developer may release mappings that are no longer needed (via `munmap()`). This operation may also occur without the knowledge of the developer, as memory allocation libraries such as jemalloc, libc, and tcmalloc return memory to the OS via `madvise(MADV_DONTNEED)`, `munmap()`, `brk()`. At such points when the process do return memory to the OS, the OS will unmap the returned page from the process address space. In a conventional system, at this point a TLB shootdown is triggered to synchronize the mapping updates. With a naive implementation of virtual caching, one can choose to trigger a cache shootdown with the TLB shootdown.

Another option is to prevent reusing the virtual memory address space. That is, once the memory is mapped once, it is never re-used again. The virtual memory address can be re-used after an explicit shootdown of the entire cache hierarchy for the address space identifier (ASID) of the given address space.

To understand why preventing the re-use of virtual address space, we use a simple example. Assume that there is a mapping to a virtual page number (VPN) `0xA`. The process returns this page back to the OS. At this point, there are dirty cache lines of the VPN `0xA`. Soon after, the process requests for memory and the OS reuses the VPN `0xA`. At this point, the kernel will zero out the page, however as the kernel uses the kernel address space to clear out the page, the dirty cache line from the previous mapping are still dirty in the cache. Afterwards, the process accesses the newly allocated VPN 0xA. At this point, if the dirty cache lines are evicted and write back to VPN 0xA, then we are involuntarily corrupting VPN 0xA.

Thus, we choose to disable virtual page re-use. We mark the used pages as 'expired' and do not reuse the pages. Any dirty cache lines, when evicted, will lookup the TLB before writing the blocks to DRAM. At this point, the TLB lookup will resolute that this page is no longer valid. The eviction of the dirty page will be skipped. Therefore, a TLB flush, and marking the page table entry of the freed page table as expired is required, however a cache flush can be evaded.

There is a circumstance of not re-using the virtual address space: depletion of the virtual address space. The user level virtual address space of current 48bit machines are 2^{47} bits in Linux, and that equates to 128 TiB. On systems that have enabled the 5-level page table [29], the user space jumps up to 2^{56} bits and 64 PiB. In such cases, depleting the address space will take a long time. However, to design a system, such corner cases have to be dealt with.

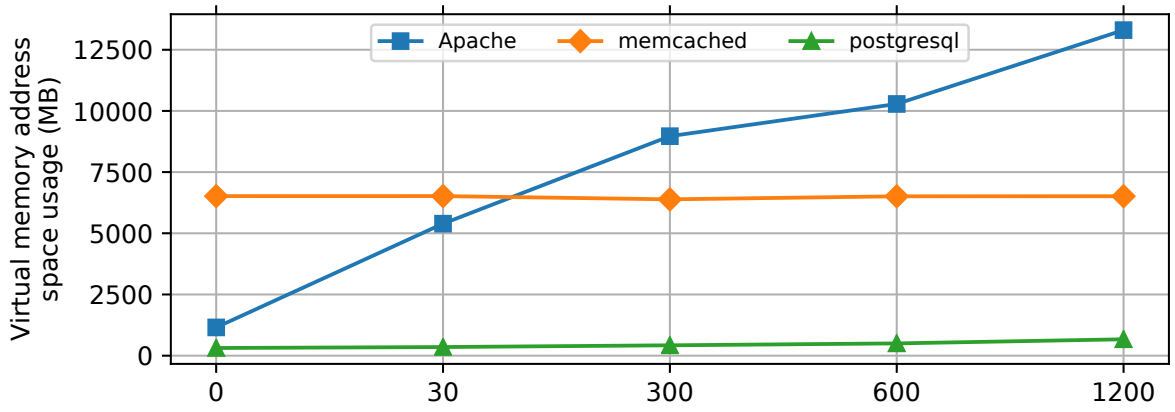


Figure 2.12: Virtual address space usage of long running applications in seconds of elapsed time

Applications such as SPECCPU, biobench, parsec suite are short lived, and do not use make so much mapping requests. For this analysis we take a look at long lived server applications; `apache`, `memcached`, and `postgresql`. For postgresql we used the `pgbench` benchmark driver [60], and we use the cloudsuite twitter dataset for memcached [19]. Finally, we used `wrk` benchmark to drive apache httpd server [23]. The total amount of virtual memory space used is plotted in Figure 2.12. Postgresql and memcached do not show significant increase in virtual address space usage. Using simple linear regression, we estimate the postgresql and memcached to require 15 years, and 638 years, respectively, to deplete the address space. Apache on the other hand does use the virtual address aggressively, as all incoming queries result

in the file to be served being mapped and unmapped after servicing the request. We estimate about 308 days for apache to deplete the address space. These estimated times are for the 48bit virtual address space, and with the 5 level page table, the depletion time is increased by 512x.

However, when the address space is depleted, the system needs to be able to reclaim and reuse the expired virtual address area. To do so, we propose flushing the entire cache hierarchy for the address space. As the reason we are not reusing address space is due to the danger of lingering cachelines of previous mappings, if we can flush the entire cache hierarchy, we can prevent such corruption issues. This approach is a trade off between cache shootdowns for every memory unmappings vs. an entire cache hierarchy flush once every year or so.

2.7.3 Handling protection changes

The virtual memory system allows giving different permission to different pages. On any permission change to a page, the TLB entry of the corresponding page needs to be synchronized, requiring a TLB shootdown. With virtual caches, as the permission bits are embedded into the cache lines, in a basic implementation the cache shootdown is also required.

However, it is worth noting that TLB and cache shootdowns are not required for permission upgrades (i.e. read-only to read-write), as these page faults will be naturally handled by the page fault handler. The shootdowns are only required for permission downgrades, or cases where the permission becomes less permissive than before.

Some examples of permission changes are listed below. Page guards for thread stacks require permission downgrades to read-write-execute permissions. Making a region read-only will also downgrade a region from read-write. Finally, Just in time (JIT) compilation may also need a mapping change from read-write to read-execute. Some JIT engines actually allocate the memory region for the compiled code as read-write-execute, however as more OSes are adopting the W^X policy [2, 1], permission changes for JIT will be necessary

As an illustrative purpose, in the remainder of this section we use the JIT case as an example. Making read-write pages read-only follows the same steps. The steps that are required for permission change is as follows. First, the read-write region is created. Second, data is copied into the region. Third, `mprotect()` is called on the region to change the permission to read-execute. On the third step, TLB shootdown is issued, and for basic virtual cache implementations, the cache shootdown is required at this point.

To prevent causing cache shootdowns for permission downgrades, we can use shared mappings to deliberately cause a synonym situation. First, a shared mapping of read-write permission is made. Second, data is copied into the region. Third, the shared mapping from the first step is mapped (to a new virtual space) as read-execute. Fourth, the first mapping is unmapped, and the third mapping is used by the process to execute code.

It is critical to note that a synonym is created by the two different virtual address mappings to the same physical address. Therefore, At the end of step 2, the cache lines holding the copied data of step 2 need to be written back to the main memory. This can be done by `clflush`, `clflushopt`, or `clwb` in x86 architectures, however this is identical to doing a SW based cache shootdown. Instead, we can use non-temporal move instructions when copying the data in step 2. The Intel manual states that "the processor does not write the data into the cache hierarchy" [31]. By using the non-temporal move instruction, we can copy the data into the main memory without keeping dirty cache lines in the cache

hierarchy. However, it is also critical to enforce the ordering of the actual non-temporal memory copy (step 2), and the actual accessing/consuming of the data in step 4.

To explore the performance characteristic of various permission change methods, we evaluated the latency of mapping, copying and changing permission of one 4KB page. The results are plotted in Figure 2.13. The `traditional` approach is the `mmap`, `memcpy`, and `mprotect` method. `Traditional + clflushopt` is the traditional approach with an extra cache shootdown, where the cache shootdown is emulated by invoking SW `clflushopt` instruction. The remaining five series show the shared-mapping approach discussed as our proposal. The `not safe` series does not flush the cache, and is only shown for comparison with the traditional approach. `Memcpy + clflush/clflushopt/clwb` are the methods that use shared mapping and temporal memory copy with explicit cache shootdowns. The `clflush` instruction performs bad due to the ordering between the 64 cache flush operations. The optimized `clflushopt` and `clwb` instructions show much better latency. Finally the right-most bar shows our proposed shared-mmap using non-temporal memcpy. It is worth noting that the proposed scheme performs better than the traditional mmap memcpy mprotect + cache shootdown scheme. Our proposal performs slightly less than the current mmap-memcpy-mprotect scheme, however it performs the best for virtual cache schemes.

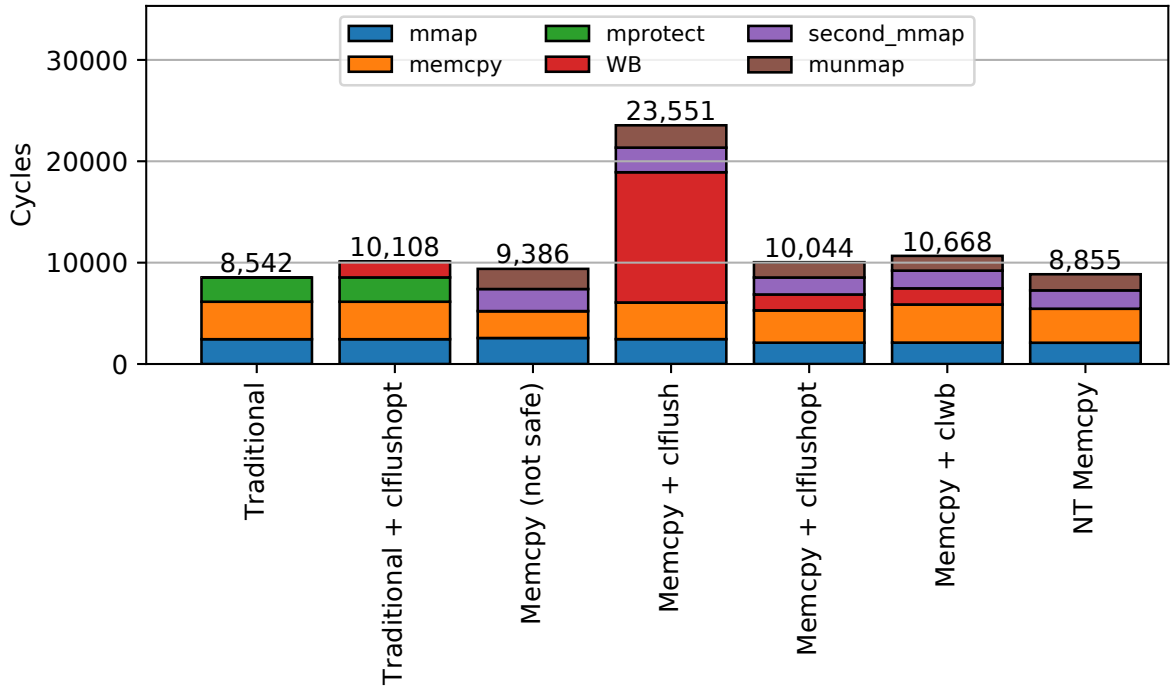


Figure 2.13: Performance of various JIT mechanisms to fill out the executable memory region

2.7.4 Handling memory migrations

Finally, we discuss supporting memory migration while avoiding cache shootdowns. Memory migration is the process of changing the backing physical mapping from one physical frame to another. The virtual address within the page being migrated stays the same. This provides a interesting opportunity for virtual caching. In physical cached systems, when a page is being migrated, the corresponding address translation misses in the TLB and is paused by the page fault handler. Thus, the load/store to the cache

lines of the page being migrated is paused during the migration. With virtual caching, for all cached lines, the load/store can keep using the cache lines in the cache. However, any misses, or evictions will need to wait as with the conventional system.

The steps taken during a page migration is as follows. First, the page mapping is discarded from the page table and the TLB shootdown occurs. Second, the page is locked to prevent subsequent access to this page by other threads. Third, the data is copied from the old physical frame to the new physical frame. Fourth, the page table is updated to point to the new physical frame, and the lock is released.

For a basic virtual cache implementation, the cache shootdown needs be issued at step 1. However, to allow virtual cache to enjoy the aforementioned advantage of being accessible during migration, we propose using non-temporal memory copies in stage three and skipping cache shootdowns. The implication of this approach is that any cached lines in the cache hierarchy can be accessed during the migration, while the cache misses or write backs due to eviction need to stall similarly to the migration on conventional systems. Furthermore, as we use non-temporal memory copy we can skip the cache shootdown. Thus the non-temporal memory copy provides two benefits for virtual caching.

To examine the performance characteristics of the current migration and proposed scheme, we studied the effective memory copy bandwidth of moving a single 4KB page, 2MB page, and 1GB page (for the un-cached scenario). The results presented in Figure 2.14. The `cached` series show the situation where the pages are cached in the cache hierarchy, whereas the `not cached` series shows the case where we explicitly flushed the pages prior to invoking memory copy. The first scheme is the current memory copy method. The second scheme is the current memory copy followed by a `clflushopt` to illustrate a basic virtual cache system. Finally, we show the non-temporal memory copy. The results show that the non-temporal copy always outperforms the basic virtual cache scheme (temporal copy + cache shootdown). Furthermore, for pages that are not cached, non-temporal copy outperforms the baseline copying scheme. This also shows that the memory copying in tiered memory may need to be rethought, as cold pages being migrated to large but slow memory may benefit from non-temporal copy, and at the same time prevent from polluting the on-chip caches.

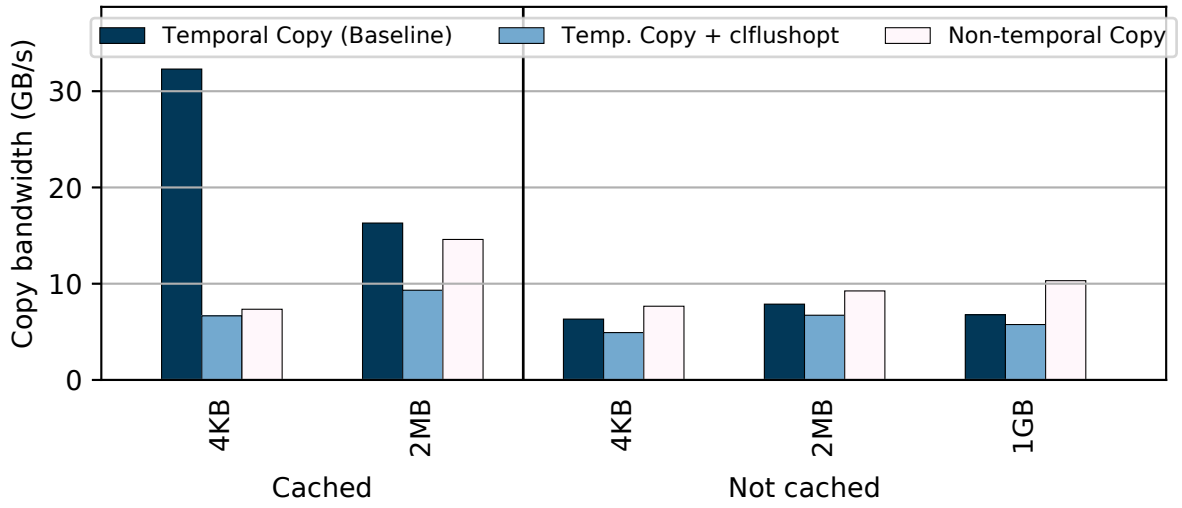


Figure 2.14: Performance of temporal and non-temporal migration

To summarize, page mapping changes need to be addressed when systems employ virtual caching. In some cases cache shootdowns may be necessary. In this work we have explored some scenarios where

cache shootdowns can be averted, and we have proposed schemes that can outperform the virtual cache system that uses cache shootdowns. In this work we propose making use of the non temporal move instructions to tame the otherwise dangerous synonym problem. We permit synonym mappings to be made, however by making sure that the data is moved via non-temporal instructions and enforcing the access to the data is made only after the non-temporal move is finished. Such disciplined approach allows systems with virtual caches to perform correctly in the presence of synonym mappings and still perform relatively well.

2.8 Conclusion

This study proposed a new memory virtualization architecture with hybrid virtual caching. The key component enabling the efficient virtual caching is the synonym filter which can identify synonym pages efficiently with few false positives. With the synonym filter, a physical cacheline can exist in caches only with a single name either by its physical address (synonym case) or by its `ASID+VA` (non-synonym case). By extending hybrid virtual caching to the entire cache hierarchy, the proposed mechanism reduces both energy consumption and translation latency. However, as the memory requirements increase, even delayed translation with page-based TLBs cannot scale effectively. This paper proposed a many segment translation architecture for better translation scalability and more flexible memory allocation by the OS.

Chapter 3. Increasing TLB Coverage with *Hybrid TLB Coalescing*

3.1 Introduction

With ever increasing memory capacity demands for large memory applications, address translation for virtual memory support has become a critical performance bottleneck of the applications. To improve the address translation efficiency, there have been two different approaches. The first approach, *coverage improvement*, expands the translation coverage of the translation looakaside buffer (TLB) within a given area and power budget [59, 58, 37, 38, 10, 17, 54, 55, 74]. The second approach, *TLB miss penalty reduction*, decreases miss handling latencies after memory requests miss in the TLB [13, 8, 9, 53, 63, 35]. Although both approaches are important for translation performance, enhancing translation coverage can provide direct performance improvements, as the latency can be completely hidden for TLB hits.

