REDUCING ADDRESS TRANSLATION OVERHEADS WITH VIRTUAL CACHING

by

Hongil Yoon

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2017

Date of final oral examination: 3/3/2017

The dissertation is approved by the following members of the Final Oral Committee:
Gurindar S. Sohi (Advisor), Professor, Computer Sciences
Mark D. Hill, Professor, Computer Sciences
David A. Wood, Professor, Computer Sciences
Karthikeyan Sankaralingam, Associate Professor, Computer Sciences
Mikko H. Lipasti, Professor, Electrical and Computer Engineering

Dedicated to Sohyun, Mom and Dad, for their unconditional love and support.

ACKNOWLEDGMENTS

There are numerous people who have supported me throughout my graduate school journey. Without their unconditional support, I would never have finished my dissertation.

First and foremost, I would like to thank my advisor, Prof. Gurindar S. Sohi. From him, I learned how to approach a challenging problem, how to organize my thoughts, and how to clearly present my ideas in an efficient manner. He patiently encouraged me to find my research topic described in this dissertation. He was also generous with financial support and advice not only on my research but also on life in general.

I would like to thank the other professors on my dissertation committee, including Mark D. Hill, Mikko H. Lipasti, Karthikeyan Sankaralingam, and David A. Wood. In particular, David Wood and Mark Hill gave me insightful comments on my virtual cache proposal in terms of practicality and efficacy. Mikko Lipasti taught me the fundamentals of modern general purpose processors, which led me to think of being a computer architect. Karu's intensive feedback improved the quality of my papers. In addition, I was fortunate to take a course of performance analysis by Prof. Mary K. Vernon. She spent her own time to help me make up pre-requisite courses I had not taken. I was impressed by her enthusiasm.

I have learned a lot from my academic siblings. Matthew Allen helped me get used to being a member when I joined Multiscalar group. Gagan Gupta helped me learn how to approach a problem and organize answers when I had a hard time in a qualification exam. He also gave me insightful comments for my research topics. Srinath Sridharan was a great friend. He was generous with his advice on my research. I also thank him for his unconditional support of my job search, especially for Google which is my first employment. Philip Wells and Jichuan Chang gave me great comments on my research

presentations at the annual architecture affiliate meeting.

I also have benefited from interacting with other friends in computer architecture. First of all, I would like to thank all the members of our qual study group - Lena Olson, Jason Lowe-Power, and Marc Orr - for great discussions about papers on the architecture reading list. Jayneel Gandhi and I discussed design implications of virtual caching for virtual memory features. I came by his office when I needed a break. Jason, Rathijit, Lena, Jayneel, and I talked a lot about non-architecture topics as well. Nilay Vaish gave me comments on implementation of my research projects based on gem5 simulator. We also prepared job interviews together. Clint Lestourgeon bugged me a lot whenever he needed a break. We talked about diverse topics from architecture research to console games. He saved me so many times in an FPS game, called Destiny.

I would also like to thank other colleagues, including Muhammad Shoaib bin Altaf, Newsha Ardalani, Raghuraman Balasubramanian, Arka Basu, Chris Feilbach, Vinay Gangadhar, Dibakar Gope, Venkatraman Govindaraju, Swapnil Haria, Eric Harris, Joel Hestness, Derek Hower, Sungjin Kim, Marc de Kruijf, Mitchell Manar, Michael Mishkin, Jaikrishnan Menon, Tony Nowatzki, Sankaralingam Panneerselvam, Somayeh Sardashti, Rathijit Sen, Vijay Thiruvengadem, and Aditya Venkataraman.

I would like to thank my girlfriend, Sohyun Kang, for her love and support during my Ph.D. program. Without her, I could not finish my dissertation. I also would like to thank my parents for the love and support that they have given me during my entire life. Also, my brother, Hongin Yoon, and sister-in-law, Yoonsun Chung, have mentally supported me. Meeting my adorable nephew and niece - Junu Yoon and Junyeong Yoon - over a video chat has always made me encouraged.

TABLE OF CONTENTS

ABSTRACT

This dissertation research addresses overheads in supporting virtual memory, especially virtual-to-physical address translation overheads (i.e., performance, power, and energy) via a Translation Lookaside Buffer (TLB). To overcome the overheads, we revisit virtually indexed, virtually tagged caches. In practice, they have not been common in commercial microarchitecture designs, and the crux of the problem is the complications of dealing with virtual address synonyms.

This thesis makes novel, empirical observations, based on real world applications, that show temporal properties of synonym accesses. By exploiting these observations, we propose a practical *virtual cache design with dynamic synonym remapping* (VC-DSR), which effectively reduces the design complications of virtual caches. The proposed approach (1) dynamically decides a unique virtual page number for all the synonymous virtual pages that map to the same physical page and (2) uses this unique page number to place and look up data in the virtual caches, while data from the physical page resides in the virtual caches. Accesses to this unique page number proceed without any intervention. Accesses to other synonymous pages are dynamically detected, and remapped to the corresponding unique virtual page number to correctly access data in the cache. Such remapping operations are rare, due to the temporal properties of synonyms, allowing our proposal to achieve most of the benefits (i.e., performance, power, and energy) of virtual caches, without software involvement.

We evaluate the effectiveness of the proposed virtual cache design by integrating it into modern CPUs as well as GPUs in heterogeneous systems. For the proposed L1 virtual cache of CPUs, the experimental results show that our proposal saves about 92% of dynamic

energy consumption for TLB lookups and achieves most of the latency benefit (about 99.4%) of ideal (but impractical) virtual caches. For the proposed entire GPU virtual cache hierarchy, we see an average of 77% performance benefits over the conventional GPU MMU.

Chapter 1

INTRODUCTION

Virtual memory and caches are two of the most common elements of computers today. Most modern computer systems take advantage of fast caches to increase the performance of a processor, and virtual memory provides access protection, an illusion of large physical address space, efficient memory management, and support of data sharing. However, there is no free lunch.

A processing core generates virtual addresses. Then the virtual addresses should be translated to the corresponding physical addresses to access the appropriate data in caches or memory. Operating system (OS) page tables maintain virtual-to-physical address translations. However, accessing the OS page table on every memory access is not latency and energy efficient because it requires multiple memory accesses.[1] Hence, an efficient address translation is necessary to support virtual memory.

To mitigate such overheads, modern processors have Memory Management Units (MMUs) keeping track of frequently used address translations in a small cache, called a Translation Lookaside Buffer (TLB) [28, 34]. In practice, they usually use physical addresses for cache accesses, and thus the TLBs are consulted in parallel with (or prior to) every L1 cache lookup. TLB accesses consume significant power [13, 96] and add latency to the cache access. The TLBs are also hotspots because of the high-power dissipation [86]. Due to the long TLB miss latency, in addition, frequent TLB misses significantly degrade performance. Over the years, a plethora of techniques have been proposed and deployed to solve the challenges due to the use of TLBs for address translations [11, 14, 51, 56, 81, 99]. However, the fundamental problem remains in physical caches—where a physical address is used

[1] Regarding future big memory workloads, some processor vendors and OS communities have started to discuss supporting 5-level paging [47] (e.g., previously 4-level paging on x86).

for cache access.

This dissertation research addresses overheads in supporting virtual memory, specifically virtual-to-physical address translation overheads (i.e., latency, power, and energy) due to TLB accesses. To do so, we revisit *virtually indexed, virtually tagged caches* (hereafter "virtual caches"). This dissertation makes several empirical observations for temporal characteristics of virtual address synonyms complicating the deployment of virtual caches. By leveraging these observations, a practical virtual cache design is proposed, not only for modern CPUs, but also for a heterogeneous computing environment supporting shared address space between CPUs and a GPU.

1.1 Emerging Opportunities for Virtual Caches

Virtual caching has been proposed many times over the past several decades as a way of reducing the impacts of address translation [13, 40, 57, 78, 88, 104, 109]. The use of virtual caches can *lower the access latency and energy consumption of address translations* compared to physical caches, because the virtual-to-physical address translation via a TLB is required only when a cache miss occurs. Thus, virtual caches act to filter TLB lookups. Furthermore, virtual caches also filter TLB misses when the virtual caches hold lines for which the matching translation is not found in the TLB [110]. That is, virtual caching can play the roles of a **TLB lookup filter** and a **TLB miss filter**.