To improve the translation coverage, there have been a spectrum of techniques employed in commercial systems or proposed in recent studies. One of the most common approaches is to increase the page size. In commercial x86 architectures, 2MB and 1GB page sizes are supported in addition to the traditional 4KB page size. Increasing the page sizes dramatically improves the coverage of a TLB entry by 512 times with a 2MB large page. However, to fully exploit the improved coverage, operating systems (OS) must be able to allocate a 2MB chunk of physical memory for each page. A more radical way of improving the translation coverage is to use variable-sized HW segment translation instead of page-based translation [37]. Its effectiveness also relies on whether the OS can allocate a very large contiguous memory chunk for each segment. An alternative to large pages and segments is HW-based coalescing techniques [59, 58]. *CoLT* and *cluster TLB* coalesce multiple page translations into a single TLB entry as long as their physical locations are in a contiguous region. Unlike segments or large pages, the pure HW-oriented techniques opportunistically find the contiguous pages and coalesce them to a single TLB entry. The operating system can improve the coalescing chances by allocating contiguous pages, but it does not need to guarantee certain chunk sizes.

However, these approaches have trade-offs in two aspects: *allocation flexibility* and *scalability of translation coverage*. Supporting large pages allows only limited numbers of page sizes, which restricts the scalability of coverage. Segments have the highest scalability of coverage, as they can support variable-length translation with virtually no limit. However, memory allocation requires a much stricter discipline, applicable only when a limited number of very large chunks of physical memory can cover the entire memory footprint. On the other hand, HW coalescing allows flexible allocation, but coverage scalability is limited to four to eight pages, as coalescing must be purely done by the HW components.

A common requirement of the prior techniques is that the OS must be able to consistently provide contiguous memory chunks suitable for each technique. However, recent studies show that such contiguous chunk allocation is not always possible or can even degrade the performance of multi-socket NUMA systems [22, 43]. In addition to the common NUMA architectures, the emerging new memory architectures, such as 3D stacked DRAMs, network-connected hybrid memory cube (HMC), and non-volatile memory (NVM) , can further increase the non-uniformity in memory [49, 57, 61, 18, 51, 36]. Such memory heterogeneity requires fine-grained memory mapping to place frequently accessed pages on fast near memory, complicating the allocation of large contiguous memory chunks.

Due to the non-uniformity in memory architectures, the OS cannot always provide the best allocation tuned for different coverage improvement techniques. Even for the same application, the actual memory allocation status changes drastically depending on the system state, which incurs severe performance variation [43]. One technique may work well with a certain memory allocation scenario, but may not work well, if the OS cannot provide the optimal allocation for the technique. Therefore, an ideal coverage improvement technique needs to be able to adapt to diverse memory allocation states. In this paper, we propose a hybrid address translation technique adaptable for diverse memory allocation scenarios, while improving the translation coverage whenever possible. The hybrid technique utilizes the high-level mapping information available to the operating system, and requires minor changes in translation architectures, using mostly the same TLB structures and page tables.

The new translation architecture, called *hybrid coalescing*, encodes the contiguous allocation information in a subset of page table entries, called *anchor entries* designated at every N page table entries. The anchor entry must contain how many following pages are allocated in contiguous physical memory. For an L1 TLB miss, if the requested page address is not in the L2 TLB, the nearest anchor entry for the page number is searched in the L2 TLB. If the requested page is part of the contiguously allocated memory region, the anchor entry can provide the translation by simply adding the virtual address difference between the anchor and requested pages, to the physical page address of the anchor entry. The rationale behind the anchor-based translation is that if the majority of application memory is allocated in some distributions of contiguous chunks, most of the TLB entries will be filled with anchor entries. Each anchor entry can cover a large portion of memory translation.

The proposed translation techniques can adapt to various memory allocation scenarios, since the OS can change the density of anchor entries in page tables. If memory allocation can provide mostly contiguous chunks of memory, the anchor entries are sparsely located. If the memory allocation is fragmented to small chunks, the anchor entries are densely populated. Using the flexible anchor density, hybrid coalescing can exploit the memory allocation contiguity as much as possible, even if the contiguity states are diverse. As the OS encodes the contiguity information in anchor entries, the proposed scheme can eliminate the HW overheads of finding and coalescing contiguous pages in HW-based coalescing techniques. Furthermore, without the HW restriction, the proposed technique can vastly increase the translation coverage of each anchor entry.

This paper is the first study to propose a HW-SW hybrid TLB coalescing, providing both scalable coverage and allocation flexibility. The proposed technique has the following strengths over the prior approaches. First, it can dynamically change different chunk sizes for translation coalescing to adapt to the currently available contiguity in memory allocation. Second, unlike fixed large pages, the OS does not need to provide a strict fixed chunk allocation. Hybrid coalescing can extract the available contiguity as much as possible, even if a certain fixed contiguity is not provided. Third, the proposed scheme can support a highly scalable coverage improvement, as the contiguity is encoded in the page table. Finally, the changes to the current TLB and page table structures are minor.

In the experimental results, the paper shows that under various memory allocation scenarios, the proposed scheme can provide the best performance consistently. The proposed scheme outperforms or performs similar with the best prior scheme for each mapping scenario, achieving the best average performance across diverse scenarios.

The rest of the paper is organized as follows. Section 2 discusses challenges posed by address translation and memory heterogeneity. Section 3 describes the proposed hybrid translation techniques, and Section 4 discusses how to find the best distance for anchor placements. Section 5 presents the

	THP	Cluster/CoLT	RMM	Our Approach
Scalability	Mod.	Mod.	Good	Good
Flexibility	Mod.	Flex.	Restr.	Flex.

Table 3.1: Comparison of scalability and allocation flexibility (**Mod.**: moderate, **Flex.**: flexible, **Restr.**: restricted)

experimental results. Section 6 discusses the related work and Section 7 concludes the work.

3.2 Motivation

3.2.1 Translation Coverage Improvement

The first approach to improve the translation coverage is to use multiple page sizes. In the current x86 architectures, 2MB and 1GB page sizes are supported in TLBs. Applications may explicitly request for large pages during memory allocation, or may make use of the transparent huge page (THP) support, the operating system can assign 2MB pages, if 2MB chunks are available. Using a couple of different page sizes does not incur significant complexity in the TLB designs, and thus, the latest architecture can support both 4KB and 2MB pages in the L2 TLBs without requiring separate TLBs for each page size, although the 1GB pages use a separate and smaller 1GB page L2 TLB. However, a disadvantage of large page sizes is that its coverage is still limited with only a few possible page sizes, and the scalability of its coverage will be eventually limited. Furthermore, the OS must always assign a fixed large chunk to benefit from the translation coverage improvement.

To drastically increase the translation coverage, the second approach uses variable-sized segments. Direct segments and Redundant Memory Mapping (RMM) support one or multiple segment regions of variable length [10, 37]. For each segment region, the operating system must allocate a contiguous chunk of memory. As long as such contiguous memory allocation is possible, the translation coverage of a single segment can scale to a very large region of virtual address space, practically eliminating much of the address translation costs. However, as the number of HW segment translation entries is much smaller than the current TLB size due to the fully associative range search required for segment translation, each process can only use a limited number of segments at a time. RMM supports 32 segment translation entries (*range TLB*) to match the latency of the L2 TLB [37]. Furthermore, for its effectiveness, segment-based translation relies on a very strict huge chunk allocation.

The third approach is the HW-based coalescing technique. As proposed by CoLT and clusterTLB, there are some levels of contiguity in memory allocation as the operating system uses a buddy algorithm to reduce memory fragmentation [59, 58]. In the HW coalescing techniques, the HW TLB controller searches the page table entries and finds contiguously allocated pages. Since a cacheline contains multiple page table entries, the logic can efficiently search through multiple page table entries without issuing separate memory accesses. Although this approach does not rely on the strict allocation of contiguous pages, as the HW controller exploits the contiguity opportunistically, the scalability of translation coverage is quite limited. To allow efficient lookups of coalesced entries in TLBs, they support only a limited coalescing capability of 4-8 pages in a TLB entry. CoLT additionally provides a fully associative mode that supports

a much larger number of coalesced contiguous pages. However, it requires a fully associative lookup, which in turn restricts the number of entries available.

The three approaches have different trade-offs in their HW cost, allocation flexibility, and scalability of coverage. Large page supports have the smallest extra HW cost, but the allocation flexibility and coverage scalability are moderate. The segment-based translation has the highest coverage scalability, but requires a strict memory allocation with some extra HW for the direct segment registers. RMM requires a fully associative range search, which severely limits the size of the range TLB. The limited range TLB restricts the OS to allocate only a very limited number of huge contiguous memory regions for each process. Finally, HW coalescing allows flexible memory allocation with fine-grained memory mapping, but the coverage scalability can be the most limited among the three approaches. Table 3.1 presents the comparison of the prior techniques.

3.2.2 Increasing Non-Uniformity in Memory

Although improving translation coverage requires allocation of contiguous memory chunks, increasing non-uniformity in memory architectures within a system can pose a challenge for using large pages. The majority of datacenter systems use multi-socket NUMA nodes for better system density. In the NUMA systems, thread scheduling can often mismatch the memory locations the threads are accessing. Gaud et al. showed that that using 2MB large pages often degrades the performance of multi-threaded applications, and thus large pages must be selectively used to prevent causing unnecessary remote memory accesses [22]. Kwon et al. further investigated the unfair allocation of large pages [43]. In real systems, the availability of large page allocation can fluctuate significantly, and processes often receive large pages inconsistently, causing performance variations. Both studies show that allocating even moderate sized 2MB pages is not trivial in real systems, when memory non-uniformity exists.

Furthermore, such memory non-uniformity is expected to increase in future, with the advent of 3D stacked DRAM, network-based hybrid memory cube (HMC) [49, 51, 57, 40], and non-volatile memory (NVM) [61, 18, 36]. With the emerging memory technologies, the memory hierarchy can change to multiple levels of memory with different latency and bandwidth characteristics.

The recent studies show that the capacity of stacked memory is large enough to be part of the main memory, and thus the physical address space consists of fast-near memory and far-slow memory regions [51]. In a generic system model for such hierarchical main memory, the operating system uses the virtual-to-physical mapping using page tables to assign either near or far memory pages to the virtual memory space of applications [51]. The HMC architecture provides multiple memory modules connected by on-chip networks [40]. In such networked memory systems, latencies for accessing different modules vary, increasing the non-uniformity of memory access times.

Emerging non-volatile memory also accelerates the heterogeneity of memory. NVM is projected to be slower (in terms of latency and bandwidth) compared to DRAM [61, 18, 36]. However, they are also expected to provide a higher density along with the non-volatility characteristic. Coupling these characteristics, DRAM can be used as a SW cache for hot pages, and NVM as a large backing memory for cold pages. Agarwal and Wenisch proposed an application-transparent page management that makes use of the non-uniformity and different benefits of DRAM and NVM [3]. Such multi-level memory requires fine-grained page mappings to fully exploit its potential benefits. For example, within a 2MB virtual page, only part of the page can be accessed frequently. Storing the entire 2MB page in the near memory can waste the precious near memory space.

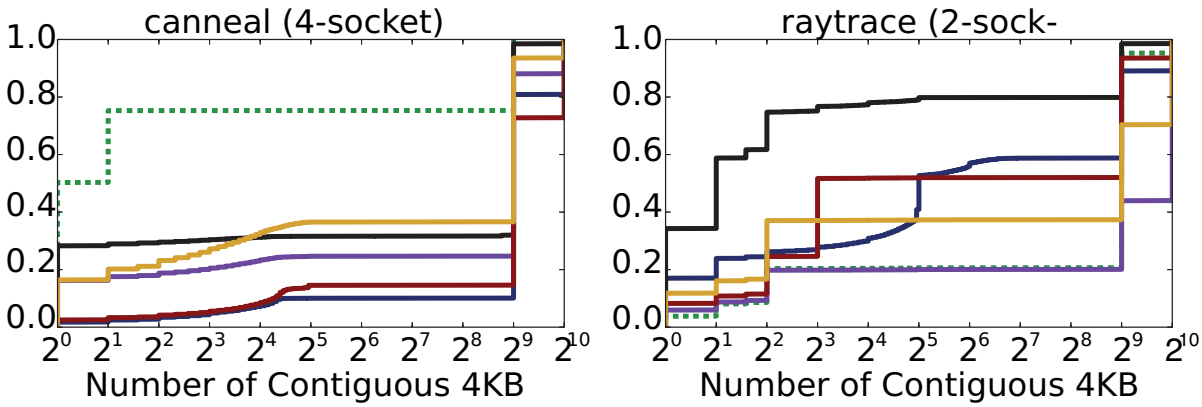


Figure 3.1: Cumulative distributions of chunk sizes in `canneal` and `raytrace`

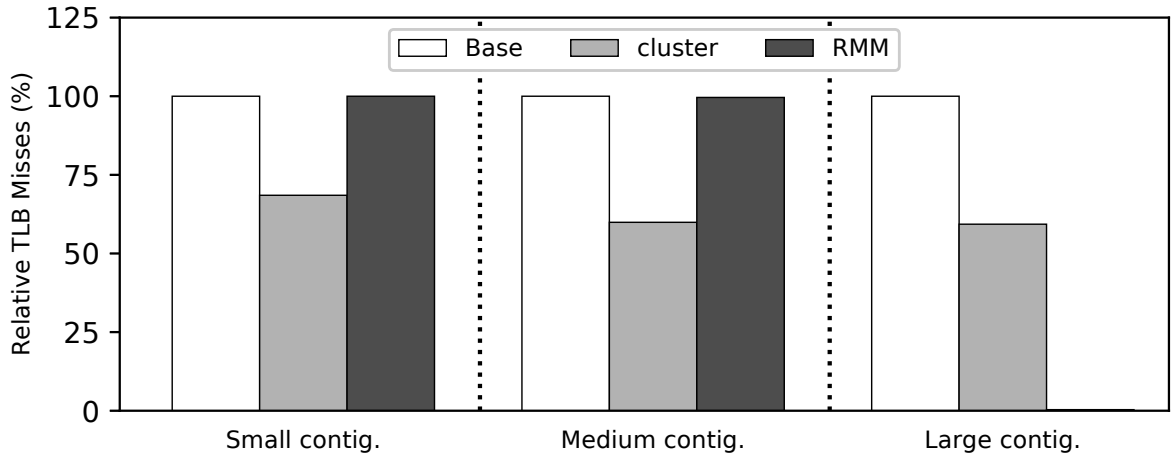


Figure 3.2: Relative TLB misses of prior techniques with three different mapping scenarios

3.2.3 The Effect of Memory Allocation Diversity

This section shows the variance in memory allocation with real systems using the PARSEC benchmark suite. We recorded the mapping contiguity on two different x86 machines, two and four socket NUMA machines, each running Linux 3.16.0 and 3.19.0, respectively. To change the memory mapping status for each run, we executed the workload of interest alone or with randomly executing background jobs chosen from PARSEC. The number of concurrent random background jobs was controlled to pressure the system memory while preventing any memory swapping. We periodically took memory map snapshots, and for each execution, analyzed the memory map at its largest allocated state.

Figure 3.1 shows the CDF of two different workloads running on the two and four socket systems. The x-axis is the number of contiguous 4KB pages. The dotted line is the memory mapping when the workload was running alone, while the other lines are the multiple executions with other random background jobs. The figures show a wide variation of memory contiguity when the memory allocation states are varied by the different co-runners or by the system configurations. Without clear patterns, the results confirm that the allocation contiguity varies somewhat randomly based on the initial state and how memory requests are generated from multiple processes.

As the contiguity diversity shows, even the same application running on the same server may receive different memory mappings. Thus, designing a system that works on a specific mapping may not perform as expected when the mapping distribution changes. The same phenomenon, of significant contiguity difference, was also observed by Cox et al., where they found that running an application with `memhog` [59] of differing intensity affected the memory allocation contiguity of an application [16].

Figure 3.2 presents our evaluation of two prior schemes at different contiguity distributions. The details of the contiguity distributions and configurations are explained in Section 3.5.1. Cluster TLB (`cluster`) effectively reduces TLB misses for the small chunk configuration, but `RMM` is not effective as the small number of range TLBs cannot cover many small chunks. However, for the large chunks, the benefit of `cluster` is almost similar to that with the small chunk configuration, not getting much more improvements from increased contiguity. On the other hand, `RMM` can almost eliminate TLB misses with the large contiguity configuration.

As the memory mapping of the process is subject to variability, it is necessary to design a translation scheme that performs well in different mapping situations. In this work, we propose a hybrid TLB coalescing mechanism that aims to provide an efficient translation for different types of memory mappings.

3.3 Hybrid TLB Coalescing

3.3.1 Anchored Page Table

Large pages, segments, and HW coalescing can expand the coverage of address translation from the limited HW resources. However, as discussed in Section 2.1, the three approaches impose different restrictions on the memory allocation flexibility on the operating systems, while providing different levels of coverage scalability. In this section, we re-balance the role of the operating system and architectural components of the prior approaches to support flexible memory allocation, while supporting better coverage scalability than HW-coalescing.

In our approach, instead of relying on the HW logic to identify contiguous pages, the operating system uses its own memory allocation information to record the contiguity status to part of page table entries. The required HW changes to the TLB lookup logic is minimized by using mostly existing components in current MMUs. To record the contiguous chunk information, every N page table entries are designated as *anchor* entries. The anchors are placed on entries aligned by N. N is the distance between two adjacent anchor entries. Each anchor entry contains how many following pages are contiguously allocated, starting from the anchor entry. The anchor entry functions as a regular page table entry as well as the anchor point.