There are two main reasons to revisit virtual caches now. First, due to the breakdown of Dennard scaling [29, 30, 45], power and energy are major constraints of processor designs [72]. The virtual-to-physical address translations, especially for accessing TLBs on cache accesses, are power and energy hungry, which contributes considerably to the power dissipation of modern processors. Second, heterogeneous computing on tightly-integrated CPU-GPU systems is ubiquitous, and to increase programmability, many of these systems support virtual address accesses from GPU hardware. Supporting virtual memory for the

GPU entails address translations on every memory access. However, simply reflecting the CPU-style memory management unit (MMU) in GPUs is not effective because of differences in the GPU microarchitecture, as well as in workloads. This greatly impacts performance (up to 4.5× slowdown, refer to Section 5.5). *These challenges are described more in detail below, creating ample opportunities for employing virtual caching as TLB lookup and TLB miss bandwidth filters.*

1.1.1 Emergence of Power and Energy Challenges

Over the years, computer architects have focused on improving the performance of processors. It has been very successful by taking advantage of Moore's Law [71]; more available transistors are employed to introduce innovative techniques such as Out of Order design, speculations, caches, etc. However, scaling down semiconductor technologies has faced serious challenges due to the failure of Dennard Scaling, that is a fundamental driver of Moore's Law [29, 30, 45]. Accordingly, power and energy are now major constraints, imposing restrictions on the innovations of processor designs [72].

Modern processors support virtual memory and perform address translations via a TLB on cache accesses (e.g., instruction fetches, load and store instructions for data). To avoid potential latency overhead, the address translation via a TLB has to be performed fast. In addition, the TLB needs to be built on a highly associative cache (e.g., a fully associative array) to reduce TLB miss overheads (i.e., multiple memory accesses for traversing the OS page table). These aspects make TLB accesses power and energy hungry. Figure 1.1 presents a breakdown of the power consumption of a processor including caches from an industry report [96]. We can see up to 13% of the core power dissipation is due to TLB lookups. In addition, the TLBs are hotspots in a processor because of the high-power dissipation [86].

These issues are well-known to processor vendors. A plethora of techniques have been proposed and deployed to address them [11, 14, 51, 56, 81, 99]. However, they fall short of

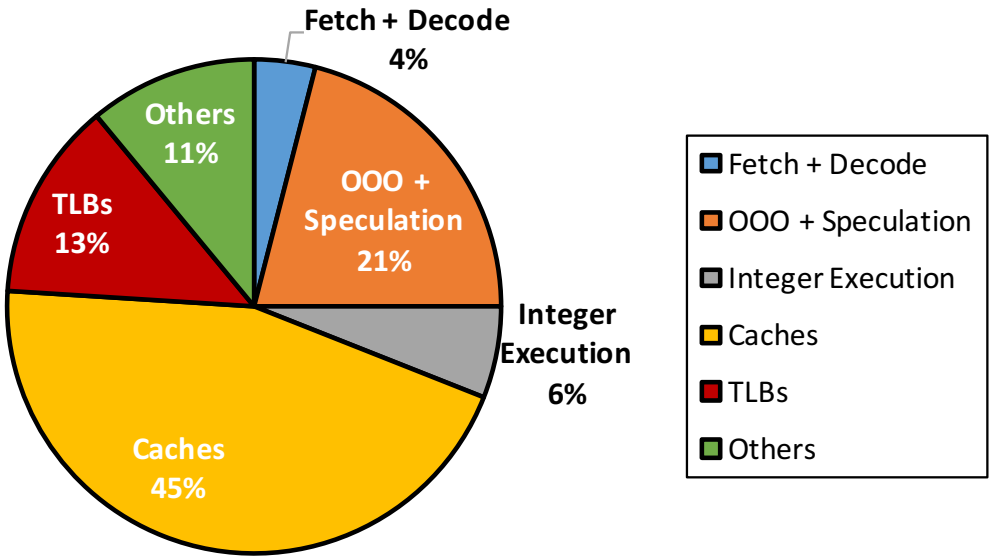


Figure 1.1: Breakdown of power consumption of a general-purpose processor (by Intel [96])

reducing both the latency and energy impacts of the TLB access. We believe that efficient virtual caching can be a practical way of reducing all of the impacts by serving as a TLB lookup filter.

1.1.2 Proliferation of Heterogeneous Computing

In recent years, specialized hardware accelerators have proliferated to achieve higher performance and lower power consumption. To that end, GPUs integrated onto the same chip as CPUs are now first-class computational devices. Many of these computational engines have full support for accessing memory via traditional virtual addresses [62]. This allows programmers to simply extend their applications to use GPUs without the need to explicitly copy data or transform pointer-based data structures. Furthermore, GPU programs can execute correctly, even if they depend on specific virtual memory features like demand paging or memory-mapped files.

However, virtual memory support for the GPU does not come for free. It also requires

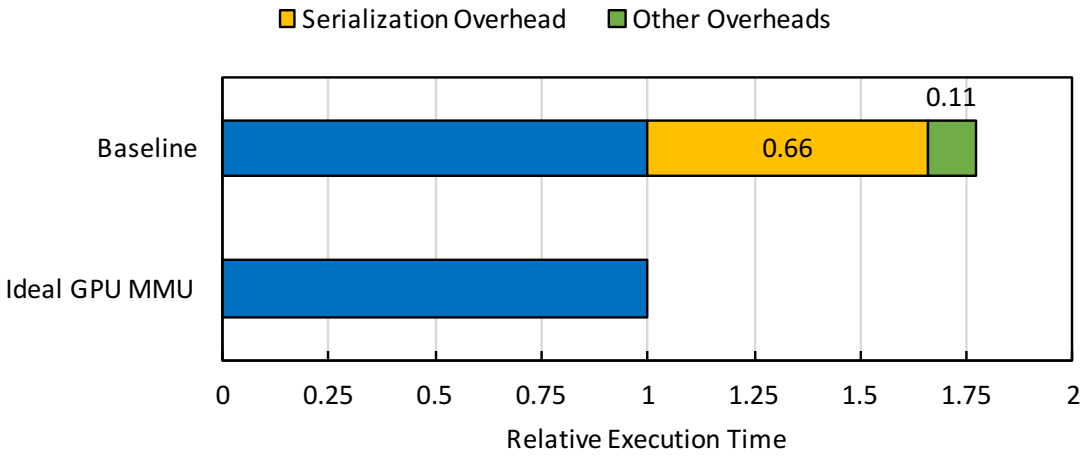


Figure 1.2: Breakdown of GPU address translation overheads. Integrated CPUs and GPUs with fully unified (shared) address space support is considered.

translating virtual addresses to physical addresses on every cache access. The overhead has significant performance and energy impacts [12, 14, 81, 96]. Conventional CPUs mitigate the overhead by employing translation lookaside buffers (TLBs). However, simply reflecting the CPU-style memory management unit (MMU) in GPUs is not effective because of differences in GPU microarchitecture, as well as in workloads.

GPUs have less locality—both spatial and temporal—in their address streams than CPUs and a higher rate of memory accesses for the following two reasons. First, GPUs have many more execution contexts than CPUs. Within one GPU computational unit, there are 10's of unique execution contexts (depending on the exact microarchitecture), each of which is capable of issuing a unique stream of memory requests. Second, each execution context has many "threads" or "lanes". A single static GPU load or store instruction can issue requests to 10's of different addresses, and these addresses could even be to different virtual memory pages! This high access rate and poor locality result in frequent TLB misses, which has significant performance and energy impacts.

Preliminary studies have shown this overhead can be much worse on GPU architectures than on CPUs, with one study showing TLB misses taking 25× longer on GPUs than on

CPUs [102]. Our empirical data mirrors these previous findings and shows that GPU TLBs have very high miss ratios (average 56%) and high miss bandwidth (e.g., more than 2 misses per cycle in some cases). Figure 1.2 shows an average overhead of GPU address translation when a practical translation implementation is used for the evaluated workloads, relative to the system with an ideal GPU MMU. We can see an average of 77% performance degradation over an ideal GPU MMU, and this translation overhead is higher for emerging GPU applications (e.g., graph-based workloads) than traditional GPU workloads. We identify that the major source of this overhead is the serialization delays (i.e., a yellow bar) at address translation hardware (e.g., a shared TLB) due to frequent private TLB misses.

We empirically observe that more than 60% of references that miss in the private GPU TLBs find corresponding data in the GPU cache hierarchy (i.e., private L1s and a shared L2 cache). By taking advantage of the property that no address translation is needed when valid data resides in virtual caches [110], we believe that efficient virtual caching as a TLB miss bandwidth filter can be a practical way of resolving issues resulting from the limited bandwidth of the shared TLB.

1.2 Thesis Statement

The confluence of these recent and emerging challenges described above has led to the following thesis statement:

> *Future microarchitecture designs should support virtual caches as effective TLB lookup and miss bandwidth filters, due to i) the emerging challenges of power constraints on microarchitecture designs and ii) the proliferation of heterogeneous computing requiring high bandwidth for address translations due to microarchitecture and workload differences.*

1.3 Thesis Proposal

To advocate for this statement, in this dissertation, we propose and evaluate a practical virtual cache design, called VC-DSR: *Virtual Cache with Dynamic Synonym Remapping*. The design leverages the *temporal* properties of synonyms that are empirically quantified in this dissertation (Section 3.3).

The temporal characteristics of synonyms suggest that it is practical to design a cache hierarchy with a relatively small upper-level virtual cache, with physical caches for larger lower-levels, because synonyms are short-lived in smaller caches. Furthermore, the virtual cache design (1) dynamically decides a unique virtual page for all the synonymous virtual pages that map to the same physical page and (2) uses this unique page to place and look up data in the virtual caches, while data from that physical page resides in the virtual cache. Accesses to this unique page proceed without any intervention. Accesses to other synonymous pages are dynamically detected, and remapped to the corresponding unique virtual page to correctly access data in the cache. Such remapping operations are rare, due to the temporal properties of synonyms, allowing a VC-DSR to achieve most of the latency and energy benefits of virtual caches in a software-transparent manner. In addition, the use of a unique virtual address for all the operations of a virtual cache prevents potential synonym issues, which greatly simplifies the design of our proposal, as well as the overall memory hierarchy.

Regarding modern CPUs, we first propose a virtual L1 cache design based on the VC-DSR (Chapter 4) and employ it as a **TLB lookup filter** to reduce the power and energy overheads of TLB accesses. Additionally, this thesis takes another important step in reducing virtual-to-physical address translation overheads of an integrated CPU-GPU system supporting a unified virtual address space (Chapter 5). GPUs are most commonly used as accelerators, not general-purpose processors. This fact simplifies the virtual cache implementation, as it reduces the likelihood of virtual cache complications (e.g., synonyms). Furthermore, this easily enables the scope of the GPU virtual caching to be extended to

more of the cache hierarchy (including a shared GPU L2 cache). This allows us to take more advantage of virtual caching as a **bandwidth filter of TLB misses**.

1.4 Contributions

This dissertation makes the following main contributions:

- **Anatomy of sources of virtual address synonyms:** This thesis describes sources of virtual address synonyms in modern systems by exemplifying relevant programming and system practices. We discuss interactions causing synonyms between user and kernel operations, and they are classified into three groups as follows: synonyms due to i) activities in user spaces, ii) user-kernel interactions, and iii) activities in kernel space. This classification will be referred to in the rest of this dissertation to characterize synonyms of tested applications.

- **Challenges of supporting virtual memory with virtual caches:** Regarding the deployment of virtual caches, we discuss problems in terms of each aspect of virtual memory: (1) changes in virtual to physical address mapping, (2) access permission checking, and (3) virtual address synonym (due to data sharing). Especially for problems due to virtual address synonyms (i.e., the third one), we also consider specific microarchitectural details that are critical to a viable commercial implementation of any virtual cache, e.g., memory consistency issues and consistency issues of buffered instructions due to self-modifying codes. Prior literature has not discussed them.

- **Temporal behavior of synonyms:** In addition, we raise an important distinction between synonym accesses and *active* synonym accesses depending on whether a synonym access can cause actual problems in virtual caches. Regarding this, the dissertation presents novel empirical observations for the temporal behavior of vir-

tual address synonyms in caches, based on real world server, mobile, and more conventional (SPEC) workloads.

To present an accurate analysis, a full-system simulator is employed to consider all sources of virtual address synonyms such as access to dynamically linked libraries and access to user/kernel address space. Some prior works also analyzed synonyms behavior [13, 78]. However, they were based on instrumentation tools [66]. While relatively fast, it is an inappropriate approach for analyzing synonym behavior because it provides inadequate full-system support. For example, user-kernel interactions that are a major source of synonyms are not considered by other techniques.

- **Practical virtual cache design for CPUs:** By leveraging the temporal properties of synonyms, a virtual L1 cache using dynamic synonym remapping **(VC-DSR)** is proposed for modern CPUs as a *TLB lookup filter*. We show how our proposal efficiently solves the challenges of supporting virtual caches. This substantiates that VC-DSR is a practical solution to a problem that has long vexed computer architects: achieving most of the benefits of virtual caches, without *any* software involvement.

- **Analysis of GPU memory access patterns:** For diverse GPU workloads (e.g., conventional workloads like Rodinia [24] and emerging graph-based workloads like Pannotia [23]), TLB access patterns are studied, which substantiates that the GPU virtual cache hierarchy can filter out frequent private TLB misses to a shared TLB.

  In addition, we empirically analyze the synonym behaviors of these workloads and describe the unique characteristics of GPU programs that allow our design to efficiently handle virtual address synonyms.

- **Practical GPU virtual cache hierarchy based on VC-DSR:** By leveraging these observations, we propose the first pure virtual cache hierarchy for GPUs based on VC-DSR. The proposed design allows the flexibility for GPUs to keep their unique cache architectures (e.g., L1 caches without support for cache probes). And it demonstrates

using virtual caching for the entire GPU cache hierarchy is a practical, efficient *TLB miss bandwidth filter*.

Previous works [82, 85] mainly focus on how to support GPU address translation, while our proposal aims to reduce the overhead of the address translation.

1.5 Dissertation Outline

The remainder of this dissertation is organized as follows:

Chapter 2 provides background information of different cache organizations with respect to how caches are indexed and tagged (e.g., physically indexed, physically tagged cache, virtually indexed, physically tagged cache, etc.) and pros and cons of each design with respect to address translation via a TLB. And then, we briefly review virtual memory and study design complications of supporting virtual memory features with virtually indexed, virtually tagged caches, especially for *virtual address synonyms*.

Chapter 3 first describes sources of virtual address synonyms, and presents an empirical evaluation of the temporal nature of synonyms that will be referred to in the rest of this dissertation.

Motivated by this data, in **Chapter 4**, the rationale of the VC-DSR is discussed. And then we discuss the overall design of the VC-DSR and details of the hardware structures needed to ensure correct operations for virtual caching in CPUs. Also, the effectiveness is evaluated.

Chapter 5 discusses the integration of the proposal into a GPU virtual cache hierarchy for heterogeneous computing. First, GPU address translation and its overhead is discussed. Next empirical observations of GPU memory access patterns causing the significant overhead of GPU address translation is presented. To mitigate the overhead, the design of a GPU virtual cache hierarchy is proposed. Finally, we present the evaluation.

In **Chapter 6**, related works on prior virtual cache design and other approaches reducing

the overheads of address translations are discussed.

Chapter 7 concludes this thesis with future directions.

Chapter 2

BACKGROUND

In this chapter, we first review different cache designs in terms of how the cache access (i.e., indexing and tag matching) is carried out. Then, we discuss why *virtually indexed, virtually tagged (VIVT) caches* are compelling relative to other designs by comparing their potential overheads with respect to address translation via a TLB. The rest of the sections overview aspects of virtual memory. Regarding each feature of virtual memory, we discuss the design complications of VIVT caches.

2.1 Caching and Cache Organization

Cache memory is one of the most common elements of computers. It was first introduced with the IBM System/360 Model 85 [65]. The cache memory is a small, but high speed buffer. This buffer keeps track of useful data which is frequently referenced (in the future) by leveraging the temporal and spatial locality of memory access patterns [31]. Therefore, the cache memory enables latency and energy efficient data accesses by reducing the speed gap between a fast processor and slow main memory.

In practice, modern processors with a cache hierarchy also support virtual memory. A processing core generates virtual addresses, and thus, virtual-to-physical address translation needs to be performed at some point prior to main memory access with a physical address. Depending on when the address translation is carried out, different types of addresses between virtual and physical addresses are used for the indexes and tags of the cache access.

Accordingly, there are four possible combinations of cache organization, as follows: 1)