Figure 3.3 presents an anchored page table. In the example, the anchor distance N is 4, and thus every 4th entry is designated as an anchor entry (`A` entry type in the figure). The anchor page table entry uses unused bits to record how many following pages are contiguously mapped, as shown in Figure 3.4. The first anchor entry at the virtual page number 0x40 in Figure 3.3 has two pages that have been consecutively allocated (contiguity = 2). The contiguity counts are maintained by the operating system. If memory pages are newly allocated, relocated, or deallocated, the operating system must update the contiguity information in the corresponding anchor entry, in addition to updating the entry for the page. The anchored page table uses the same page table organization as the conventional ones, and only encodes the extra contiguity information using the unused bits in a subset of page table entries.

The anchor distance N should be determined to reflect the memory contiguity status of the process.

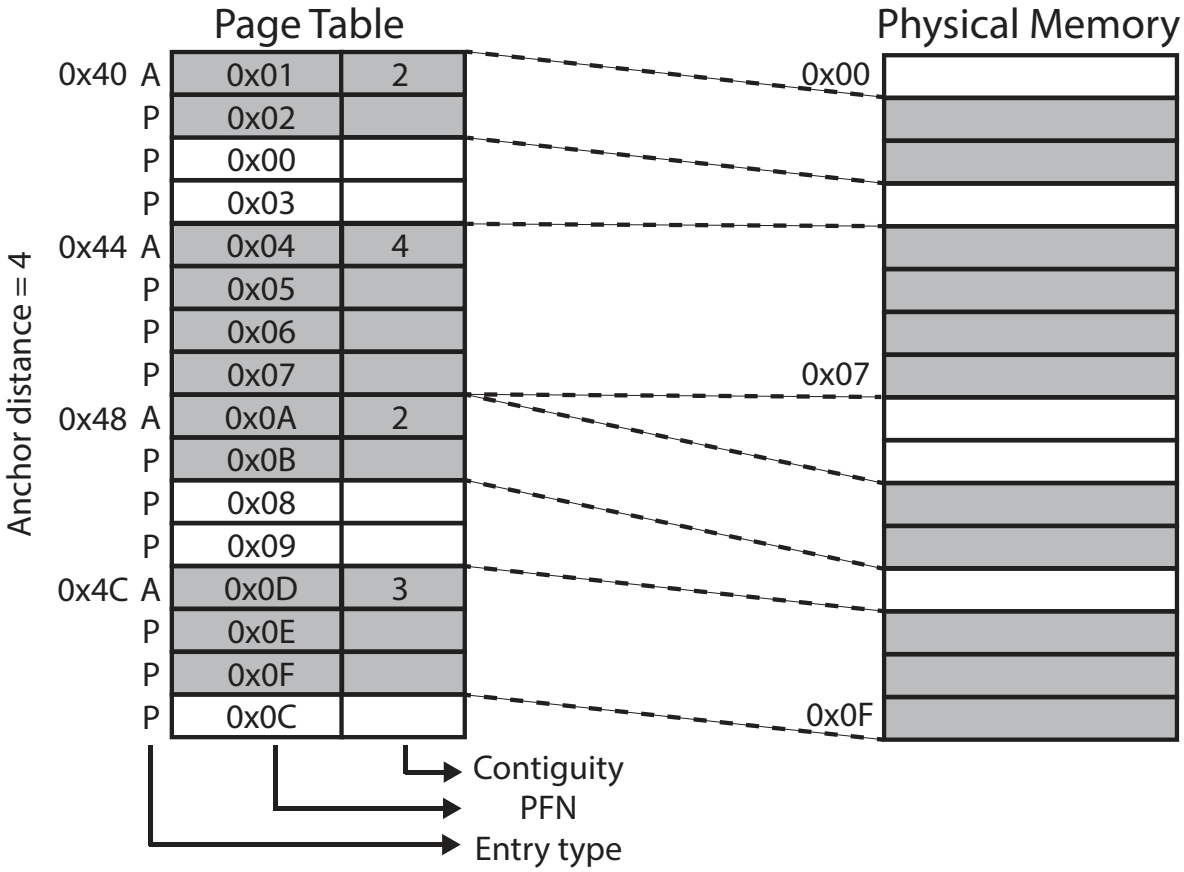


Figure 3.3: Anchor entries marked in a page table (anchor distance = 4). Each anchor entry maintains mapping contiguity starting from the anchored address

For example, if all memory pages are allocated in 64KB chunks, the optimal anchor distance is 16 ($64KB/4KB$). However, real memory allocations consist of various different chunk sizes, depending on the memory allocation behavior of the application, the OS allocation scheme, and the system memory configuration along with its current fragmentation status. We will discuss an OS algorithm to find the best anchor distance in Section 4. The anchor distance must be added to the context information of each process, along with the page table pointer (CR3 in x86). For every context switch, the anchor distance should be restored to the anchor distance register.

Figure 3.4 shows the details of page table entry used for normal and anchor entries. The anchor entry is also a regular page table entry, but it uses unused bits to store the contiguity information. To store the contiguity field in the TLB, each TLB entry may need to be increased slightly compared to the conventional ones, which do not need to store the unused bits of page table entry in the TLB.

However, in order to make anchor TLB future proof, and to provide sufficient scalability, we propose distributing the contiguity among multiple page table entries. Page table entries are always stored in groups of 8 entries per cache block of 64B. For an anchor distance of 8, the first page table entry of the cache block is the anchor of the seven other entries in the cache block. Thus 3 bits[1] are sufficient to represent an anchor distance of 8, which will fit into a single page table entry.

[1] 3 bits will represent 0 - 7. If we make the embedded contiguity exclude the anchor entry itself, we can use the embedded value 7 to represent an anchor entry of contiguity 8.

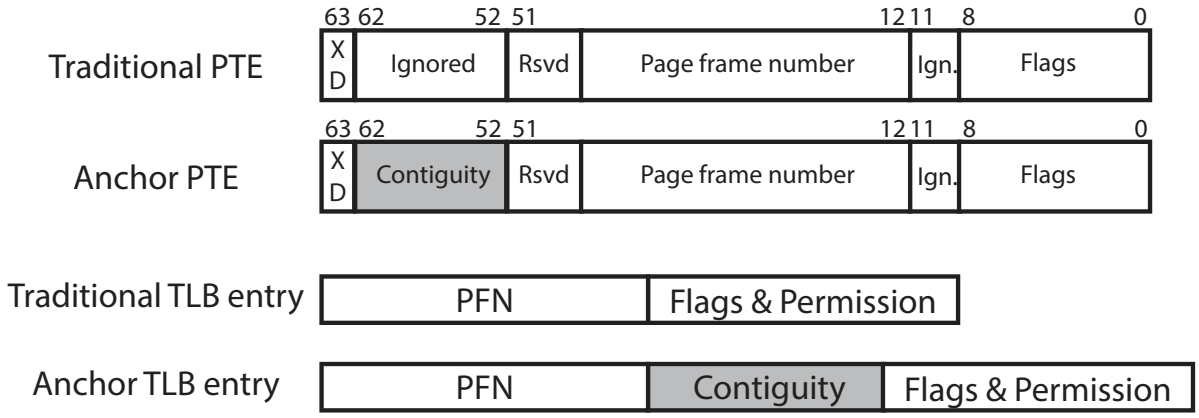


Figure 3.4: Page table and TLB entries for anchor TLB compared to traditional ones

If we need to represent a larger anchor distance, there are still 8 bits that are usable in the page table entry. Furthermore, we can use the unused bits of multiple page table entries of the same cache block, starting from the actual anchor entry. As any anchor for anchor distances larger than 8 will always be aligned to the first page table entry of the cache block, the anchor contiguity reading logic can always read from the first entry of the cache block when reading contiguity bits distributed across page table entries in the same cache block. This allows us sufficient contiguity of up to $2^{(11\times8)}$ if the current physical address maximum of 2^{52}B is maintained. If the physical address maximum is bumped up to 2^{57}B (to match the virtual address space of 5-level paging), $2^{(6\times8)}$ contiguity is still available. This amount of contiguity is more than enough. It is worth mentioning that page table entries are fetched from the main memory in units of cache blocks, and will not result in any additional memory access to read the contiguity bits from different page table entries residing in the same cache block.

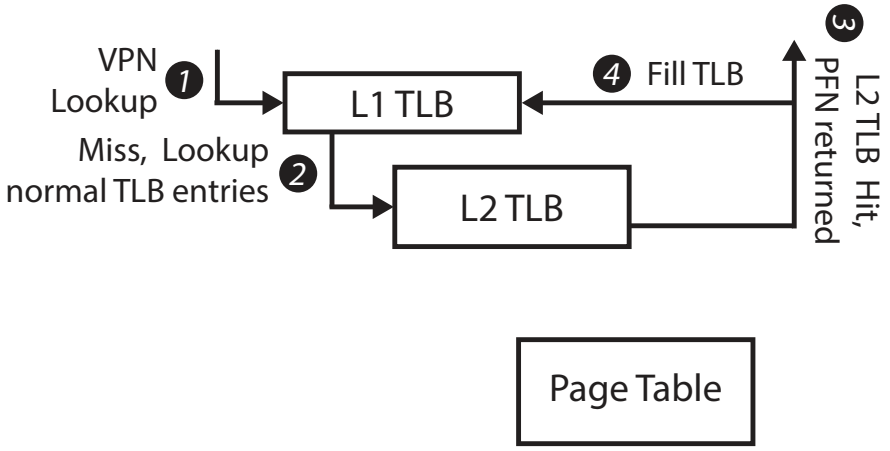
For the evaluation of this paper, we use 16 bits (maximum contiguity of 2^{16}) for the contiguity field to represent the number of contiguous 4KB pages from the anchor entry.

3.3.2 Translation with the Anchored Page Table

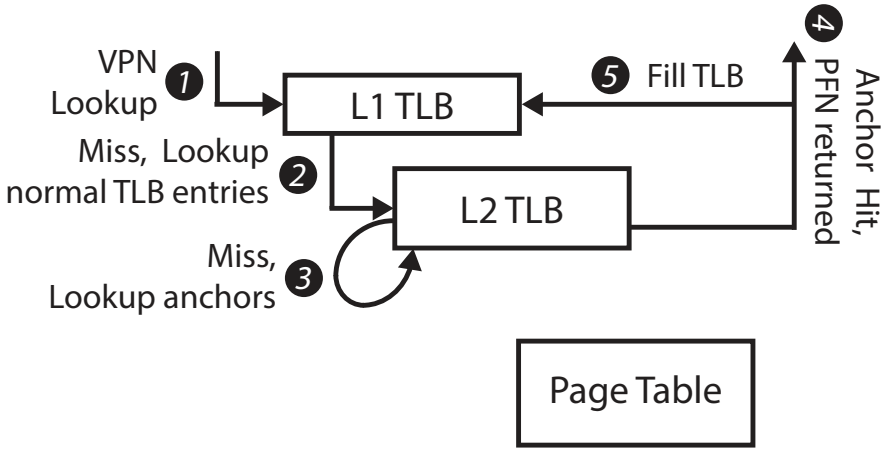
To translate with the anchored page table, only minor changes are required to the MMU. The TLB structure does not need to be modified except for a few additional bits per entry for storing the contiguity field, as both anchor and normal page table entries share the same TLB. As the L1 TLB is tightly integrated with the core, and the performance is sensitive to its access latency, the support for anchor TLB is added to the L2 TLB.

Figure 3.5 illustrates the TLB and page table lookup for L1 TLB misses. On an L1 TLB miss, the L2 TLB is looked up and if it is a hit, as shown in Figure 3.5a, the translation is completed by using the physical page number stored in the TLB entry. For an L2 TLB miss, instead of starting the conventional page table walk, the corresponding anchor entry for the VPN is looked up in the L2 TLB, and if a matching anchor entry is found, the translation is completed, as shown in Figure 3.5b. If an anchor entry misses or the contiguity misses, as shown in Figure 3.5c, a page walk is triggered. Note that the L2 TLB holds *both* regular and anchor TLB entries.

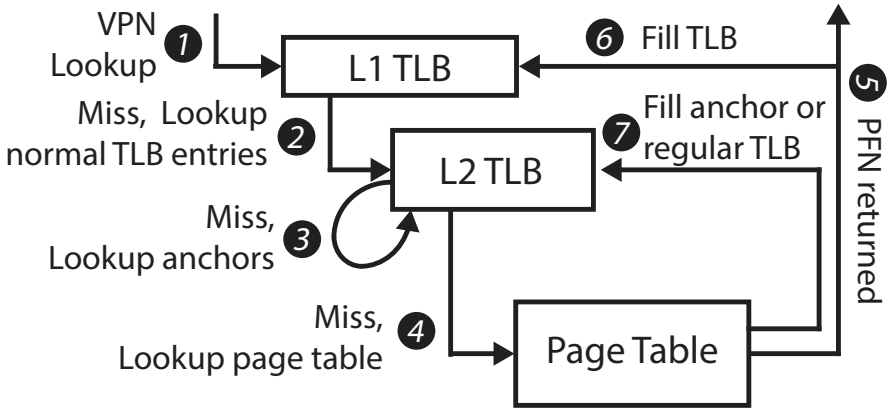
Table 3.2 summarizes the flow of L2 TLB operations for all cases. The first two rows represent regular TLB hit and anchor hit, respectively. The third row represents a case where the anchor lookup succeeds, but the corresponding VPN does not belong to the anchor's contiguous block, resulting in a



(a) L2 TLB hit



(b) L2 TLB miss, anchor hit



(c) L2 TLB miss, anchor miss

Figure 3.5: L2 TLB lookup flows of regular and anchor entry lookups

miss. In this case, the translation information for the VPN only exists in the corresponding page table entry, requiring a page table walk followed by a regular TLB fill.

Next, the final two rows show the case where both regular TLB and anchor lookups miss, causing a page table walk. However, it is unknown whether the VPN is part of an anchor block or not until the page table walk finishes, and thus both regular TLB entry and anchor TLB entry need to be fetched.

Regular Entry	Anchor Entry	Contiguity Match	Operation
Hit	.	.	Translation done
Miss	Hit	Yes	
Miss	Hit	No	Fetch the page table entry and fill in the TLB
Miss	Miss	Yes	Fetch regular entry and anchor entry. Fill only anchor entry in the TLB
Miss	Miss	No	Fetch regular entry and anchor entry. Fill only regular entry in the TLB

Table 3.2: L2 TLB operations

Due to the urgency of execution, the regular TLB entry is fetched first and passed onto the core ❺ and the L1 TLB ❻ as shown in Figure 3.5c. The anchor page table entry is then fetched and the VPN is checked to see if it belongs in the anchor block ❼. This fetch and checking step is no longer in the critical path of core execution. If the VPN belongs in the anchor block (confirmed by the `contiguity match`), the anchor TLB entry is inserted into the L2 TLB. If the VPN does not belong to the anchor block (denied by the `contiguity match`), the regular TLB entry is inserted into the L2 TLB.

Now we will discuss how anchor entries are looked up. For a given VPN to translate, the anchor virtual page number (AVPN) of the incoming VPN needs to be located. The AVPN is located at every alignment boundary of the anchor distance. For example, if the anchor distance is 4, AVPNs are located at frames 0, 4, 8, etc. Thus locating the AVPN of an incoming VPN is calculated by aligning the VPN by the anchor distance. This calculation is executed by clearing out the $log_2(anchor\ distance)$ LSB bits of the VPN which results in the AVPN. Henceforth, we will use the symbol d to denote the $log_2(anchor\ distance)$, which is also denoted in Figure 3.6.

The indexing scheme of TLB requires a modification to store anchor entries effectively. Figure 3.6 represents the relation between the virtual address and VPN, and how VPN is indexed in the TLB. AVPNs have bits with zero values in `[12:d+12)` bits, as the AVPNs are aligned to 2^d pages. To ensure all consecutive AVPNs are mapped to different sets of the TLB, and thus to use all the sets of the TLB for anchor entries, `[d+12:d+12+N)` bits of the virtual address are used as the index bits, where N denotes the $log_2(\#\ of\ sets\ in\ the\ L2\ TLB)$. The bits `[12:d+12)` hold the distance between the VPN and the corresponding AVPN. This distance is compared against the contiguity value of the anchor TLB entry to decide whether the VPN is part of the anchor subblock or not. To finalize the translation on an anchor hit, the physical page number is calculated directly by adding the distance between the VPN and its anchor, $(VPN - AVPN)$, to the physical page number stored in the anchor entry (APPN). The final physical page number is $APPN + (VPN - AVPN)$.

As more pages are allocated contiguously by the anchor entry, translation requests to those pages only need to use the anchor entry, without needing to add their regular entries to the TLB. If the translation is completed successfully with the anchor entry, the actual page table entry for VPN is not fetched from the page table, preventing page walks and preventing unnecessary pollution of the TLB. In an ideal scenario, the TLB will be populated only with anchor entries, each of which can provide translation for multiple pages within its contiguity count. The translation coverage of each anchor entry is limited by the process-wide anchor distance, which is set by the operating system according to the

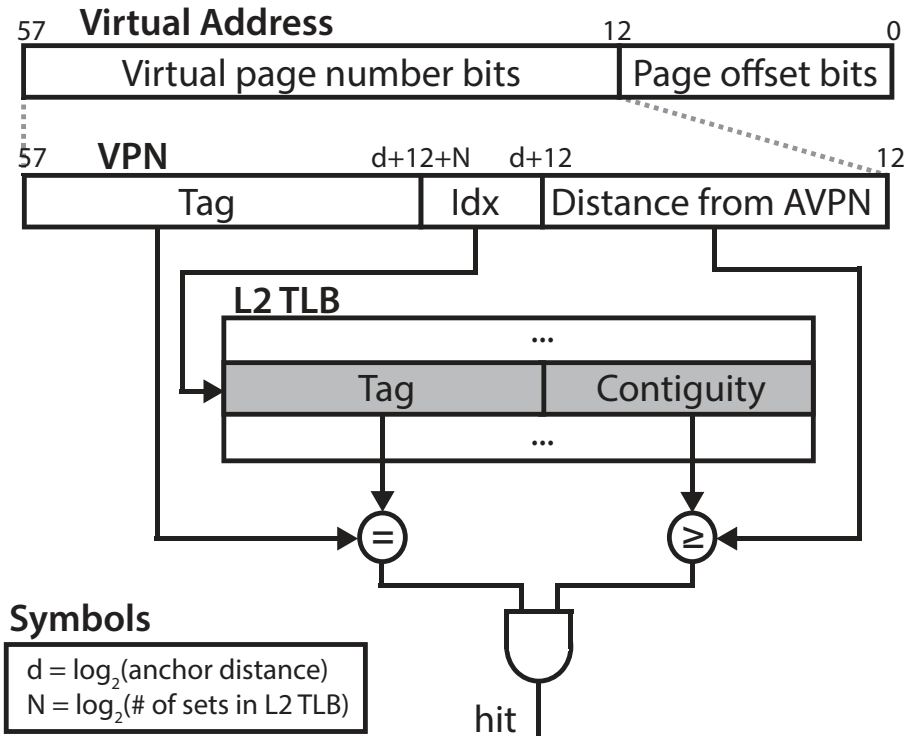


Figure 3.6: Anchor lookups require two comparison units to check the equality of the tag and to check whether the incoming address belongs within the anchor contiguity

contiguity of memory chunks allocated to the process. Therefore, anchor TLB is adaptable and scalable.

3.3.3 OS Implication

To support hybrid coalescing, the extra HW components are limited to the additional register to hold the anchor distance, and adders to produce the physical address from the anchor physical address. To maintain the contiguity information in anchor entries, the operating system requires modest changes.

Updating Memory Mapping: A physical memory frame can be allocated, relocated, or deallocated for a process by the operating system. Whenever the OS updates the memory allocation for a process, including the initial memory allocation for the process creation, in addition to updating the page table entries of changed pages, the related anchor entries may need to be updated, if the continuity bits in the anchor entries are affected by memory allocation changes. After updating the page table entries and anchor entries, a conventional TLB shootdown is invoked, and it invalidates both the page table entries and anchor entries from the TLBs of all cores.

Anchor Distance Change: The second change is to decide the optimal anchor distance for each process, based on its memory allocation status. The OS infrequently checks the fragmentation in allocated memory of a process, and decides the best anchor distance. Section 4 will elaborate the selection algorithm.

When the OS changes the anchor distance, those changes must be propagated through the entire page table. Changing of the anchor distance requires two costs: updating the page table, and synchronizing the TLB. The anchor entries for the new distance are updated to store proper contiguity information. During the updating process, only the page table entries that lie on the anchor distance alignment need to be changed, as the page table walker will only look at those entries when generating an anchor TLB

entry. For example, anchor entries at 0, 4, 8, etc. for the anchor distance of four will be updated. Thus, when changing the distance to a larger distance, smaller number of anchor entries need to be updated, as the distance between anchors are increased. On the other hand, when the distance is changed to a smaller distance, more anchor entries need to be updated. We conducted an experiment to collect the overhead of changing the anchor distance. The cost of sweeping through th entire page table when the process uses 30GB of memory is 452ms, 71.7ms, 1.7ms for changing the anchor distances to 8, 64, and 512, respectively. When a process is created, the default anchor distance can be set to any number chosen by the OS, and it changes to the selected anchor distance once sufficient amount of memory is allocated. If the memory mapping of an application frequently changes during the execution and results in different optimal anchor distances, the OS can make a decision based on the cost and benefit of changing the anchor distance.

The second cost is synchronizing the TLB after updating the page table. In our work, we will update the entire page table, and then we will invalidate the entire TLB. Considering the fact that the `native` Linux kernel for x86 flushes the TLB on context switches, the cost of invalidating the TLB can be relatively minor. Also, in this work we assume the memory map change is checked in periodic epochs of one billion instructions. Even with the periodic checks, the actual anchor distance change is rare, as different epochs still use the same level of memory chunk allocation.

We have found that the anchor distance change to be rarely executed. For specific applications or situations where the memory mapping changes dramatically, the OS can set the limit on how often it can be invoked. In the next section, we will show how the distance is selected.

Permission and Page Sharing: Even if the address mapping is contiguous, pages may have different `r/w/x` permissions. Hybrid coalescing can support any fine-grained permission, by simply treating a page with a different permission as the non-contiguous page. The translation for the page must not be handled by the anchor entry. Although such fine-grained permission differences reduce the effectiveness of any TLB coalescing techniques, the prior work has shown that permissions are commonly homogeneous in much larger granularities[10], and the actual performance impact is expected to be minor in common applications. For page sharing across processes, the contiguity is set in the page table of each process. Therefore, the contiguity in shared regions can also be exploited, although its effectiveness may vary by the anchor distance selected for each process.

3.4 Dynamic Anchor Distance

In this section, we propose a dynamic anchor distance selection algorithm to find the best density of anchor entries in the page table. The per-process anchor distance is determined based on the distribution of contiguous memory chunks allocated to the process. At the beginning of a process execution, only a small amount of memory is allocated, but many processes allocate the majority of memory they use for the rest of execution in the early phase of execution, as shown by Basu et al. [10]. Once the initial memory allocation phase is stabilized, the anchor distance can be selected from the chunk distribution.

However, memory mappings can change even during the execution of a process, as the process itself may dynamically allocate and deallocate memory causing changes in the memory mappings. Furthermore, the operating system may reorganize the memory mappings to optimize the performance. The Linux kernel may try compacting memory as an effort to create more large pages for the process [43, 59]. Operating systems may also promote pages into a super page when sufficient reserved pages have been touched [53]. Large pages may be demoted by the operating system when pages of the large pages are

Algorithm 1 Dynamic distance selection algorithm

```
1: // contiguity_histogram is a list of (contiguity, frequency) pairs
2:
3: // List of available anchor distances in the system
4: Distances ← [2, 4, 8, ... 2^16]
5:
6: // Costs for different anchor distances
7: ∀d ∈ Distances    cost_d ← 0
8:
9: for each d in Distances do
10:    // Calculate distance cost for each contiguity of histogram
11:    for each (cont,freq) in contiguity_histogram do
12:        anchors ← cont/ anch_dist × freq
13:        large_pgs ← remainder / 512 × freq
14:        pages ← remainder × freq
15:
16:        // Weigh down costs of entries with larger coverage
17:        cost_d + = anchors /anch_dist
18:        cost_d + = large_pgs/512
19:        cost_d + = pages
20:    end for
21: end for
22:
23: // Pick anchor distance with min cost
24: min_dist ← min_{∀d∈Distances}(cost_d)
25: setProcAnchorDistance(min_dist) // Set distance of the process
```

unmapped, or even initiated by tools that optimize the system for NUMA-ness [22].

As the memory mapping of any running application may change dynamically, and each execution of the same application can result in different mappings, the operating system needs to provide a means of setting the appropriate anchor distance based on the information available in the OS. In the proposed scheme, the operating system periodically checks the chunk distribution of each process, and recalculates the optimal anchor distance. If the new anchor distance is sufficiently different from the current one, the anchor distance is updated, incurring costly operations of anchor distance change.

3.4.1 Selection Process Overview

The main aim of the selection algorithm is to minimize the number of TLB entries (anchor, large page, and 4KB page entries) required to provide coverage for the active pages mapped to a process. To assess the memory contiguity status, the OS maintains a histogram of contiguity distribution. The contiguity histogram holds how many contiguous memory chunks of varying contiguity are allocated to the process.

Using the contiguity histogram, the capacity cost of storing the page mapping in TLBs is estimated by a heuristic approximation. An anchor distance, which minimizes the capacity cost, will be selected as the best anchor distance. Note that in this method, the frequency of page accesses is not considered, as accurately collecting such information is costly in the current systems. The anchor distance selection

process only uses the static memory mapping information summarized as the contiguity histogram. Our results show that this static estimation can provide a reasonable accuracy for finding the best anchor distance. The algorithm is shown in Algorithm 1 and described in the following paragraphs.

Heuristic Selection: The OS maintains the contiguity histogram to reflect the contiguity of memory pages of a process. Each entry of the contiguity histogram will hold two values: `contiguity` and `frequency`. Contiguity represents the size of chunk, and frequency represents how many chunks of the corresponding contiguity have been allocated. Using the contiguity histogram, the OS estimates how many TLB entries are required to cover the entire memory footprint of the process. For every possible anchor distance value, the OS goes through the contiguity histogram and assembles a *cost* for each anchor distance.

The high level description of the Algorithm 1 is as follows. To calculate the cost per anchor distance, the number of required hypothetical TLB entries is approximated. Once the costs have been calculated for all anchor distances, the distance with the smallest cost is selected.

The counts of required TLB entries are separately counted for 4KB pages, 2MB large pages, and anchor regions. For example, an anchor distance of 16 allows a single anchor entry to cover the size of 64KB (16×4KB). In such a case, a contiguous chunk of 64KB memory will require a single anchor TLB entry, while a 128KB chunk of memory requires two anchor TLB entries. If there are any remaining pages in the chunk after the coverage of anchor entries is deducted from the chunk, the number of required 2MB pages is calculated. The remaining pages after 2MB page deduction must be covered with 4KB pages. Once the required numbers of entries for different types of pages have been calculated, the total cost is obtained from the weighted sum of different types. The weight is the inverse of the coverage of each type.

Distance Stability: Stability of the distance selection algorithm is important, as changing the anchor distance is a costly task, as described in Section 3.3.3. The distance selection algorithm is executed periodically to adapt to any significant memory mapping distribution changes. Although our dynamic selection mechanism periodically checks the best anchor distance, the best anchor distance for a process does rarely change, as once a large amount of memory pages allocated to a process, their chunk distribution does not change significantly for the rest of execution.

In our simulations, we executed the distance selection algorithm every one billion instructions. From the execution based on real machine traces, the distance selection algorithm did not make any changes after making the initial selection decision. Our proposed simple algorithm provides stability of anchor distance selection, preventing any frequent distance changes that may cause significant overheads.

3.4.2 Discussion & Future work

The dynamic distance selection scheme that we have introduced implicitly assumes that the entire address space of a running process has a single clusterable distance. However, an address space has different semantic memory regions: code, data, shared libs., heap and stack. Different regions may have different contiguity. Also, even within the same semantic region, the contiguity distribution may be different, as the OS may have a different memory condition during the execution of the process.

Thus, to further improve the performance of the anchor TLB, *region* may be introduced. A region is part of virtual address space with a separate anchor distance optimized for the region. An address translation for a given region uses the region-specific anchor distance. To support such a multi-region anchor TLB, an additional hardware must hold multiple region definitions consisting of the starting VPN, ending VPN, and anchor distance for each region. The additional HW component is similar to

Schemes	TLB Configuration
Common L1	4KB: 64 entry, 4-way 2MB: 32 entry, 4-way
Baseline/THP	4KB/2MB(shared) 1024 entry, 8-way
Cluster	Regular TLB: 768 entry, 6 way Cluster-8: 320 entry, 5 way
RMM	Baseline L2 TLB + RMM: 32 entry, fully associative
Anchor	4KB/2MB/Anchor (shared): 1024 entry, 8-way
Latencies	7 cycle L2 hit latency [30] 8 cycle clust./RMM/anch. hit lat. 50 cycle page table walk lat. [38]

Table 3.3: TLB configuration used for evaluation

the range TLB structure in RMM. The *region table* will be looked up in parallel with the 4KB/2MB TLB lookup. Since all the regions must be searched in parallel for fast accesses, the number of regions is limited. If the TLB lookup misses, the anchor distance from the matching region is used to lookup the anchor entry in the L2 TLB. The dynamic distance selection can be extended to partition the memory into different regions if there are contiguity variances. We believe that such approach would further improve the anchor efficiency, and we leave this as future work.

3.5 Evaluation

3.5.1 Methodology

We simulated our work on a trace-based simulator that models the cache and TLB structures. The configuration of the HW components are shown in the Table 3.3. We executed benchmarks from the `SPECCPU2006`, `biobench` suites, and additionally `graph500` and `gups`. The working set sizes of `graph500` and `gups` are set to 8GB. To generate the trace of each application, we used the Pin binary instrumentation tool [50] and generated a memory access trace of 12 billion instructions. At the same time, at every billionth instruction boundary, we periodically captured the virtual to physical memory address mapping on the real machine, using the `pagemap` [47] interface provided by Linux.

Scenario	Contiguity
`low contiguity`	1 - 16 pages (4KB - 64KB)
`medium contiguity`	1 - 512 pages (4KB - 2MB)
`high contiguity`	512 - 65,536 pages (2MB - 256MB)
`max contiguity`	maximum

Table 3.4: Synthetic mapping scenarios

For the evaluation, we show the performance of our system using a total of 6 mapping scenarios: two mappings were captured from actual system executions, and four mappings were synthetically generated.
Real Mappings: we generated a trace on a vanilla Linux 3.18.29 machine, which uses `demand paging`. With demand paging, memory pages are not allocated until actual accesses (via page fault) occur. This approach is the default behavior currently used. It minimizes unused pages, but the contiguity of the page mappings can be potentially reduced. We also generated a trace when the system uses `eager paging`, which allocates memory pages immediately upon a memory allocation request from the process. We modified the Linux kernel to map pages for the eager allocation. However, our eager paging implementation does not directly allocate large chunks of contiguous space, but rather requests pages through the buddy allocator system sequentially. The use of eager paging increases the mapping contiguity, compared to the contiguity of demand paging. Linux transparent huge page support was enabled when generating traces of both real mappings.
Synthetic Mappings: we generated four synthetic mappings: `low contiguity`, `medium contiguity`, `high contiguity`, and `maximum contiguity`. Table 3.4 shows the contiguity distributions for the mappings. For each mapping, chunk sizes are selected from the given range with a uniform random distribution. The `low contiguity` and `medium contiguity` represent the scenarios which cannot provide large chunks. `maximum contiguity` is an extreme contiguous mapping where every contiguous virtual address region is mapped to the same amount of contiguous physical region. This is an ideal mapping for segment-based translations such as RMM. The synthetic mappings are used to exercise the proposed scheme and prior ones with various allocation scenarios, revealing the advantages and disadvantages of the approaches under different allocation situations.
Comparison: We compare our work to `THP` (Transparent Huge Pages [48]), an implementation of large page in Linux, cluster TLB (`cluster`) [58], and `RMM` [37]. Cluster TLB requires a partitioned TLB with regular and cluster entries. However, our baseline TLB is larger than that used by Pham et al. [58]. To scale the TLB, we increased both of the numbers of cluster and regular entries proportionally, while keeping the set-way settings reasonable. The original cluster TLB does not use the 2MB large page, and for fair comparison, we also evaluate `cluster-2MB` which can use the large page in the regular TLB entries in addition to clustering at 4KB. `RMM` requires an additional fully associative range TLB. We employed a 32 entry range TLB as used by Karakostas et al. [37], in addition to the 1024 entry baseline TLB. The 1024 baseline TLB with `RMM` supports both 4KB and 2MB pages.

We show the performance of our method in two schemes: `dynamic` and `static ideal`. In `dynamic`, the proposed dynamic distance selection algorithm is used to pick the anchor distance. In `static ideal`, we show the results with one optimal distance which performs best for each application and mapping,

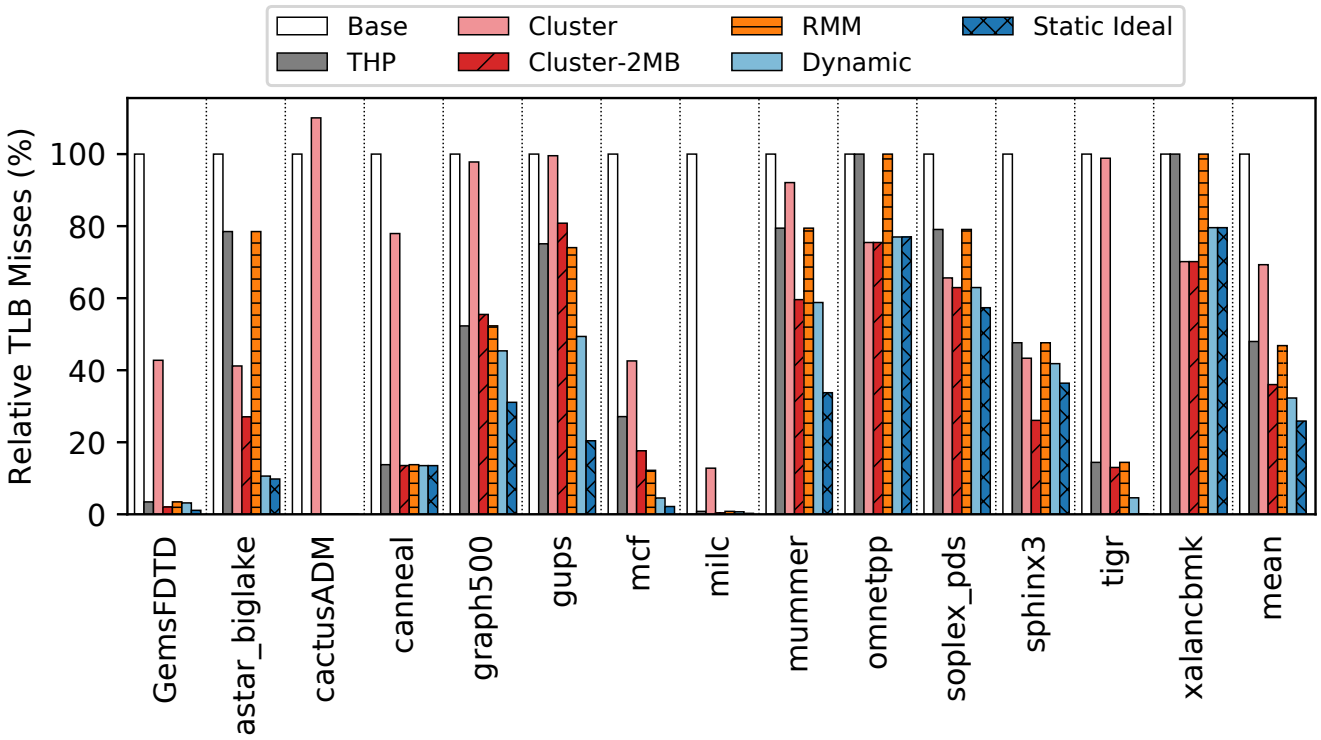


Figure 3.7: Relative TLB misses for `demand paging` mapping

by exhaustive evaluation of all possible distances. The `static ideal` scheme is used to illustrate the near maximum TLB miss reductions achievable by the anchor selection algorithm.