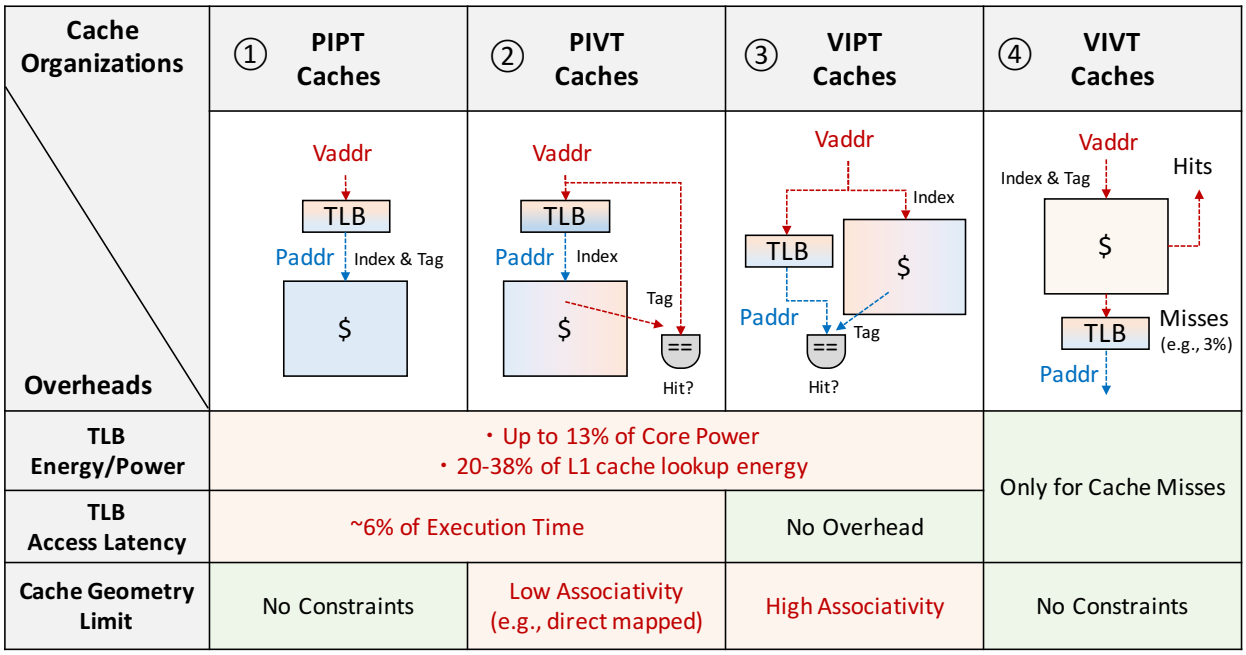


Figure 2.1: Schematic overview of four cache organizations according to how they are indexed and tagged, and analysis of their address translation overheads via a TLB.

physically indexed, physically tagged caches, 2) physically indexed, virtually tagged caches, 3) virtually indexed, physically tagged caches, and 4) virtually indexed, virtually tagged caches. Figure 2.1 shows the schematic overview of each cache design and summarizes potential power, energy, and latency overheads resulting from address translation via a TLB. The details will be described below.

1. **Physically Indexed, Physically Tagged (PIPT) caches.** First, PIPT caches use only physical addresses for cache access. Hence, virtual-to-physical address translation should be performed, via a TLB, prior to cache access. This is a power and energy hungry operation, which contributes considerably to the power dissipation of modern processors (up to 13% of the core power dissipation including caches, from an Intel report) [96]. In addition, the TLBs are hotspots in a processor because of the high power dissipation [86]. The operation also adds latency to the cache access. Our experiment shows that more than about 6% of the execution time is wasted on

consulting TLBs before cache lookups.

2. **Physically Indexed, Virtually Tagged (PIVT) caches.** With a PIVT cache, a physical address is employed to decide a set index of a cache, and thus the address translation via a TLB is also needed prior to cache access, like PIPT caches. Hence, PIVT caches have overheads which are similar to those PTPT caches have in terms of address translation via a TLB. However, the use of a virtual tag could lead to synonym issues in the same set. That is, it is possible for the same data to be cached with different virtual addresses in the same set; the complications of synonyms will be discussed in detail later in Section 2.2.2. This will impose additional constraints on the cache geometry, e.g., a direct mapped cache, to avoid the problems.

3. **Virtually Indexed, Physically Tagged (VIPT) caches.** VIPT caches use virtual addresses for indexing a cache, while tag matching employs physical addresses. Hence, TLB lookup does not have to be performed prior to cache lookup, unlike physically indexed caches (i.e., PIPT and PIVT caches). Consulting a TLB can be performed in parallel with the cache indexing with a virtual address generated by a processor, and the corresponding physical address obtained from the TLB is used to do tag comparison. Doing so removes TLB accesses on a critical path of memory access, which hides the latency overhead due to TLB lookups, making them a very popular choice in ubiquitous commercial designs.

   However, VIPT caches still consume power and energy due to TLB lookups for tag comparison on cache accesses. In addition, this design also has a design constraint on the cache geometry. Due to indexing with virtual addresses, it is possible for the same data be cached with different virtual addresses in different sets in the cache. To avoid this synonym data duplication issues (see details in Section 2.2.2), the page offset, which is not changed with a virtual-to-physical address translation, should be employed as cache indexing bits. And thus the size of each way should be smaller

than a page size. This restricts the cache geometry (i.e., requiring a larger associativity) [13, 49, 80]. For example, a 32KB cache with the size of a 64B cache line and 4KB page will have at least 8 ways. This significantly affects not only access latency but also the power and energy consumption of the cache access [43].

Over the years, a plethora of techniques have been proposed and deployed to reduce the latency and energy impacts of the (prior) TLB access [26, 33, 35, 51, 53–55, 63, 108]. However, *the basic problem still remains in physical caches, where a physical address is used for cache access (i.e., PIPT, PIVT, VIPT caches).*

4. **Virtually Indexed, Virtually Tagged (VIVT) caches.** Different from the other cache designs discussed above, VIVT caches use only virtual addresses for cache access. They have potentially *lower access latency and energy consumption* than the physical caches described above because a TLB is consulted only for cache misses (e.g., 3% of cache accesses, see the last column of Figure 2.1). In addition, VIVT caches allow a flexible cache geometry by removing the large associativity constraint of a VIPT cache design [43]. This can be more power and energy efficient for cache accesses. However, the use of virtual addresses for cache accesses leads to synonym problems, which complicates the cache design.

Figure 2.2 presents examples of a variety of general purpose processor designs with different cache organizations according to the four classifications discussed above. The figure considers some designs whose cache organizations are described in conference proceedings, journals, software developer manuals or technical reference manuals of processor vendors, etc. Actually, there are a lot of other processor designs that are not presented in this figure, especially for *VIPT and PIPT caches*. The trends are what we want to highlight.

Initially, processor designs supporting virtual memory used PIPT cache designs since the late 1960s, when a TLB was introduced. To avoid the latency overhead of address

translation, virtually indexed cache designs (VIPT or VIVT caches) had been proposed and studied. Since then, for the last several decades, most processor vendors, e.g., Intel, AMD, Sun, ARM, IBM, etc., take advantage of VIPT cache designs for their processor designs; for example, some ARM Cortex designs normally use a PIPT cache design for data caches[1], while a VIPT cache design is used for an instruction cache, in general.

Figure 2.2 shows that a few processor designs attempted to use VIVT designs. Despite the fact that VIVT caches are more latency and power/energy efficient relative to other cache designs using physical addresses for cache lookups, they have not been practically used in commercial designs. The primary reason is the complications due to virtual address synonyms, which will be discussed in detail in Section 2.2. Actually, these complications may be dealt with if demands are placed on OSes (or the full system stack from processor designs to the OSes). Note that most of the designs before 1990s require OS involvement. However, as processor designs have become ubiquitous, it is increasingly difficult to place demands on software over which a processor design/vendor has little or no control.[2] Accordingly, a *software-agnostic, pure hardware solution* that reduces the complications due

[1]They can also be non-aliasing virtually indexed, physically tagged (VIPT) caches in some designs [6].
[2]Currently, Apple may be the only example that has the access to the full stack of the system.

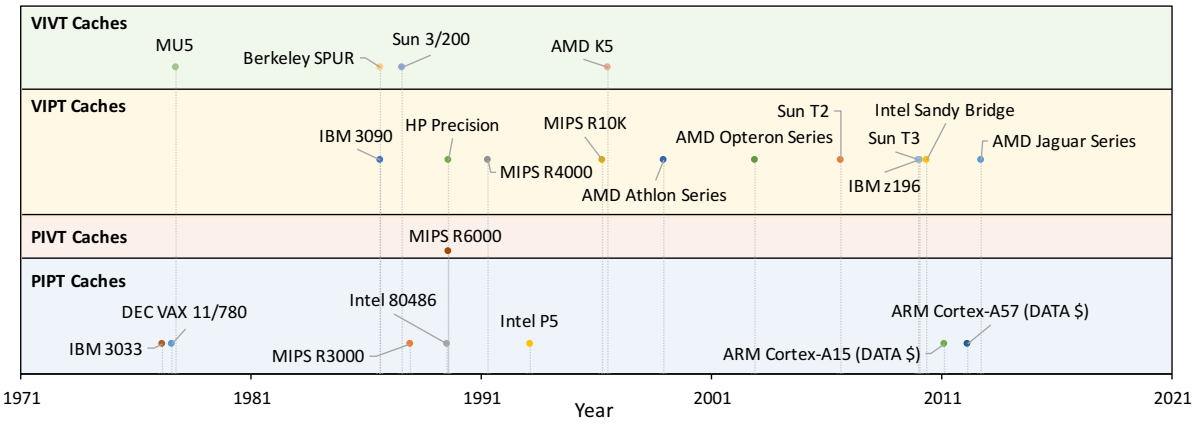


Figure 2.2: Examples of general purpose processor designs with different cache organizations. Each row presents different (series of) processors in the same category over time.

to synonyms is essential.