TLB Parameters: Table 3.3 describes the TLB parameters for the prior schemes and hybrid coalescing. The L1 TLB configuration is the same for all schemes. The L2 TLB capacity is set to 1024 entries. However, `cluster` partitions it to the regular and cluster entries. In addition to the L2 TLB, `RMM` has the 32-entry fully associative range TLB. The latency parameters are derived from those used by Karakostas et al. [38].

3.5.2 Results

We present the evaluation results of all benchmarks for `demand paging` in the real mapping and `medium contiguity` in the synthetic mapping. For the rest of mapping scenarios, we present the mean TLB reductions due to the space constraint.

TLB Misses for Demand Paging and Medium Contiguity

Figure 3.7 shows the relative TLB misses with `demand paging`, normalized to those with the baseline configuration. With the transparent huge page support turned on, the mapping of demand paging contains many 2MB contiguous chunks. Therefore, all the techniques other than `baseline` and `cluster` which do not use the large page, benefit from the large page allocation. Across all the applications except for `omnetpp` and `xalancbmk`, THP alone can reduce the TLB misses effectively by average 60% for all the other applications. Although the original `cluster` reduces TLB misses effectively for `mcf` and `milc` significantly, their reductions are limited for some applications (`canneal`, `gups`, `mummer`, and `tigr`) without support of the large page. However, `cluster-2MB` can benefit both from HW coalescing and

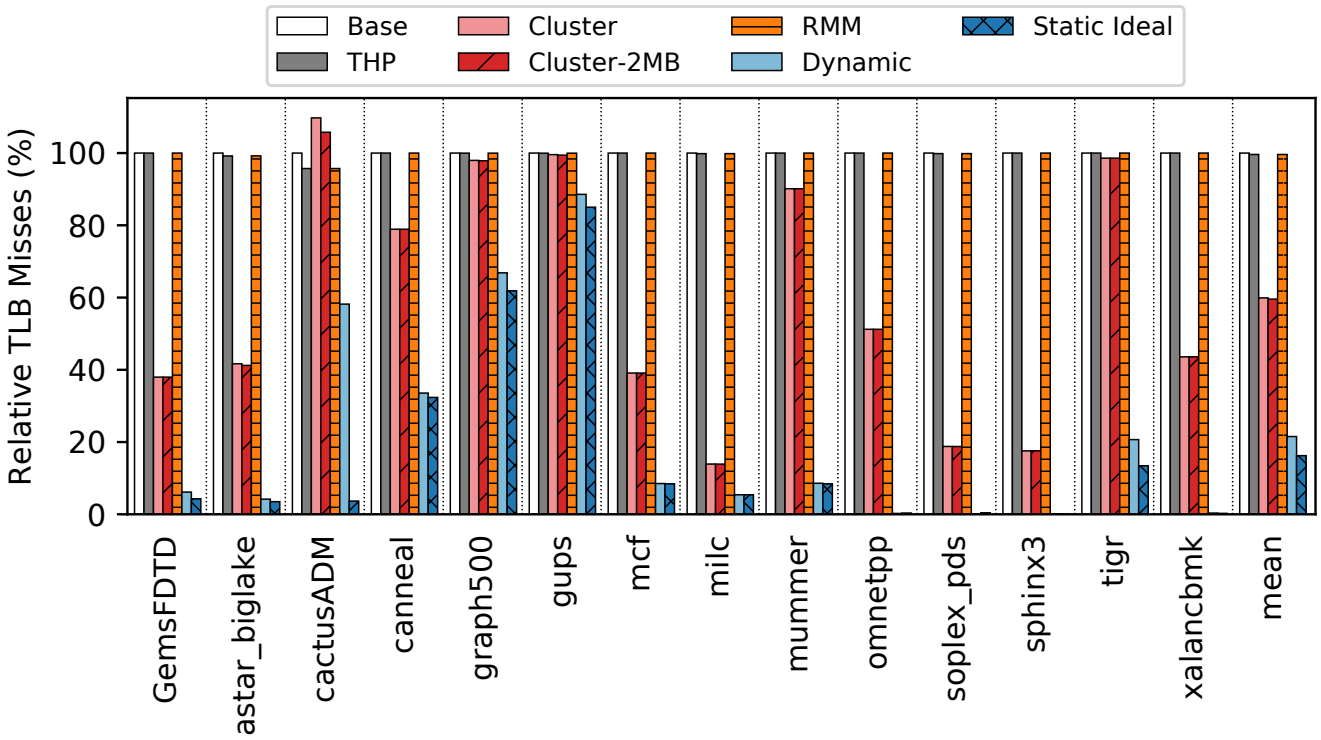


Figure 3.8: Relative TLB misses for `medium contiguity` mapping

large page, achieving 64% reduction on average. `RMM` is also effective for the majority of applications, and their reductions are on average 53.2% which are similar to `THP`. However, both `THP` and `RMM` are not very effective for `omnetpp` and `xalancbmk`, as those applications do not exhibit large chunk contiguity. For the two applications, HW coalescing of `cluster` and `cluster-2MB` effectively reduces TLB misses among the prior schemes.

Unlike the prior schemes, hybrid coalescing can consistently reduce TLB misses, by exploiting both large page contiguity and fine-grained coalescing. With `demand paging`, the proposed dynamic hybrid coalescing can provide the reduction of 67.3% on average, which is better than the best prior scheme (`cluster`) for the mapping.

Figure 3.8 plots the evaluation on the `medium contiguity` mapping. Unlike `demand paging`, this mapping has contiguity mostly less than 2MB page, and thus `THP` is ineffective. `RMM` also shows similar results to `THP`, due to the lack of high contiguity. For this fine-grained contiguity existing in the mapping, `cluster` and `cluster-2MB` reduce misses effectively for many applications, but they are not very effective for `graph500`, `gups`, and `tigr`. For `cactusADM`, the misses get worse due to the statically partitioned TLB between regular and clustered entries. In `cactusADM`, the clustered TLB entries are underutilized, while the regular entries are full.

The proposed hybrid coalescing can effectively exploit the available contiguity, excelling the other schemes. It can effectively utilize the medium contiguity of less than 2MB, which cannot be exploited by `THP` and `RMM`. At the same time, hybrid coalescing has higher coverage than `cluster` and `cluster-2MB`. Even for the worst case `gups` application, it can reduce TLB misses by 11.4%. The difference between `dynamic` and `static-ideal` is relatively small. However, for `cactusADM`, the dynamic algorithm does not find the best optimal distance, although `dynamic` still reduces TLB misses significantly. It is due to

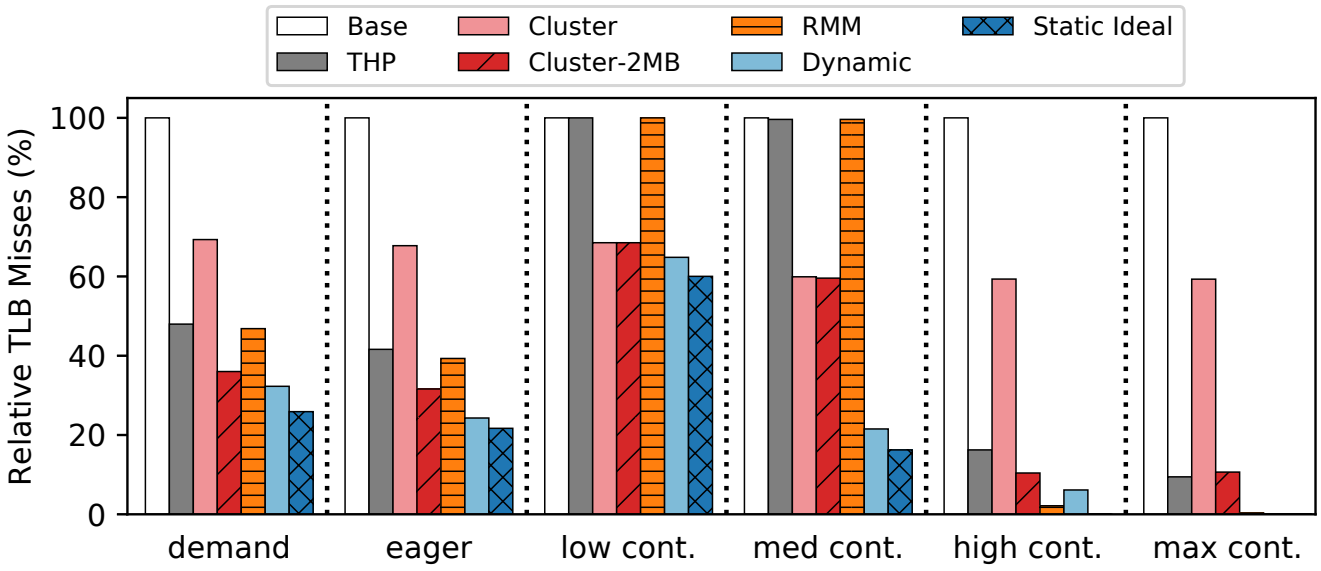


Figure 3.9: Average TLB misses of each translation scheme for all mapping scenarios

the static nature of the dynamic algorithm which finds the distance based on the allocation snapshot, without knowing access frequency.

Table 3.5 shows the access breakdowns at the L2 TLB for the `demand paging` and `medium contiguity` mappings. As discussed with Figure 3.7, the 2MB large page is very effective for `demand paging`, and thus many L2 TLB accesses hit on the regular L2 TLB entries containing large pages. For applications with relative low regular hit rates, such as `canneal`, `xalancbmk`, and `gups`, anchor entries additionally absorb 16-55% of TLB accesses. In `medium contiguity`, the regular entry hit rates are much lower than those in `demand paging`, due to the lack of large page accesses. In such cases, anchor entries can be very effective in reducing TLB misses.

TLB Miss Summary of All Mapping Scenarios

Figure 3.9 summarizes the TLB miss reductions for all six mappings. `Demand paging` and `eager paging` provide both the 2MB large page and fine-grained coalescing chances depending on applications, with a higher overall benefit from the 2MB large page. Effectively utilizing the two opportunities, `cluster-2MB` can provide the best reduction among the prior schemes, reducing TLB misses by 64% and 68.4% on average for demand and eager mappings, respectively. The proposed hybrid coalescing further reduces TLB misses benefiting from a higher coalescing capability than `cluster-2MB`, achieving 67.7% and 75.7% reduction.

The `low` and `medium continuity` mappings provide few 2MB or larger chunks, and thus only the prior HW clustering and the proposed hybrid coalescing can reduce TLB misses. However, with better coalescing scalability, the hybrid coalescing can reduce TLB misses by 35.2% and 78.5%, compared to 31.5% and 40.4% of `cluster-2MB` in low and medium mappings. For these mappings, `THP` and `RMM` are nearly ineffective. The `high` and `maximum contiguity` mappings provide large contiguous chunks which help both `THP` and `RMM`. `RMM`, with better coverage scalability, almost eliminates TLB misses. Even in these cases, the proposed hybrid coalescing almost matches the miss reductions of `RMM`.

From the results, we conclude that our scheme outperforms or performs similar to the best prior

	demand			medium-contiguity		
	R.hit	A.hit	L2 miss	R.hit	A.hit	L2 miss
astar	43 %	49 %	6 %	52 %	46 %	2 %
cactus	49 %	51 %	0 %	11 %	44 %	45 %
canne.	33 %	55 %	12 %	25 %	59 %	16 %
GemsF.	91 %	8 %	1 %	13 %	85 %	2 %
mcf	91 %	8 %	1 %	66 %	32 %	2 %
milc	74 %	25 %	1 %	3 %	92 %	5 %
omnet.	48 %	29 %	23 %	62 %	38 %	0 %
soplex	75 %	12 %	13 %	57 %	43 %	0 %
sphinx	87 %	3 %	10 %	53 %	47 %	0 %
xalan.	18 %	16 %	66 %	66 %	34 %	0 %
mummer	39 %	5 %	56 %	70 %	22 %	8 %
tigr	61 %	34 %	5 %	61 %	22 %	17 %
gups	27 %	20 %	53 %	11 %	1 %	88 %
graph.	49 %	5 %	46 %	29 %	5 %	66 %

Table 3.5: L2 TLB hit/miss statistics. The regular L2 TLB hit rate (R.hit), anchor TLB hit rate (A.hit), and L2 TLB miss rate are shown.

	demand	eager	low	medium	high	max
astar_biglake	16	256	4	16	128	256
cactusADM	4K	8K	4	32	256	512
canneal	1K	512	4	8	256	1K
GemsFDTD	8K	8K	4	32	256	1K
mcf	64K	64K	4	32	512	64K
milc	16K	8K	4	32	256	256
omnetpp	4	4	4	16	128	256
soplex_pds	2	2	4	16	64	64
sphinx3	4	4	4	32	32	32
xalancbmk	4	4	4	32	128	128
mummer	2K	32K	4	32	128	256
tigr	2K	512	4	32	256	512
gups	32K	32K	4	32	1K	64K
graph500	64K	16K	4	32	1K	64K

Table 3.6: Anchor distances in pages, selected by the dynamic distance selection algorithm.

scheme for each mapping scenario, achieving the best average performance across diverse scenarios. It can adjust the coalescing capability dynamically to match various allocation contiguity distributions. Such dynamic adjustment allows it to extract whatever available contiguity as efficiently as possible.

Anchor Distance Selection

Table 3.6 shows the anchor distances selected by the dynamic selection algorithm for different mapping scenarios. The values shown are the number of consecutive pages. For `demand paging` and `eager paging`, the contiguity distributions are highly skewed for many applications with a small number of very large contiguous chunks in memory allocation. As these chunks tend to dominate the memory footprint, the selection algorithm chose large distances. Note that 64K is the largest anchor distance used for the evaluation. However, for a few applications, such as `omnetpp` and `xalancbmk`, a small distance of 4 pages are selected. As shown in Figure 3.7, the two applications have a fine-grained contiguity which can only be coalesced by `cluster`, `cluster-2MB`, and the proposed scheme.

However, for synthetic mappings, the chunk distributions are uniform. For the chunk distribution of each configuration, the heuristic algorithm finds a reasonably good distance for each range, with four

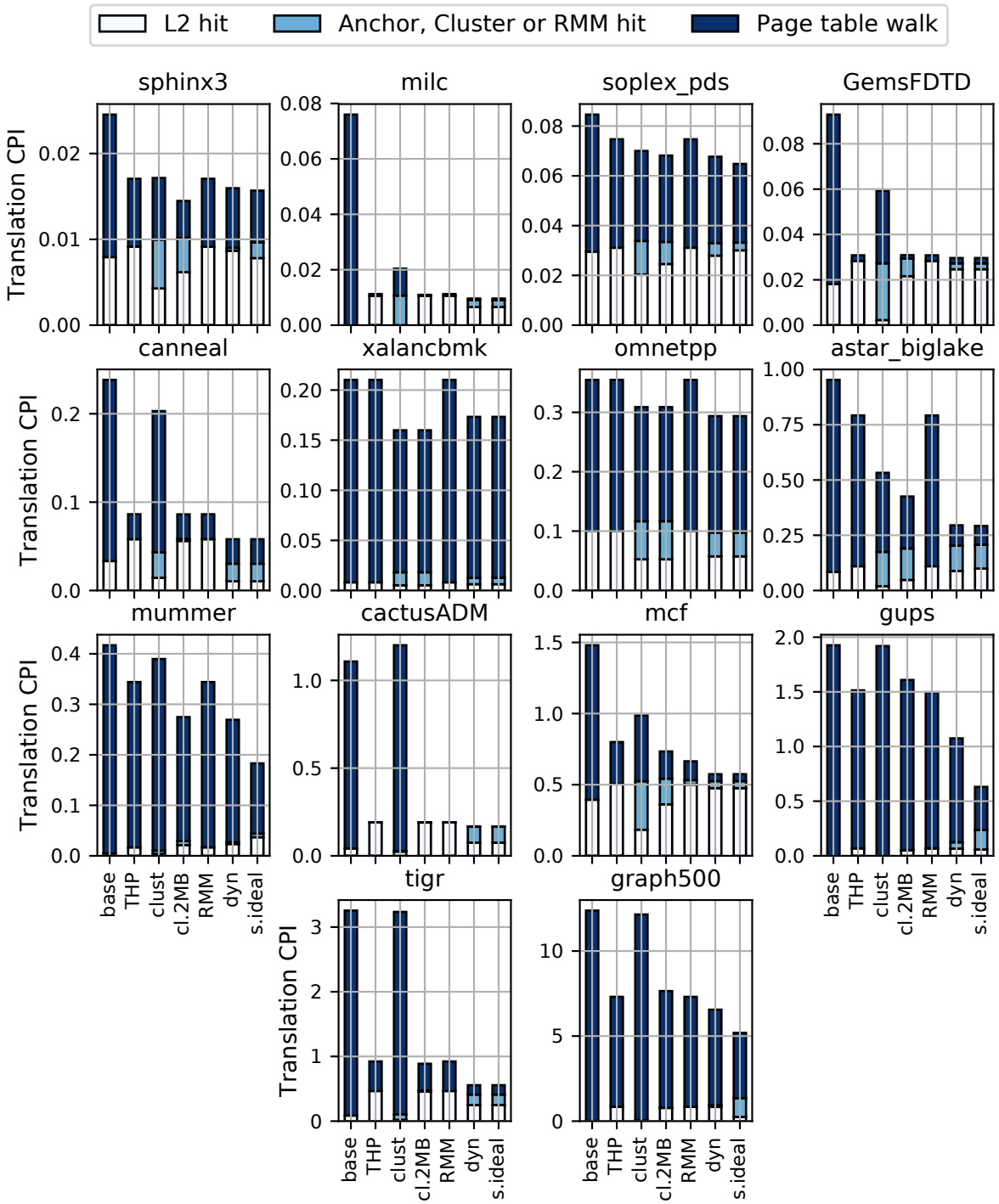


Figure 3.10: CPI breakdown of translation overhead for `demand paging`

in `low contiguity` to 32-1K in `high contiguity`.

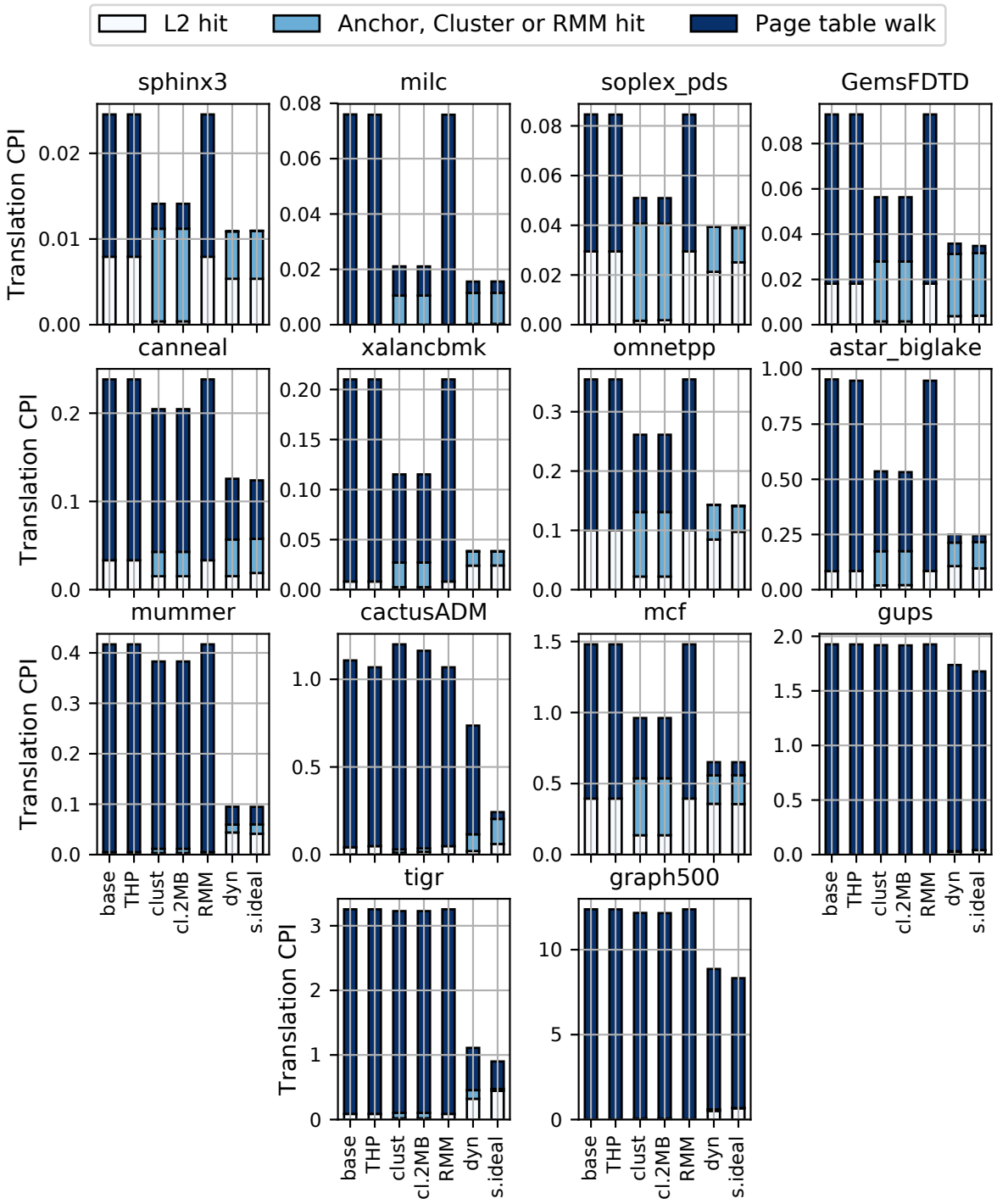


Figure 3.11: CPI breakdown of translation overhead for `medium contiguity`

Translation CPI

Figures 3.10 and 3.11 show the breakdown of the cycles spent per instruction in the address translation. The L1 TLB will be accessed in parallel with the cache access and so the L1 TLB access

latency is hidden [38]. We estimate the CPIs based on the latencies in Table 3.3. The L2 hit portion denotes CPIs spent for regular L2 TLB hits. The next CPI portions are specific to each technique: cycles spent for `anchor hits`, `cluster TLB hits`, `range TLB hits` for hybrid coalescing, cluster, and RMM, respectively. Firstly, the CPI results are consistent with the TLB miss analysis in Figures 3.7 and 3.8, although the actual CPI changes are affected by TLB misses per instruction. The proposed hybrid coalescing is better than or almost match the best prior scheme for each mapping scenario. For applications with high TLB miss rates in the baseline configuration, the performance improvements of the proposed scheme are significant, with the CPI reduction of 0.85, 2.7, and 5.82 for `gups`, `tigr` and `graph500`, respectively, for the `demand paging` mappings. In the case of `graph500` executing on the `medium contiguity` mapping, up to 3.51 CPI reduction is expected.

The translation CPI can be used to estimate the performance benefit extractable. To estimate the achievable performance benefit in terms of IPC, we first measured the IPC, ratio of time spent walking the page table, and the TLB miss stats from a Intel Haswell machine, as the TLBs modeled in this work are based on the Haswell machine. We took hardware counter measurements every second, and looked for the TLB miss rate that was the closest to the miss rate at the point of execution that was simulated. Finally, we used the translation CPI, the instructions and cycle counts, and the number of cycles spent serving TLB misses, and generated the estimated IPC improvements. Figure 3.12 presents the estimated IPC gains achievable. However, it is worth noting that the estimate is only an estimate, and the results are prone to error, as the memory mapping situation which the simulations are based on may be very different from the real execution.

3.6 Virtualization

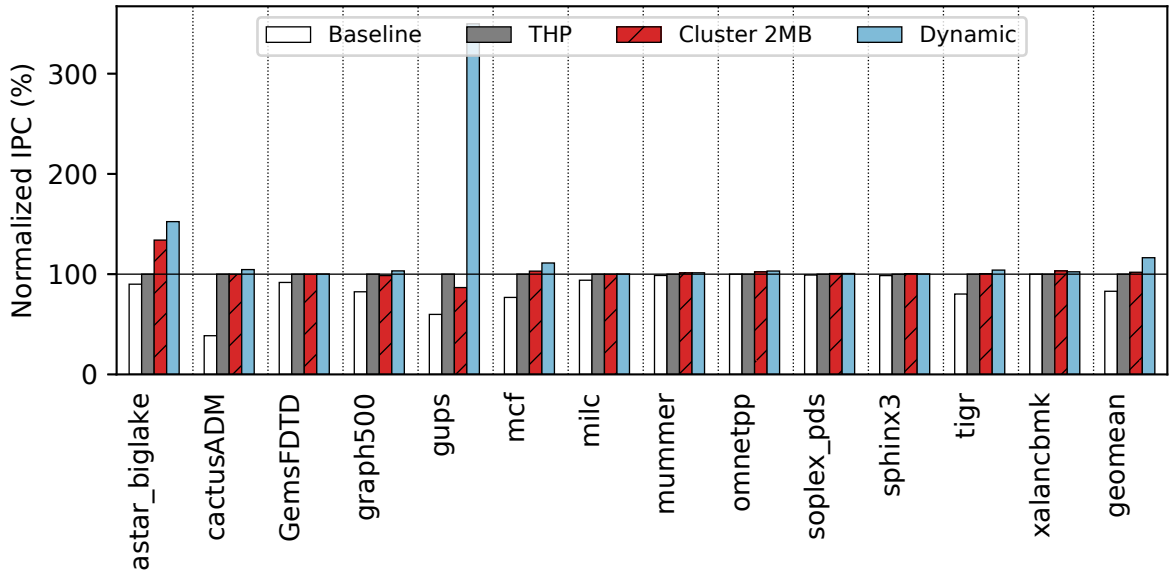
Virtualization introduces even more challenges for the virtual memory. Hardware assisted virtualization implemented by both Intel and AMD result in a two dimensional page table. What used to be four memory accesses to translate an address now becomes twenty four memory accesses.

The exacerbated address translation latency puts more emphasis on an efficient translation scheme. Two utilize hybrid TLB coalescing to its fullest extent two requirements need to be met. Firstly, allocating memory in a contiguous manner both in the guest operating system (guest OS) and the host operating system (host). Secondly, during the page table walk, incurred by TLB misses, the contiguity allocation information of both the guest OS and the host needs to be utilized. In the remainder of this section we discuss how to support virtualization with HTC.

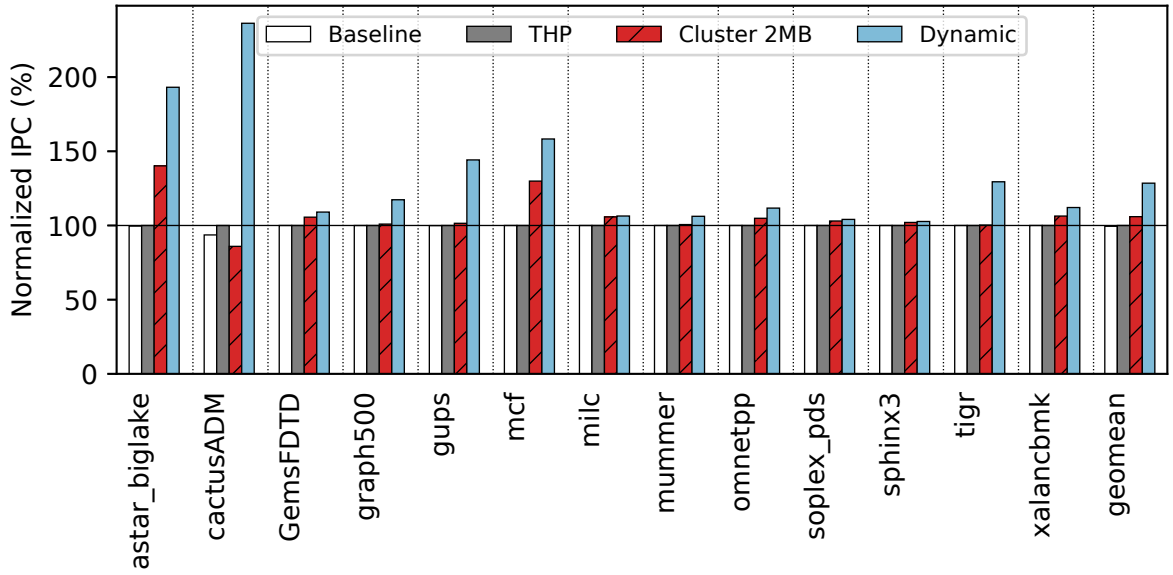
3.6.1 Memory Allocation

With the advent of hardware assisted virtualization technology, the guest and host have been able to independently manage memory. In the current implementation of linux and the kernel virtual machine (KVM) with transparent huge page support (THP) the guest and the host are not coordinated in any way when allocating memory. This in turn results in the gap between the guest and host allocations.

To explore the discrepancy between the memory allocation mappings of the guest and host OS, we used the `pagemap` utility to get the guest virtual address (gVA) to the guest physical address (gPA) mapping. To log the host mapping of gPA to host physical address (hPA) we added an interface at the host to allow the guest to query the gPA to hPA mapping by passing a gPA. The host services the request, where the guest calls a hypercall, and returns the associated hPA. Using this interface we



(a) Estimated IPC improvement for `demand paging`

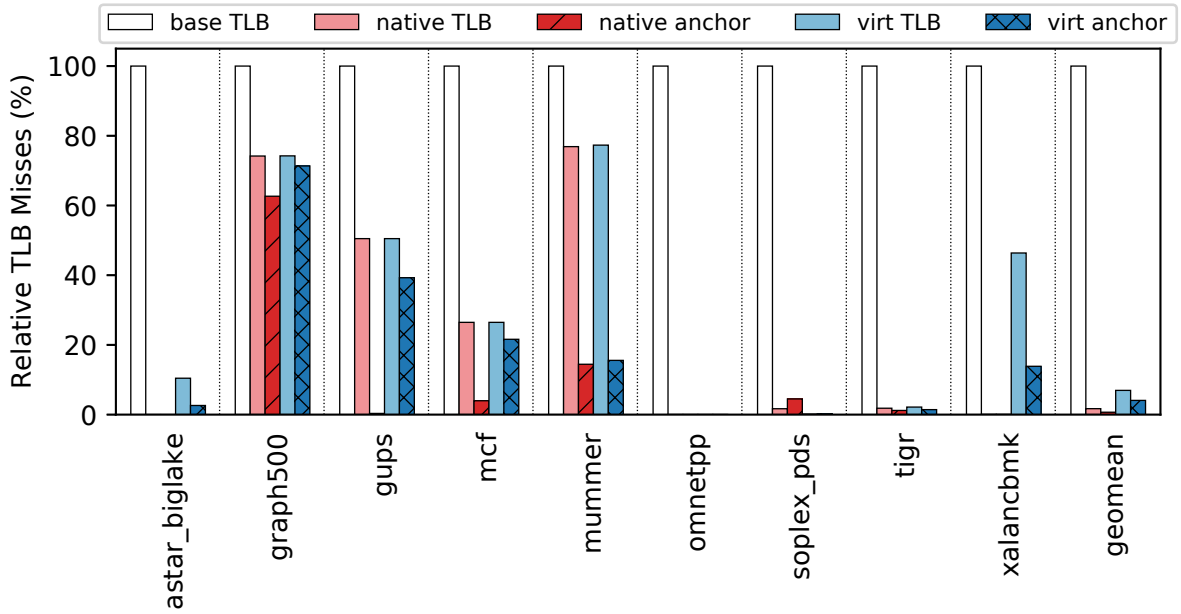


(b) Estimated IPC improvement for `medium contiguity`

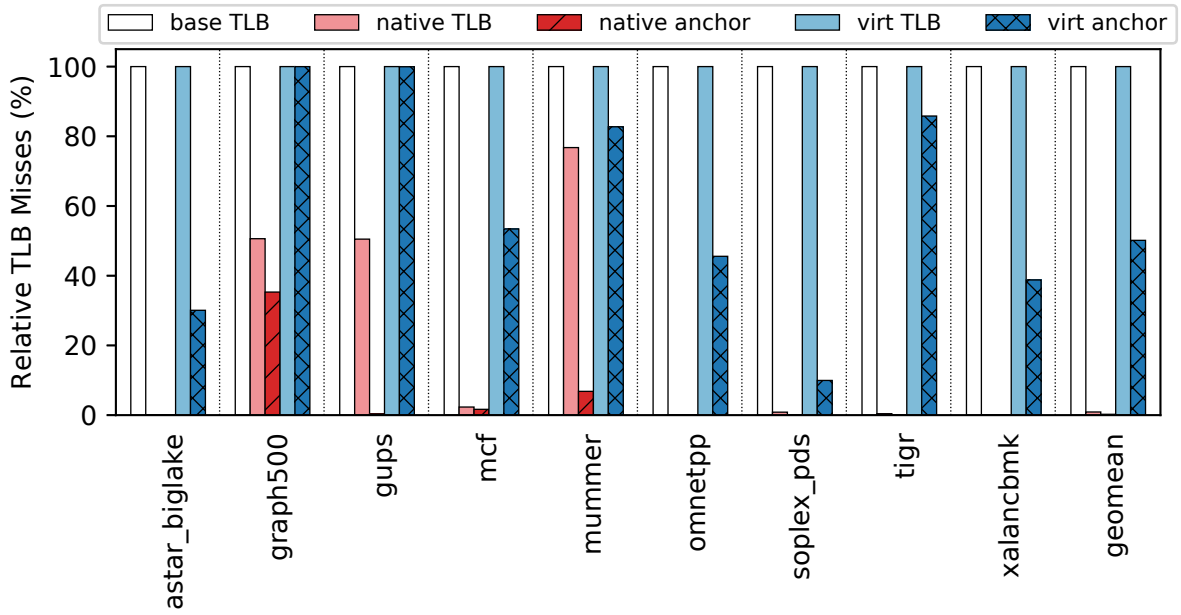
Figure 3.12: The normalied IPC gains estimated based on translation CPI. The IPCs are normalized to `THP` IPC

studied the memory mapping of the current Linux and KVM. The contiguity of memory allocation in the guest and host mapping are shown in Figure 3.14 and Figure 3.15. The red curve illustrates the guest mapping, and the orange curve illustrates the host mapping. We explored both cases with transparent hugepage enabled and disabled in the host (hypervisor) to additionally show a system having difficulty providing large pages. The vertical rises in the figure represent the allocation contiguity presented by the x axis. For example if there is a 20% rise at 2^9, this means that out of the entire allocated memory of the system, 20% are allocated in blocks of 512 contiguous pages.