To the best of our knowledge, there is one example of using a PIVT cache: MIPS R6000 microarchitecture [90]. To reduce the space and latency overheads of a conventional TLB, the design introduced a TLB slice [100] that is a small direct mapped cache (e.g., 8 entries). It is indexed with a few least significant bits of a virtual page number, and the corresponding entry simply provides a few bits of a physical page number, without a virtual tag comparison. The TLB slice, while relatively fast and power/energy efficient, could lead to more cache misses due to the likelihood of providing a wrong set index for the current access, which complicates cache miss operations.

In the next section, we will first review diverse aspects of virtual memory. Then, we will delve into design issues of VIVT caches which are associated with each aspect of supporting virtual memory. They complicate the cache design and impair the potential benefits of virtual caches, and thus efficiently resolving the issues is critical to viable commercial implementation of any virtual cache.

2.2 Supporting Virtual Memory with the Virtually Indexed, Virtually Tagged Cache

As discussed above, despite the desirability of VIVT caches [19, 41], virtual caches have not been considered to be practical. The crux of the problem preventing the practical use of VIVT caches is the complications resulting from supporting virtual memory, especially for virtual address synonyms. To fully understand the problems, we first review the aspects of the uses of virtual memory.

2.2.1 Uses of Virtual Memory

Modern general-purpose processors and operating systems support virtual memory. Virtual memory (VM) was first motivated by the need to deal with the difficulties of *dynamic*

storage allocation for the multiprogramming environment. In addition to this main drive (i.e., transparent memory management), the use of virtual memory helps to significantly improve programmability in several ways [32].

- **Demand Paging and Swapping.** Virtual memory allows physical memory for data to be allocated when the data is accessed for the first time (i.e., *demand paging*), which helps to begin a process fast and to efficiently manage physical memory resources without allocating a large amount of memory that would be required by a program in advance. By supporting *swapping*, in addition, virtual memory provides an illusion of physical memory which is larger than the actual physical memory size installed. This significantly lessens the burden of programmers.

- **Protection.** Virtual memory also supports the *protection* mechanism of memory access. There are two types of protection: i) privileged level and ii) access permission. For the former, each process is in a particular privilege level at a given time (e.g., unprivileged user or privileged kernel mode). Depending on the current privilege level (CPL), acceptable operations of a process are determined. For example, in the user mode, access to particular data (e.g., kernel address space) and I/O port are denied, and the execution of specific privileged instructions is restricted. These operations are reserved only to the kernel mode.

  The privileged level basically explains the type of data that a process can access depending on its state. In contrast, *access permissions* define the type of acceptable operations (e.g., read, write, and executable) to particular data at the page-level. These protection mechanisms are helpful to avoid malicious or unintended operations.

- **Data Sharing.** Virtual memory also allows the same data (e.g., codes and data structures) to be shared by multiple processes by mapping different virtual addresses of the processes to the same physical address. This helps to efficiently manage memory capacity by avoiding data replication.

To take advantage of the benefits of VIVT caches in modern processors, all these features of virtual memory should be seamlessly supported. However, the straightforward use of VIVT caches can lead to several design issues, as discussed below.

2.2.2 Challenges of Supporting VM with VIVT Caches

Changes in Virtual-to-Physical Address Mapping. First, virtual-to-physical address mapping can be dynamically changed by the operating system during a program's execution. For instance, a physical page (A) for a virtual page (B) can be swapped out at some point, and the virtual page (B) later can be mapped to another physical page (C). In this case, when the virtual page mapping is unmapped, data in a virtual cache that is from the corresponding virtual page (B) should be identified and invalidated. Otherwise, *stale or wrong data* could be provided to a processor based on the previous mapping between the virtual page (B) and physical page (A).

Access Permission Checking. For conventional physical caches, permission checking is performed via a TLB on every cache access. However, with VIVT caches, the TLB is consulted only when cache misses occur, and thus, the location of checking (keeping) the page permission bits should be performed at different times and locations. Simply each cache line of a VIVT cache can keep track of the information, and the permission check can be performed when a matching cache line is found (i.e., tag hit). However, like the case of virtual-to-physical mapping changes, which is discussed above, the corresponding cache lines need to be identified and invalidated on page permission changes. Or the permission information of the cache lines should be updated. Otherwise, cache accesses may not respect the up-to-date permissions in the page table.

Virtual Address Synonyms. In addition, the *data sharing nature of virtual memory* enables different virtual addresses to be mapped to the same physical address (i.e., *virtual address synonyms*). As we will see later in the following chapter (Chapter 3), for a variety of reasons, synonymous accesses are present over the execution of an entire program. For

programs with frequent user-kernel interactions, synonym accesses frequently occur. Due to the synonyms, with VIVT caches, the same data can be placed and accessed later with different multiple virtual addresses, which could lead to several complications.

- **Data consistency:** The same data can be accessed with synonymous virtual addresses. Thus, it is possible that multiple copies of the same data reside in the cache with different virtual addresses (i.e., *data duplication*). This could reduce the cache capacity [107]. Second, some of the (duplicated) data could be modified. Thus, the cache could provide stale data to a CPU for synonymous accesses (i.e., *data consistency issue*) [109].

- **Cache coherence:** A coherence event typically uses physical addresses. To perform correct coherence operations for virtual caches, physical-to-virtual address translation is required. This adds latency and consumes energy. Hence an efficient way of performing the reverse translation is required.

- **Sequential semantics of a program:** With virtual caches, virtual-to-physical address translation is performed only for virtual cache misses. So, only virtual addresses will be used for conventional load-store queues. This can result in a *violation of the sequential semantics of a program* [88, 109]. That is, a load may not identify the matching latest store in the store queue due to synonyms. Thus, stale data could be returned from caches or from a matching older store.

- **Memory consistency:** In a similar vein, using virtual addresses for a load queue could potentially violate memory consistency models. When a coherence event, e.g., an invalidation or eviction, occurs in L1 caches, a load that has been carried out (speculatively) for the corresponding data may need to be identified and replayed [18, 39, 76, 93, 97]. In some commercial processors, these issues are handled by finding potentially offending loads by matching the *unique* physical addresses and replaying

them. With virtual caches, however, synonyms can complicate the identification operations.

- **Buffered instruction consistency issues:** To increase the bandwidth of the instruction fetch, some fetched (or decoded) instructions are buffered and reused in pipeline stages (e.g., trace cache, loop stream buffer (or loop cache), etc) [1, 64, 92]. Code modifications due to self-modifying codes could lead to another consistency issue for them.

  A processor core performs the modifications through store operations, and new instructions can reside in data caches. However, it is possible that stale instructions still reside in the buffers. Thus, we need to efficiently identify the corresponding, stale instructions. Like the memory consistency issue discussed above, the identification process could be complex due to virtual address synonyms. In practice, some commercial designs resolve this problem by employing a *unique* physical address for the identification. In addition, another consistency issue could occur between instruction and data caches when the old instructions are being cached in instruction caches.

Other considerations. In practice, depending upon the microarchitecture, additional challenges can arise. For example, to handle TLB misses, some microarchitectures, e.g., x86, support a hardware page-table walker [48, 50]. The page table is traversed by the page-table walker with physical addresses. However, page table entries (PTEs) may be accessed using a virtual address by the OS, and they may end up in the virtual caches. Thus, we need an efficient mechanism allowing the page table walker to access virtual caches to find the corresponding PTE. Additionally, supporting multiprogramming could lead to *homonym* issues because of the same virtual address mapped to the different physical addresses.

As we discussed in the previous section, the widespread practical deployment of virtual caches has been thwarted for several decades. The key reason is the complications resulting from virtual address synonyms. Actually, several virtual cache designs have been proposed

[13, 41, 57, 59, 88, 104, 106, 110]. However, they fall short of effectively obviating the complications (refer to Section 6.1). Some approaches require complex data management (e.g., data duplications/relocations). Other proposals require OS involvement, which could restrict programmability. Furthermore, most of the prior literature did not consider several issues that are critical to a viable commercial implementation of any virtual cache, e.g., data/memory consistency issues, the violation of sequential semantics of a program, the consistency issue of buffered instructions, etc. Hence, we need *a software-agnostic, pure hardware solution that achieves the benefits of virtual caches in a practical manner*.

Chapter 3

EMPIRICAL CHARACTERISTICS OF VIRTUAL ADDRESS SYNONYMS

A variety of common programming and system practices lead to synonyms [13, 19, 88]. In this chapter, first we will discuss sources of synonyms. Then, several empirical observations about temporal characteristics of synonyms are made, which motivates a practical design of the VIVT caches that we propose in this thesis.