We found that there were several workloads that make use of large pages very well, and both guest

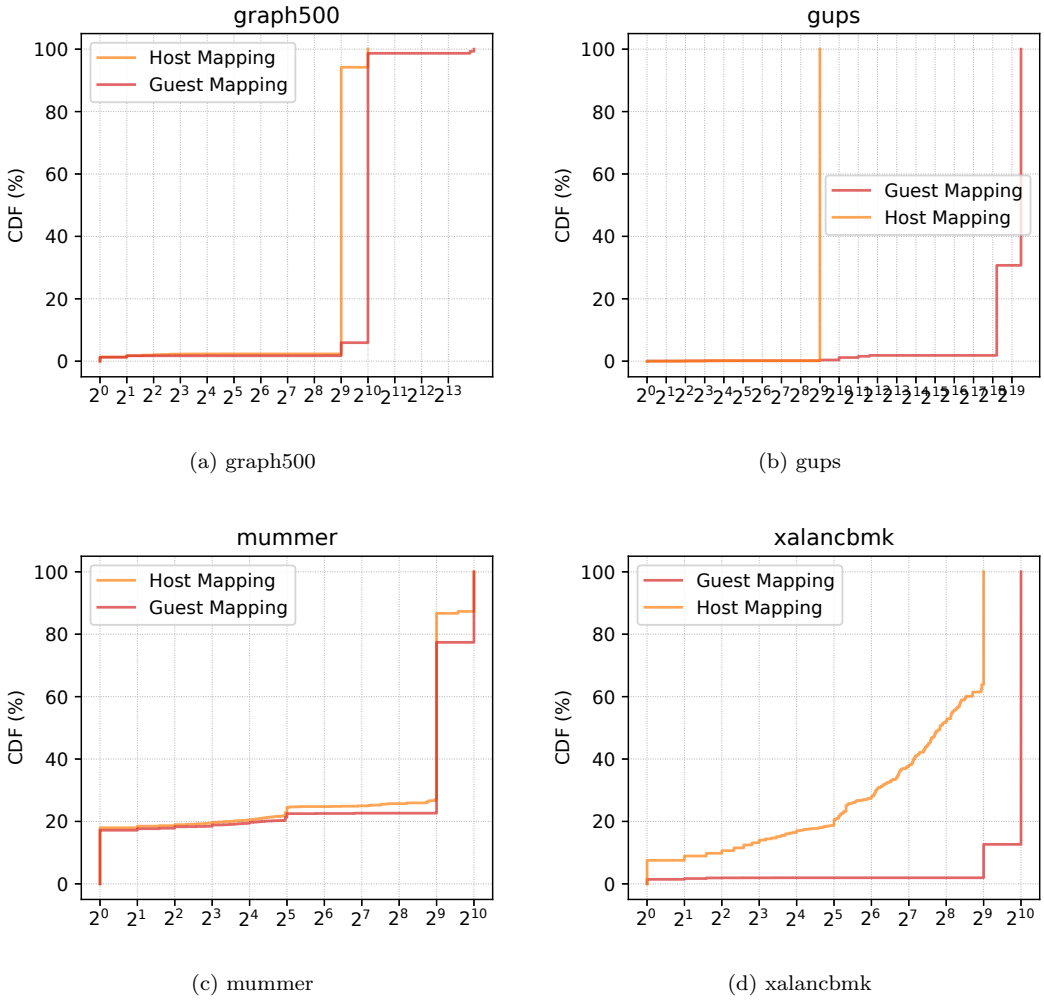


(a) Host with largepage support



(b) Host without largepage support

Figure 3.13: Normalized TLB misses of workloads on native systems and virtualized systems, in differeing memory allocation scenarios



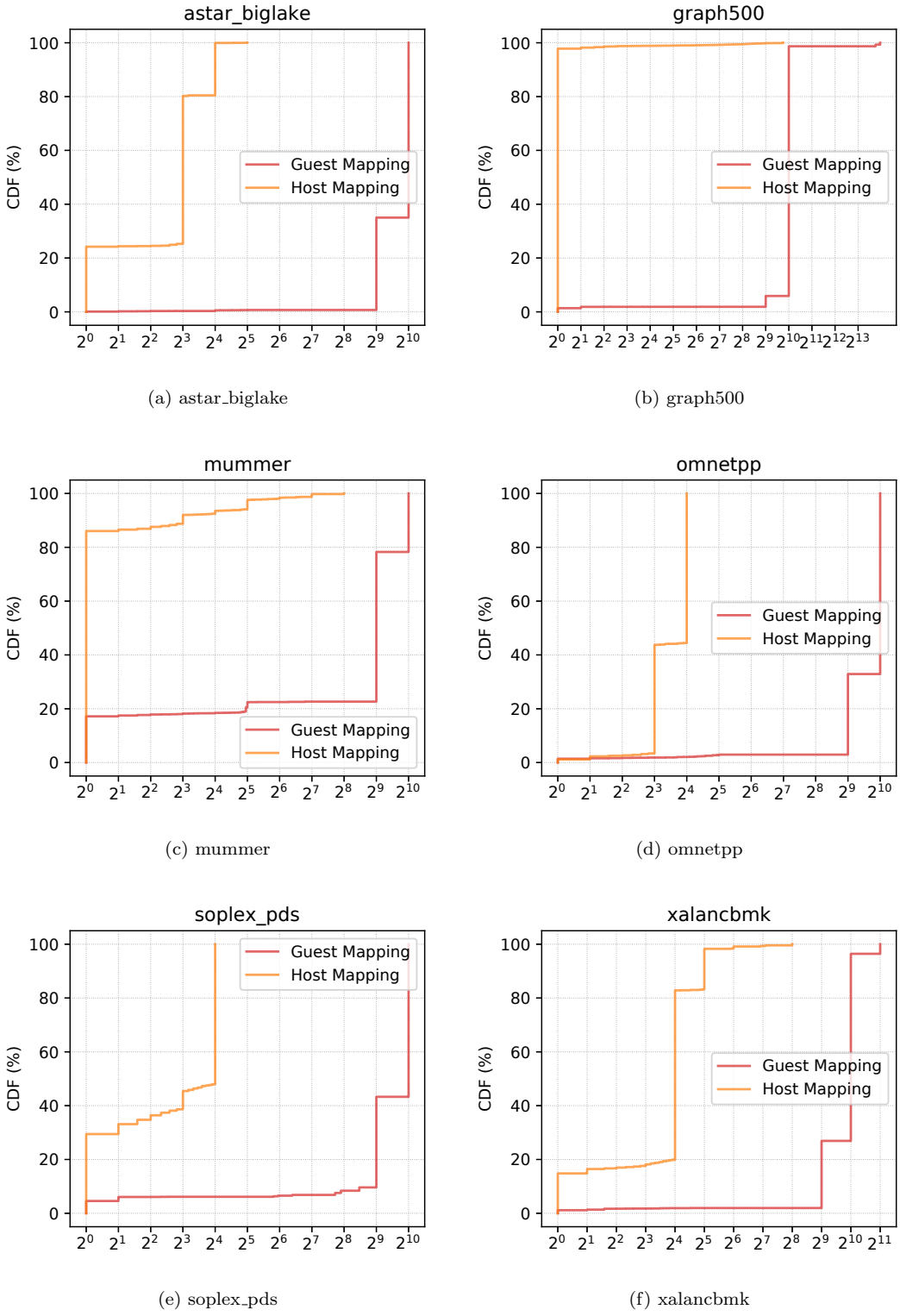
(a) graph500

(b) gups

(c) mummer

(d) xalancbmk

Figure 3.14: The contiguity CDF of the total allocated memory of the benchmark. Each line plot the contiguity of allocation of mappings allocated at different layers of the system. These mappings were acquired with THP enabled on both guest and host



(a) astar_biglake

(b) graph500

(c) mummer

(d) omnetpp

(e) soplex_pds

(f) xalancbmk

Figure 3.15: Contiguity CDF plots of execution where the host THP was turned off to emulate a system that does not allocate large pages

and host show very similar contiguity CDF curves. Figure 3.14a and Figure 3.14c represent such cases. These workloads show near native TLB hitrate with HTC, with the only penalty of virtualization evident in the page walks. In some other cases, there was a wide gap between the allocations at the guest and the allocations at the host, as represented by Figure 3.14b and Figure 3.14d.. With such allocations, the performance of the anchor TLB in native (assuming the guest mappings) and virtualized system (combination of guest and host mapping) is shown in Figure 3.13a. `Graph500` and `mummer` show very similar performance for both native and virtualized cases. However, `gups` and `xalan` show different performance characteristics. `Gups` provides contiguous mappings larger than 2MB in guest mappings, allowing the native anchor TLB to outperform native TLB. (`Gups` incurs high TLB misses even when using large pages in conventional TLBs because of the large working set). However, as the host mapping does not allocate contiguities larger then 2MB, the virtualized anchor TLB fails to perform as well as regular virtualized TLBs.

To show the possible memory mapping scenario when the host is severely fragmented, for example due to significant memory pressure, we experimented with the host transparent huge pages (THP) turned off. The mapping contiguity CDFs are plotted in Figure 3.15 and the TLB MPKI results are plotted in Figure 3.13b. For most of the workloads, the gPA to hPA show a very fragmented allocation, where most of the allocations have single page contiguity(graph500 and mummer, and many more omitted due to space restriction). However, there are a few workloads that show some intermediate contiguities of eight or even 16 pages (`astar`, `omnetpp`, `soplex` and `xalancbmk`). In such cases anchor TLB can make use of these intermediate contiguities to outperform the native TLB. The MPKI results show that for the aforementioned workloads with intermediate contiguities do show significant miss reductions. On the other hand, the workloads that are very highly fragmented and do not have contiguities over one page do not show any performance improvement. However, even with such improvements by anchor TLB in virtualized systems, the TLB miss of the virtualized anchor TLB is still much larger than TLBs of native systems. This is due to the independent memory mapping at the guest and the host OS.

To illustrate the mapping between the guest and the host we present an example. Figure 3.16 illustrates the two hierarchy of mappings, the guest virtual to guest physical is the `guest mapping` and the guest physical to the machine is the `host mapping`. The first group of allocation (blue rectangle on the left) shows a case where the host mapping is contiguous, but the guest allocation is not, restricting in the effective contiguity of the anchor TLB to two. Only the first two guest virtual pages will be directly translatable in the anchor TLB. The next two contiguous block will need to translate to gPA, then again from gPA to hPA. In the case of the second rectangle, the guest allocates contiguous chunks of four, but the host only allocates in contiguous chunks of two, resulting in the effective contiguity of two. The first to pages will be translated by an anchor TLB entry with the contiguity of two, directly to hPA. However, the third and fourth page of the anchor block will need to take the two step translation as above. Finally, the best case is shown in the rectangle on the right. Both the guest and host allocate in equally sized blocks, resulting in an effective contiguity of four, and allowing a single anchor TLB entry to provide translation from the gPA to hPA for the four pages.

3.6.2 Guest and host coordinated memory mapping

To overcome the fragmented memory mappings resulting in poor virtualized TLB performance, we propose coordinated memory mapping between the guest and the host. Firstly, we propose for the guest and host to share the information of the mapping from the guest to the host. Secondly, we propose a contiguous memory allocation, where the OS proactively allocates memory based on the anchor distance

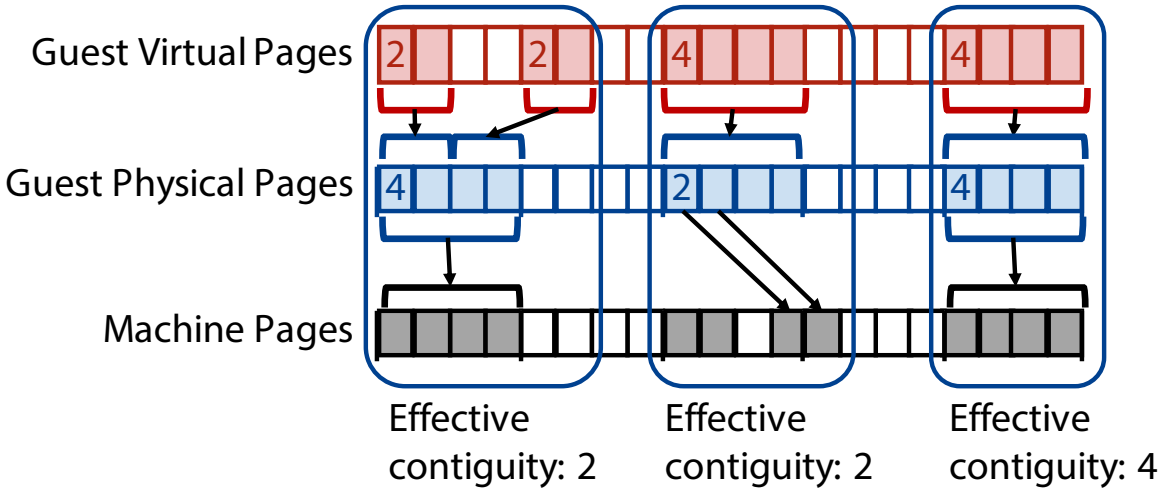


Figure 3.16: Contiguities are created at different level of the address hierarchy. The *e*ffective contiguity needs to be maximized

of the given process.

Linux relies on delayed paging, where the page is physically allocated only on the first touch of the page. To prevent this, memory allocation system call can be called with a special flag (`mmap(..., MAP_POPULATE)`) to cause a pre-fault behavior. However, pre-faulting requires changes to existing programs, which takes effort. Therefore, for currently deployed applications, the allocation happens at the first touch. With Transparent Huge Page (THP), large pages (2MB) are allocated instead of the regular (4KB) when conditions allow, such as alignment, size of the virtual allocation, etc. It is worth noting that as of this writing, the Linux kernel does not try in any way to provide contiguous mappings of more than the requested page.

When virtualization is introduced, such lazy page faulting occurs in two steps. On an access to a new page, the guest first faults. The guest resolves the page fault by allocating a guest physical frame to the guest virtual page. Then the memory access is retried and the host is faulted to map a actual memory page. When the host is faulted, the virtualization extension passes on the faulting address, and some access type related information. There are reserved bits available to use in the extra information register. In the case of Intel virtualization, *VMX exit information fields* and the *exit qualification* can be used to share allocation information from the guest to the host [31]. As the host page fault handler is invoked during a two-dimensional page walk, the MMU is aware of the guest page table. The MMU can access the anchor page table entry in the guest and extract the allocation contiguity and pass this through the aforementioned VMX registers. The host can use this information to try and make an allocation of the same size. If the allocation can be made, the allocated block of pages can be populated into the anchor TLB directly from the gVA to hPA. If the host fails to allocate the same amount of contiguous chunks, the anchor TLB will need to hold both gVA to gPA, and gPA to hPA mappings. Thus, it is favorable for the host to make a contiguous allocation identical to the allocation made by the guest for maximum anchor TLB efficiency.

When allocating memory for the virtual address space, it is favorable for contiguous allocations to be made. However, Linux does not provide any effort for such contiguous allocations. A recent work has explored enabling contiguous allocations in the Linux kernel [72]. With such support, if both guest and host allocates memory in contiguous blocks equal to the anchor distance, we can expect performance

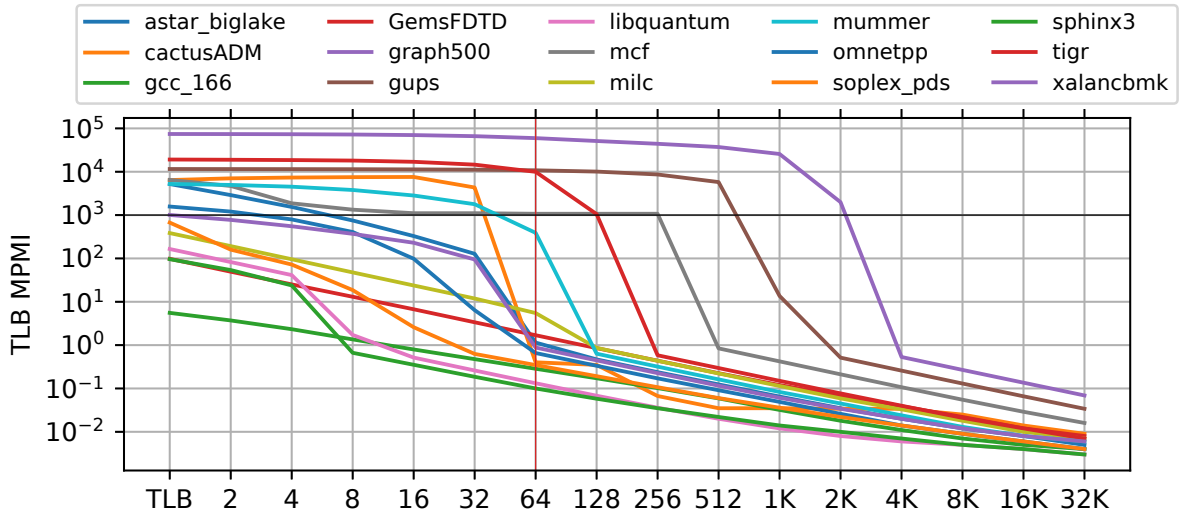


Figure 3.17: The MPMI of workloads with the favorable mappings enabled

gains. Figure 3.17 illustrates the potential TLB miss reduction (shown in misses per million instruction) with varying contiguity order. We assume that the OS is allocates memory in the same size as the anchor distance. The x-axis shows the baseline TLB, followed by allocations of varying anchor distances. We can see TLB miss reductions by employing larger contiguous allocations. This is because less anchor TLB entries are required to cover the entire memory space of the workload. In the case of `graph500`, `gups`, which had over 16GB and 8GB memory allocations, respectively, memory allocations of 64 pages (256KB) could result in significant TLB miss drops. With the exception of the two aforementioned workloads and `tigr` and `mcf` we could lower the TLB miss to below 1000 MPMI, or below 1 MPKI. The result of the figure are without THP support. If the OS can provide THP to some extent, we expect the miss curves to drop even quicker. Note that the TLB miss rate shown in the figure can apply to native systems as well.

The current independent memory allocation of the guest and host do not open up sufficient opportunity for the anchor TLB to provide sufficient performance benefits, especially when the host does not serve large pages. In this work, we propose coordinated memory allocation of the guest and host, and furthermore, proactive memory allocation of blocks of memory equal to the size of the anchor distance. With both coordinated allocations and memory allocation size is equal to the anchor distance, we have shown that anchor TLB can provide significant TLB miss reductions.

3.7 Related Work

There have been many prior work to improve different aspects of address translation to support virtual memory.

Improving TLB Coverage: Improving the TLB coverage has been a topic of research over the years [59, 58, 37, 38, 10, 17, 54, 55, 74]. Talluri and Hill proposed subblocking of TLB entries that allows a single TLB entry to represent multiple page mappings [67] within a subblock of pages. Subblocking has influenced this work and other work, including CoLT [59] and Cluster TLB [58]. In multi-core systems, data sharing across multiple cores is exploited to reduce TLB misses by collaboratively using shared translation entries [45, 14, 66]. Papadopoulou et al. proposed a prediction-based indexing approach to

support multiple page size with a single lookup when predictions hit [54]. Recently, Cox et al. proposed Mix TLB supporting multiple page sizes using a single indexing scheme [16]. With a single indexing for 4KB page, it intentionally allows multiple TLB entries of a large page to exist in the TLB. However, based on the observation that even superpages tend to be allocated consecutively by the OS, HW coalescing of superpage entries offsets the capacity waste caused by multiple TLB entries of the same large page.

Reducing TLB Miss Penalty and Prefetching: There are many parts of the page table walker that can be optimized to minimize the TLB miss penalty. The first set of studies reduces the page table walk overhead by improving the translation cache [13, 8], which minimizes the number of memory accesses to fetch intermediate page table nodes. Speculation can also be used to speed up the page table walking procedure [9]. By using the reservation scheme [53], it is possible to accurately interpolate the missing address by using the data that is available. Finally, prefetching can be used to proactively insert pages that may be used in the near future into the TLB [63, 35].

A different approach to reduce address translation is virtual caching. Virtual caching allows to defer address translation after cache misses. Several prior work on virtual caching save TLB power consumption by reducing TLB accesses significantly, or reduce page table walks as large on-chip caches can contain data which could have missed in TLBs of the conventional physical caching [74, 55, 73, 11].

Virtualized Systems: As virtual machines add an additional layer of address translation, TLB miss latencies are amplified, and thus the virtualized systems exhibit more severe performance drops by TLB misses, compared to native ones. Various prior work have tackled the translation challenge of virtualization. Gandhi et al. extended the segment-based translation to support nested translation of virtualized systems [20]. On the other axis, there were studies that improve the TLB miss latency handling by improving the translation cache [12] or improving the organization of the nested page tables [5, 21].

3.8 Conclusion

This paper proposed a novel HW-SW hybrid TLB coalescing technique. By encoding allocation contiguity information in page tables, highly coalesced address translation was possible, only with minor changes to the existing MMU designs. The hybrid approach not only provides high coverage scalability by using the OS support, but also allows adaptability to diverse memory fragmentation status. With various memory mapping scenarios, our experimental evaluation showed that the proposed scheme can provide translation performance better than or matching the best prior scheme for each scenario.

Chapter 4. Conclusion

4.1 Summary

This dissertation has explored the inefficiency of the current virtual memory system organization and proposed methods to improve the virtual memory address translation system. To improve the address translation system, this thesis leveraged two methods: skipping unnecessary translations, and dynamically improving the TLB coverage.

As we have shown previously, the first method, skipping unnecessary translation, realizes the use of virtual caching by allowing the cache hierarchy to be populated by either the virtual address or the physical address. The cache hierarchy is hybrid in that both types of cache addresses are used to populate the cache hierarchy. The majority of cache blocks are stored using the virtual address, enabling the use of virtual cache for 'safe' blocks. For memory regions that may cause synonyms the conventional physical addressing is used to store the synonym blocks. During a cache access, a synonym filter is employed to quickly distinguish between the virtual blocks and the synonym blocks. The resulting system is the *Hybrid Virtual Caching* that reaps the benefit of virtual caching, while falling back to physical caching to handle problematic synonyms.

We further reconfirmed a prior observation by Zhang et al. [74], that delaying translation to after the cache hierarchy for the virtual blocks can be beneficial for some workloads. The hits in the cache hierarchy allows skipping translation altogether. Furthermore, delayed translation provided opportunities to hit in the cache, where the translation entry was not present in the TLB. Such hits would have required a costly page walk due to a TLB miss in conventional system, however due to the larger coverage of the cache hierarchy, *HVC* did not require translating such entries. As the hits in the cache hierarchy allow skipping the corresponding TLB accesses, delaying translation significantly reduces the number of accesses (or cache misses) that need to be translated. Thus, delaying the translation also relaxes the timing constraint that was imposed on the TLB when the translation was required before the cache access. To this end, we proposed scalable delayed translation, a many segment based translation approach that is enabled by the relaxed translation latency. By efficiently utilizing the massive coverage offered by segments, and using thousands of segments, we could serve almost all the translations from without having to issue page walks to the DRAM. To conclude, delaying translations brings two opportunities: firstly, cache hierarchy can mask away significant amount of TLB lookups; secondly, the reduction of TLB lookups allows relaxing the TLB timing constraints, allowing architects to design more sophisticated but effective translation structures.

Shifting gears, the second approach to improve the address translation system was to increase the TLB coverage. We identified that even within the execution of the same applications, the mapping contiguity of each execution could change drastically, based on the other co-running processes memory allocation behavior. Previous TLB proposals assumed a certain memory mapping contiguity when trying to improve the TLB hit rate. However, as we observed, such mapping contiguity is not always guaranteed. We ventured to design a TLB system that could dynamically adapt to the changing mapping contiguity. We proposed a HW-SW co-design, *Hybrid TLB Coalescing*, that slightly changed the TLB structure, and entrusted the OS to mark contiguity information into the page table. The proposed translation lookaside buffer that we name `anchor TLB` provides a set associative structure where each entry holds

translations for multiples of regular pages.

We believe the virtual memory system provides benefits too great to overthrow the system itself. This dissertation has covered but a small aspect of the vast virtual memory system. However, we believe that our effort to improve the virtual memory address translation system will inspire other studies to be followed to improve the address translation. We hope that such research effort will allow the virtual memory system to adapt to the new emerging execution environments such as disaggregated memory systems, heterogeneous memory systems, systems with accelerators, and even more to come.

4.2 Future work

There looks to be even more opportunity for research in virtual memory system for the foreseeable future.

On the memory side, the increasing complexity of memory hierarchy introduced by high bandwidth memory and non volatile memory will result in more pressure on the virtual memory system. Furthermore, disaggregated or remote memory will add on to the complexity. The virtual memory system will be relied upon to provide efficient management of the memory resources. To provide optimal performance memory hotness and coldness need to be tracked for tiered memory systems. Recent work make use of expensive techniques such as selecting a few pages to sample and poisoning the pages, so every memory access result in OS page fault [3]. Some other implementations scan the page table for the access bits, clear the bits (also causing a TLB flush), wait for some time and repeat the process to gather page access information [44]. Some work use OS techniques and queue data structures to manage a pseudo LRU to move page between different memory [41, 71], however these techniques are approximations, made in essence due to the lack of accurate but low cost page hotness identification mechanism. It seems like the right time to research and propose efficient hardware support for hot/cold page detection. Additionally, some workloads do not exhibit locality (i.e. streaming workloads), which in turn makes them rather inappropriate for the tiered-memory systems. Hardware support to detect such workload would also be very beneficial. Furthermore, if the memory access pattern could be known in advance, prefetching pages that will be used into the fast memory could be beneficial for these workloads. Future work in improving the functionality of the virtual memory system to better support heterogeneous memory would be much appreciated.

With the introduction of data hungry data consumers such as GPUs, accelerators (both FPGAs and ASICs), and I/O devices getting even more potent, it will be critical as ever to serve these devices with as little disruption as possible. Thus, the virtual memory tax (i.e. address translation) needs to be as efficient as possible. As of this writing, the virtual memory tax imposes a significant overhead on such system. Due to the spatial locality of memory access within GPU threads of execution, even single page misses can stall eight warps out of the 64 warps on average, and up to 40 warps in the worst case [7]. Furthermore, irregular GPU workloads have been reported to run up to 3x slower due to the TLB [64]. Accelerators, using the main virtual memory via IOMMU and the IOTLB have been reported to suffer up to 8.1x, 6.4x due to the translation process [27, 26]. It is the precise time to be studying the virtual memory tax of address translation for these memory consumer devices.

The virtual memory system will be used for the foreseeable future, and more is being demanded of it. Looking forward, it is an opportune moment for even more work in this area!

Bibliography

[1] "W^x jit-code enabled in firefox," December 2015. [Online]. Available: https://jandemooij.nl/blog/2015/12/29/wx-jit-code-enabled-in-firefox/

[2] "W^x now mandatory in openbsd," May 2016. [Online]. Available: http://undeadly.org/cgi?action=article&sid=20160527203200

[3] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 631–644.

[4] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei, "Remote memory in the age of fast networks," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17. New York, NY, USA: ACM, 2017, pp. 121–127.

[5] J. Ahn, S. Jin, and J. Huh, "Revisiting hardware-assisted page walks for virtualized systems," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 476–487.

[6] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, "Biobench: A benchmark suite of bioinformatics applications," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, ser. ISPASS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 2–9.

[7] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 503–518.

[8] 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*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 48–59.

[9] 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*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 307–318.

[10] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 237–248.

[11] A. Basu, M. D. Hill, and M. M. Swift, "Reducing memory reference energy with opportunistic virtual caching," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 297–308.

[12] 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*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 26–35.

[13] A. Bhattacharjee, "Large-reach memory management unit caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 383–394.

[14] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level tlbs for chip multiprocessors," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, ser. HPCA '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 62–63.

[15] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[16] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 435–448.

[17] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem, "Supporting superpages in non-contiguous physical memory," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture*, ser. HPCA '15. IEEE, 2015, pp. 223–234.

[18] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data tiering in heterogeneous memory systems," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016, pp. 15:1–15:16.

[19] 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*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 37–48.

[20] 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*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 178–189.

[21] 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*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 707–718.

[22] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma, "Large pages may be harmful on numa systems," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 231–242.

[23] W. Glozer, "wrk - a HTTP benchmarking tool," Feb 2017. Available: https://github.com/wg/wrk

[24] J. R. Goodman, "Coherency for multiprocessor virtual address caches," in *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ser. ASPLOS II. Los Alamitos, CA, USA: IEEE Computer Society Press, 1987, pp. 72–81.

[25] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *14th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '17. Boston, MA: USENIX Association, 2017, pp. 649–667.

[26] Y. Hao, Z. Fang, G. Reinman, and J. Cong, "Supporting address translation for accelerator-centric architectures," in *2017 IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA '17, Feb 2017, pp. 37–48.

[27] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing memory in heterogeneous systems," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 637–650.

[28] "Intel optane dc persistent memory," website, Intel. Available: https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html

[29] *5-Level Paging and 5-Level EPT*, Intel, dec 2016.

[30] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Intel, Apr. 2019.

[31] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Intel, May 2019.

[32] B. Jacob and T. Mudge, "Software-managed address translation," in *Proceedings of the IEEE 3rd Symposium on High-Performance Computer Architecture (HPCA)*, 1997.

[33] B. Jacob and T. Mudge, "Uniprocessor Virtual Memory Without TLBs," *IEEE Trans. Comput.*, vol. 50, no. 5, pp. 482–499, May 2001.

[34] B. Jacob, S. Ng, and D. Wang, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann Publishers, 2008.

[35] G. B. Kandiraju and A. Sivasubramaniam, "Going the distance for tlb prefetching: An application-driven study," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ser. ISCA '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 195–206.

[36] S. Kannan, A. Gavrilovska, and K. Schwan, "pvm: Persistent virtual memory for efficient capacity scaling and object storage," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016, pp. 13:1–13:16.

[37] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant memory mappings for fast access to large memories," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 66–78.

[38] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Ünsal, "Energy-efficient address translation," in *2016 IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA '16. IEEE Press, 2016, pp. 631–643.

[39] S. Kaxiras and A. Ros, "A new perspective for efficient virtual-cache coherence," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 535–546.

[40] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-centric system interconnect design with hybrid memory cubes," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '13, Sept 2013, pp. 145–155.

[41] K. Koh, K. Kim, S. Jeon, and J. Huh, "Disaggregated cloud memory with elastic block management," *IEEE Transactions on Computers*, vol. 68, no. 1, pp. 39–52, Jan 2019.

[42] E. J. Koldinger, J. S. Chase, and S. J. Eggers, "Architecture support for single address space operating systems," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS V. New York, NY, USA: ACM, 1992, pp. 175–186.

[43] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and efficient huge page management with ingens," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 705–721.

[44] M. Lespinasse, "idle page tracking / working set estimation," Sep 2011. Available: https://lwn.net/Articles/459269/

[45] Y. Li, R. Melhem, and A. K. Jones, "Leveraging sharing in second level translation-lookaside buffers for chip multiprocessors," *IEEE Computer Architecture Letters*, vol. 11, no. 2, pp. 49–52, 2012.

[46] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 267–278.

[47] Linux Kernel Documentation, "pagemap, from the userspace perspective," May 2016. Available: https://www.kernel.org/doc/Documentation/vm/pagemap.txt

[48] Linux Kernel Documentation, "Transparent hugepage support," Mar 2017. Available: https://www.kernel.org/doc/Documentation/vm/transhuge.txt

[49] G. H. Loh, "3d-stacked memory architectures for multi-core processors," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 453–464.

[50] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200.

[51] 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*, ser. HPCA '15, 2015, pp. 126–136.

[52] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," Tech. Rep., 2009.

[53] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating system support for superpages," vol. 36, no. SI. New York, NY, USA: ACM, Dec. 2002, pp. 89–104.

[54] M. M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-based superpage-friendly tlb designs," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture*, ser. HPCA '15, 2015, pp. 210–222.

[55] C. H. Park, T. Heo, and J. Huh, "Efficient synonym filtering and scalable delayed translation for hybrid virtual caching," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 217–229.

[56] A. Patel, F. Afram, S. Chen, and K. Ghose, "Marss: A full system simulator for multicore x86 cpus," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. New York, NY, USA: ACM, 2011, pp. 1050–1055.

[57] J. T. Pawlowski, "Hybrid memory cube (hmc)," in *2011 IEEE Hot Chips 23 Symposium (HCS)*, Aug 2011, pp. 1–24.

[58] 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*, ser. HPCA '14, 2014, pp. 558–567.

[59] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 258–269.

[60] "Postgresql: Documentation: 10: pgbench," PostgreSQL. Available: https://www.postgresql.org/docs/10/pgbench.html

[61] 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*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 24–33.

[62] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, Jan 2011.

[63] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-based tlb preloading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. New York, NY, USA: ACM, 2000, pp. 117–127.

[64] S. Shin, M. LeBeane, Y. Solihin, and A. Basu, "Neighborhood-aware address translation for irregular gpu applications," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '18, Oct 2018, pp. 352–363.

[65] A. Sodani, "Race to exascale: Challenges and opportunities," MICRO 2011 Keynote.

[66] S. Srikantaiah and M. Kandemir, "Synergistic tlbs for high performance address translation in chip multiprocessors," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '10, 2010, pp. 313–324.

[67] 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*, ser. ASPLOS VI. New York, NY, USA: ACM, 1994, pp. 171–182.

[68] W. H. Wang, J.-L. Baer, and H. M. Levy, "Organization and performance of a two-level virtual-real cache hierarchy," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, ser. ISCA '89. New York, NY, USA: ACM, 1989, pp. 140–148.

[69] D. H. Woo, M. Ghosh, E. Özer, S. Biles, and H.-H. S. Lee, "Reducing energy of virtual cache synonym lookup using bloom filters," in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES '06. New York, NY, USA: ACM, 2006, pp. 179–189.

[70] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, and J. M. Pendleton, "An in-cache address translation mechanism," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ser. ISCA '86. Los Alamitos, CA, USA: IEEE Computer Society Press, 1986, pp. 358–365.

[71] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 331–345.

[72] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Translation ranger: Operating system support for contiguity-aware tlbs," in *Proceedings of the 46th Annual International Symposium on Computer Architecture*, ser. ISCA '19, 2019.

[73] H. Yoon and G. S. Sohi, "Revisiting virtual l1 caches: A practical design using dynamic synonym remapping," in *2016 IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA '16, 2016, pp. 212–224.

[74] 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*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 159–168.