3.1 Sources of Virtual Address Synonyms

A page (e.g., 4KB size) is a basic unit of virtual memory implementation, and thus, synonyms are observed at a page-level granularity. Figure 3.1 exemplifies page mappings of synonyms between two processes. Each process has its own (user) virtual address space (i.e., green and yellow boxes), while the kernel virtual address space is shared by all processes (i.e., blue boxes). Depending on programs' operations, synonym accesses can occur through different virtual address spaces in many situations, as follows:

1. **Activities in user spaces.** Synonyms are commonly used to share data (e.g., shared libraries/codes and memory mapped files) among multiple processes (see Physical page 1 in Figure 3.1). Also, *copy-on-write* (COW) enables different processes to share the same physical page until data in the page is actually modified by one of the processes. In a similar vein, *kernel same page merging* allows multiple processes to share pages with the same data (e.g., a large disk image of virtual machines) [3], while these help to consume less memory as well as to reduce the latency of the memory allocation. In addition, a file can be mapped with different virtual addresses.

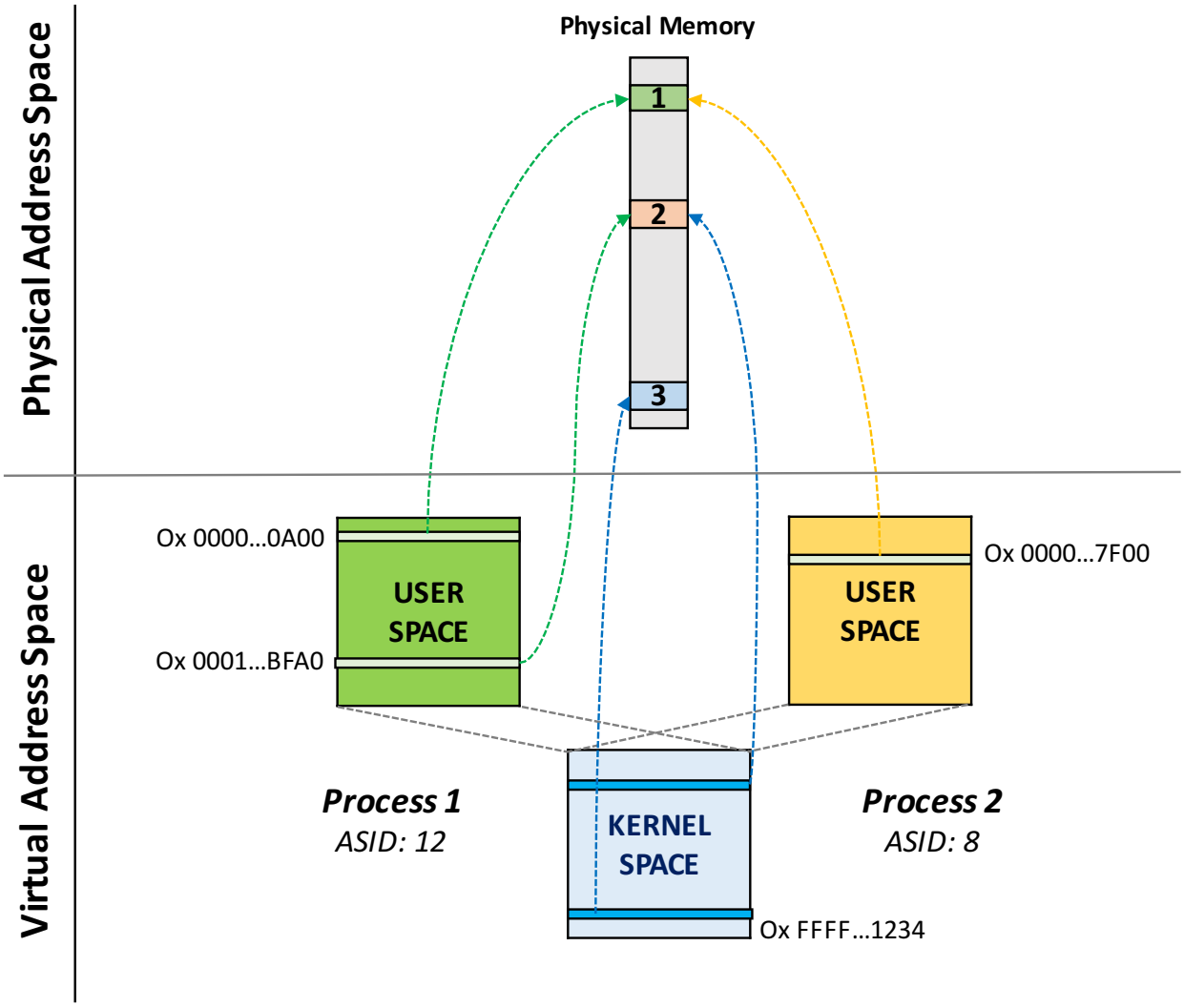


Figure 3.1: Possible cases for virtual to physical page mappings of synonyms for two processes

2. **User-kernel interactions.** Basically, a user program has restricted permission to the system. Some privileged operations are performed by the kernel of the operating system. For example, on a write to a COW page with read-only permission in the user virtual address space, the kernel allocates a new physical page and initializes it based on the old shared copy via a (temporal) virtual address mapping. Then, the actual write that causes the COW operation is performed through the original user virtual address space. In addition, for some OSes, before a physical page is allocated for a

(user) virtual page of a process for the first time, the physical page is initialized with zeros through the kernel virtual address space to avoid security problems.[1] Later, data in the page is accessed with a virtual address of a user space.

Regarding I/O requests, in addition, the kernel first keeps read (or written) data in a buffer (e.g., a page cache) in memory. The buffer is managed with kernel virtual addresses and the data in the buffer can be remapped to user space when it is referenced [88] (see Physical page 2 in Figure 3.1). Plus, some programs (e.g., databases) request direct I/O access by bypassing the page cache of the operating system. In this case, the kernel directly accesses user data through kernel virtual address space [13]. As we can easily notice, the operations due to user-kernel interactions cause read-write synonyms, resulting in many issues.

3. **Shared kernel space.** For some OSes (e.g., x86-Linux), the kernel address space is globally shared by all distinct processes for privileged operations, and the interfaces and data (e.g., network communication (TCP/IP), memory operations, locks, I/O, etc.) are accessed through the kernel space. When we suppose that virtual addresses including an address space identifier (ASID) are employed to access virtual caches, accesses to the kernel space lead to synonym issues due to differences in the ASID.[2] Figure 3.1 shows that the physical page 3 is accessed through the kernel space. But each process will use virtual addresses with different ASIDs, i.e., 12 and 8 for process 1 and 2, respectively. In this case, the ASID is used as a part of a cache tag, and thus the straightforward use of these virtual addresses for the globally shared address space could cause data duplication in the virtual cache.

[1] For the first read operation to a virtual page, a corresponding page table entry will simply point to a physical page initialized with zeros, i.e., zero page, instead of actually allocating a new physical page with zeros; this zero page is shared by many virtual pages. This is why we see zero values from uninitialized variables, in general.

[2] The ASID (or a process ID) can be a part of a tag. This approach is a common way of addressing homonym issues without flushing virtual caches on context switches [13, 109].

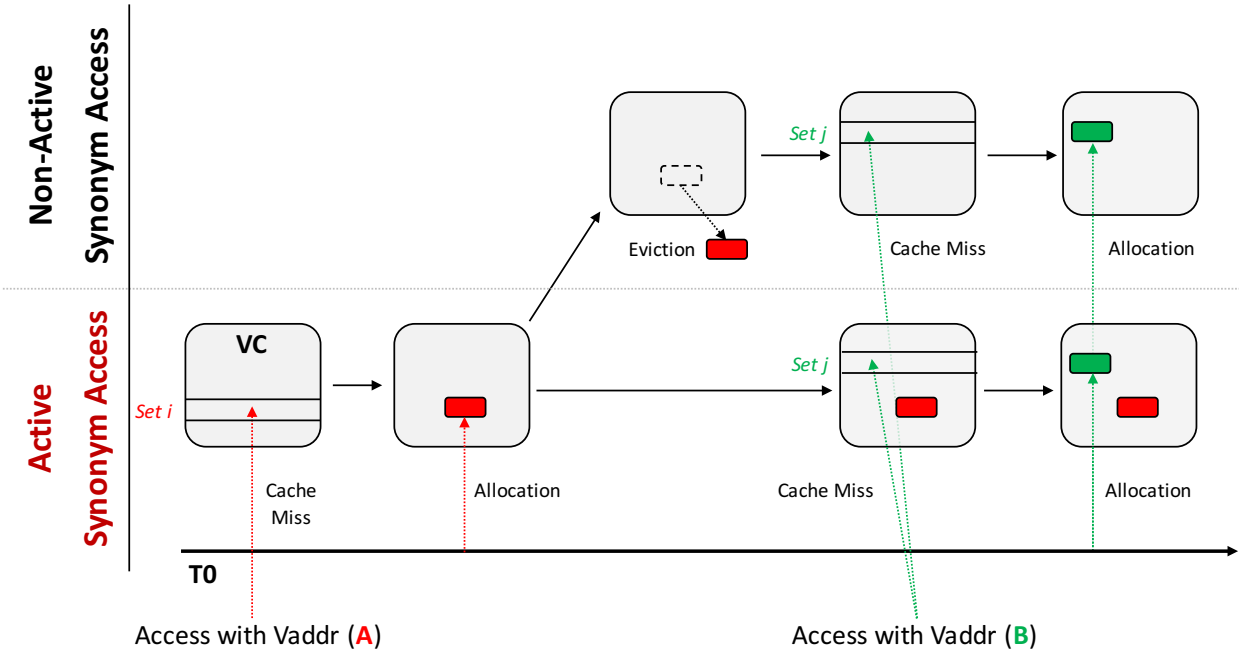


Figure 3.2: Difference between active and non-active synonym accesses in a cache

3.2 Active Synonym Accesses

As discussed above, synonym accesses are commonly present over the duration of a program. Actually, however, synonym accesses *do not always* cause issues in a virtual cache. Synonyms in a virtual cache are an actual problem only if a block is cached and accessed with different addresses *during its residence in the cache* (i.e., there is an **active synonym access**). Accordingly, *the temporal characteristics of potential synonym accesses in a cache are what is important for virtual caching*.

Figure 3.2 presents differences between an active synonym access and non-active synonym access in a VIVT cache. Importantly, both accesses are synonym accesses. For a non-active synonym access, data cached with a virtual address (A) is evicted before the following access to the same data with a different virtual address (B). Thus, the following request does not cause any issues for managing the VIVT cache, although it is a synonym access. Hence, the interest is not just synonymous accesses, but the access pattern of active

synonym accesses, especially for their temporal behavior in a cache.

3.3 Temporal (Active) Synonym Behavior in Caches

This section presents five empirical observations of temporal synonym behavior, based on real world server, mobile, and conventional workloads. As we shall establish below, the temporal characteristics of active synonyms do indeed display properties that are amenable to designing a **software-transparent, practical, L1 virtual cache** that we propose in the next chapter.

To gather the empirical data, we use several real world applications running on a full-system simulator, as we describe in Section 4.3.1. Several of the data items below are gathered using periodic sampling of the contents of a cache. Since in some cases the samples with different cache organizations may correspond to different points in a program's execution, comparing the absolute numbers of two arbitrary data points may not be very meaningful; the trends characterizing the temporal behavior are what we want to highlight. When absolute numbers are appropriate, they are presented without sampling.

Before proceeding further, we define the main terms used in this section to describe the observed characteristics. Let us call the set of one physical page (P_X) and possibly multiple virtual pages associated with P_X the *Equivalent Page Set* (EPS_X), and let C(X) be the *cardinality* of EPS_X, i.e., the number of virtual pages in EPS_X over the entire duration of a program's execution. Temporal characteristics of synonyms in a cache are our interest, thus, let TC(X,T) be the *temporal cardinality* of EPS_X, i.e., the number of virtual pages that are synonymous with physical page P_X in time interval T. Then , an *active synonym* for P_X (or the associated EPS_X) occurs when TC(X,T) $\geqslant 2$ in the time interval T. The time interval T that is of interest is where one or more lines from the page P_X are resident in the cache.

	(1) Avg. number of physical pages with active synonyms								(2) Avg. number of virtual pages in an EPS for active synonyms							
	Inst. Cache (KB)				Data Cache (KB)				Inst. Cache (KB)				Data Cache (KB)			
Size (KB)	**32**	**64**	**128**	**256**	**32**	**64**	**128**	**256**	**32**	**64**	**128**	**256**	**32**	**64**	**128**	**256**
TPC-H	6	20	57	133	2	5	8	12	2	3	8	32	2	2	2	2
SPECjbb2005	0	0	2	22	0	1	2	2	-	-	2	2	-	-	2	2
Memcached	36	87	158	282	52	93	160	284	2	3	12	28	2	2	2	2
bzip2	1	12	93	282	0	0	0	0	-	13	23	29	-	-	-	-
h264ref	0	0	5	115	0	0	0	1	-	-	25	24	-	-	-	-
Stream	0	1	23	106	1	1	2	4	-	-	4	4	-	-	2	2
Raytrace	0	0	0	0	7	15	29	57	-	-	-	-	2	2	2	2

Table 3.1: Analysis of EPSs with active synonyms in various sizes of caches

3.3.1 Number of Pages with Active Synonyms

The first set of data (1) of Table 3.1 presents the average number of physical pages with active synonyms. The time interval T is the time in which data blocks from the corresponding P_X reside in the cache of a particular size and thus it varies with cache size. Both instruction and data caches of varying sizes are considered. For smaller cache sizes there are few pages with active synonyms; the average number of pages with the potential for synonymous access increases as the cache size increases.

This phenomenon occurs due to two complementary reasons: smaller caches not only contain data from fewer pages, but all the blocks from a given page are likely to be evicted from a smaller cache earlier than they would be from a larger cache. Figure 3.3 substantiates this. The figure presents the lifetime of physical pages in a cache for `memcached`; other workloads also show similar trends. Figure 3.3 has two CDFs of the aspect for a 32KB cache and for a much larger (2MB) cache, respectively. We can see that there is a much shorter lifetime for physical pages in a smaller cache (i.e., earlier evictions) due to its limited resources.

We notice that, in Table 3.1, a large 256KB cache contains data from tens or hundreds of pages with the potential for synonyms; the number for all of the memory (not shown) is even larger. The latter confirms that there are many EPSs with $C(X) \geqslant 2$, i.e., the potential

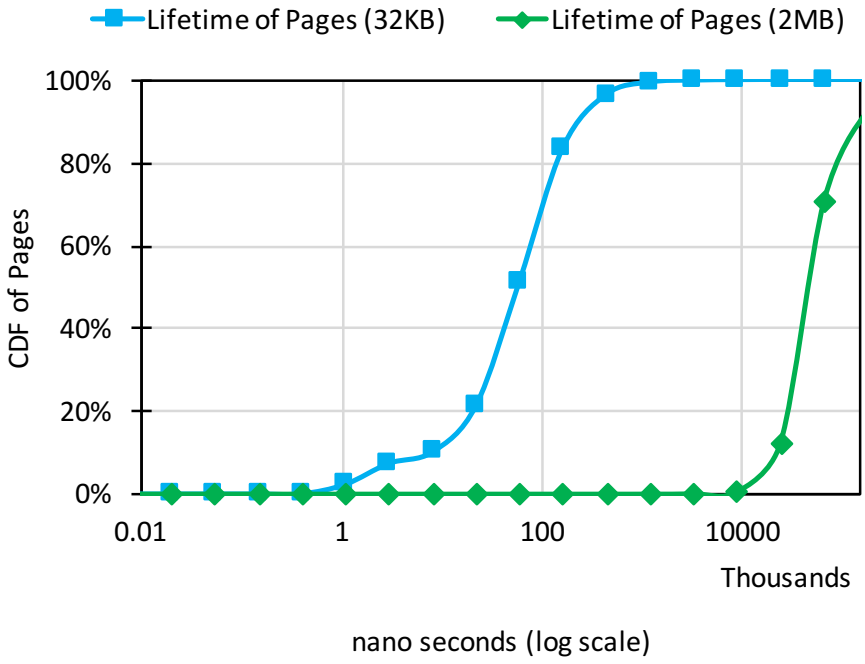


Figure 3.3: Analysis of the lifetime of 4KB pages in data caches. The lifetime of a page is defined as the period between when data from a physical page is cached for the first time and when data of the page is last accessed in the cache before it is evicted.

for synonyms is real. However, for small cache sizes, such as those one might expect to use for an L1 cache (e.g., 32-64KB), the number of pages residing in the cache with TC(X,T) $\geqslant 2$ is small.

Observation 1 (OB_1**):** The number of pages with active synonyms is quite small for small cache sizes that are typical of an L1 cache.

3.3.2 Temporal Cardinality of an EPS

The next set of data (2) of Table 3.1 presents the average number of distinct virtual pages in an EPS for which an active synonym occurs, for different sized caches.[3] This number increases with cache size, due to the longer residence time of a block (page); it can become quite large for the larger instruction cache sizes due to shared libraries and kernel interfaces.

[3]A blank entry indicates that the corresponding number is not meaningful since there are almost zero pages with active synonyms resident in the cache.

	Instruction Cache		Data Cache	
Size (KB)	**32**	**64**	**32**	**64**
TPC-H	63	57	73	74
SPECjbb2005	69	67	63	56
Memcached	4	0	5	5
bzip2	70	71	98	98
h264ref	79	75	82	82
Stream	76	70	70	70
Raytrace	100	100	26	26

Table 3.2: Analysis of frequency of changes in the LVA (%)

However:

Observation 2 (OB_2): The average number of virtual pages mapped to the same physical page, for pages with an active synonym, in a small cache is quite small.

3.3.3 Changes in Leading Virtual Address (LVA)

With a virtual cache, let's say that data is cached and looked up with a *unique leading virtual address* (page number) that is one of active synonyms (refer to Figure 3.2), while data from the corresponding physical page resides in the virtual cache.[4] Suppose at some point in time address V_i was the first virtual page in an EPS_X and was therefore being used as the leading virtual address for the corresponding physical page P_X. Now suppose that after being cached with V_i, other references were made, and all the blocks from P_X were evicted from the cache. The next time P_X was referenced, virtual address V_j was the leading virtual page for that EPS. We say that a change in the leading virtual address occurs if $V_j \neq V_i$.

Table 3.2 presents the percentage of changes in the leading virtual address (LVA). Only pages in which active synonym accesses occur at least once are considered. An entry indicates the percentage of time a change in the LVA occurs; 100 indicates that the LVA always changes and 0 means that it is always the same. The data indicates that it is quite common

[4]The leading virtual address (page number) at a given time can change in different phases of program execution.

		Set (1)			Set (2)		
	Size (KB)	**32**	**64**	**128**	**32**	**64**	**128**
Inst. Cache Access	**TPC-H**	45	105	161	15	22	55
	SPECjbb2005	0	0.1	0.2	0	0.1	0.2
	Memcached	502	695	770	282	352	492
	bzip2	0.3	2.3	2.9	0.3	2.3	2.9
	h264ref	0	0.1	0.3	0	0.1	0.3
	Stream	0.3	3	28	0.3	0.3	26
	Raytrace	0.1	0.1	0.1	0.1	0.1	0.1
Data Cache Access	**TPC-H**	48	80	87	9	12	27
	SPECjbb2005	2.7	22.8	25.5	2.6	22.7	25.1
	Memcached	478	518	530	274	294	295
	bzip2	1	1.9	2.3	1	1.9	2.3
	h264ref	0.1	0.7	2	0.1	0.7	2
	Stream	20	24	25	20	24	25
	Raytrace	32	93	119	32	93	118

Table 3.3: Number of references (per 1000) to pages with active synonyms in caches

for different virtual addresses from the same EPS to be the leading virtual addresses at different times during the execution of the program. For *memcached*, the percentage of LVA changes is small. This is due to heavy synonym access activity, which causeslines from synonym pages to be frequently referenced. Thus they are not replaced, and continue to reside in the cache with the leading virtual address, even though additional references may be made with other virtual addresses. This results in fewer LVA changes.

Observation 3 (OB_3**):** When multiple virtual addresses map to the same physical address, always using the same virtual address to cache the page can be unnecessarily constraining.

3.3.4 Accesses to Pages with Active Synonyms

Table 3.3 presents the frequency of cache access to physical pages with active synonyms for different sized instruction and data caches. There are two sets of data for both instruction and data caches; each entry is the number of references per 1000 accesses.

The first set of data (1) is the number of references to cache blocks contained in pages

with active synonyms, using any virtual address in an EPS. It is for these references that a virtual cache design may have to take additional steps to ensure correct operation. The second set (2) is the number of references made with a non-leading virtual address V_j, that is different from the current leading virtual address V_i in an EPS. These are the references for which a virtual cache design, that we describe in Section 4.2, will have to intervene to remap V_j to V_i and submit the access with V_i instead of V_j.

The first set of data suggests that the overall percentage of accesses to cache blocks in pages with active synonyms is quite small in most cases, especially for smaller cache sizes. However, in some cases, it is quite large (e.g., `memcached` and `tpc-h`). The number of references to blocks in pages with active synonyms increases as the cache size increases, but is still somewhat small in most (although not all) cases. At first glance, we would expect the percentage of accesses to synonym pages to be the same regardless of cache size. This is true. However, the percentage of accesses to **active synonyms** is smaller, as there are fewer active synonyms in smaller caches.

Looking at the second set of data in Table 3.3 and comparing an entry with the equivalent entry in the first set, notice that the entries in the second set are smaller (in some cases by a large amount) than the entries in the first set. This suggests that, of the small number of references to pages with an active synonym (1st set), even smaller number of references (2nd set) are made with a virtual address that is different from the leading virtual address (i.e., non-leading virtual address).

Observation 4 (OB_4**):** Typically only a small fraction of cache accesses are to cache lines from pages with active synonyms.

Observation 5 (OB_5**):** In most cases, a very small percentage of cache accesses are to cache lines from pages with active synonyms that are cached with a different virtual address.

According to the sources of synonyms, Figure 3.4 classifies the active synonym accesses with non-leading virtual addresses in a 32KB data cache. As discussed above, for real

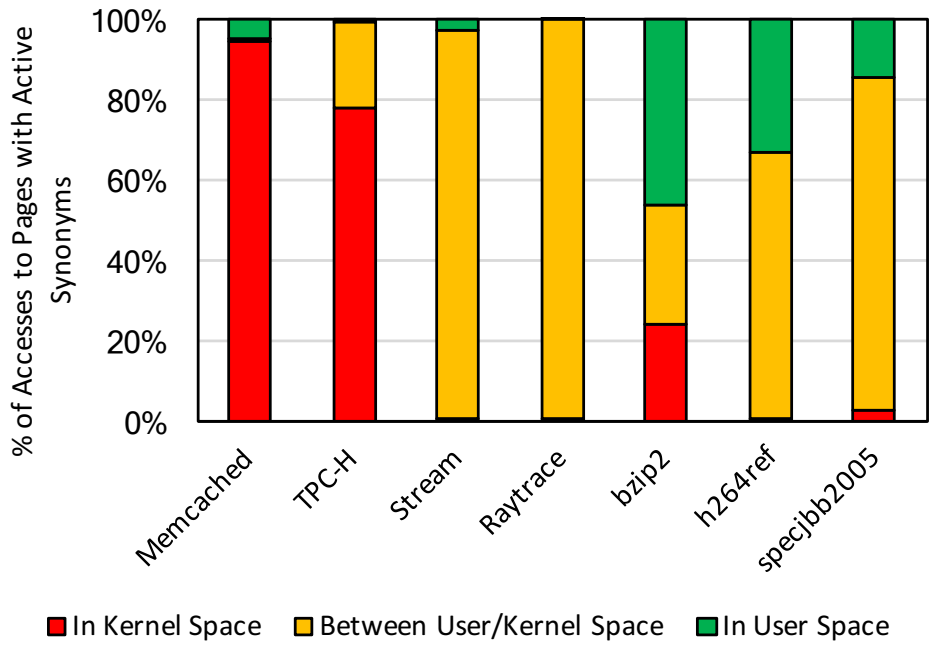


Figure 3.4: Classification of active synonym accesses. The Y-axis indicates a percentage of accesses to pages with active synonyms in a 32KB data cache. There are three sub-bars; each bar is associated with one of the categories of sources of synonym accesses discussed in the previous section.

server workloads such as `memcached` and `tpc-h`, most of the accesses result from activities in kernel space or user-kernel interactions. Notice that active synonym accesses of mobile workloads such as `stream` and `raytrace` are mostly due to user-kernel interactions (e.g., memory allocation and initialization with copy-on-write).[5] *The results suggest that major synonym issues are related to kernel operations.* Thus, for systems not executing the OS kernel codes (e.g., accelerators like GPUs), we can expect less likelihood of (active) synonym accesses, which motivates our GPU virtual cache proposal discussed in Chapter 5.

[5]Here, we do not discuss the characteristics of the other workloads (bzip2, h264ref, and specjbb2005) because they show extremely rare active synonym accesses in the small (32KB) cache, i.e., less than 0.2% of total data cache accesses.

3.4 Chapter Summary

In this chapter, we described sources of virtual address synonyms by exemplifying a variety of common programming and system practices in modern systems. Then, we classified them according to interactions between user and kernel operations. We defined the concept of an *active synonym*, and brought out the importance of *temporal properties of (active) synonyms* in caches for designing virtual caches since synonym accesses cause a real problem only if a block is cached and accessed with different addresses, while the data resides in the cache.

Regarding the temporal characteristics of potential synonyms in caches, we made five empirical observations as follows:

1. The number of pages with active synonyms is quite small for small cache sizes like an L1 cache (e.g., 4).

2. The average number of virtual pages mapped to the same physical page, for pages with an active synonym, in a small cache, is quite small (e.g., 2).

3. When multiple virtual addresses map to the same physical address, always using the same virtual address to cache the page can be unnecessarily constraining.

4. A small (but meaningful) fraction of cache accesses (e.g., 3%) is to cache lines from pages with active synonyms in a small cache.

5. A very small percentage of cache accesses (e.g., 1%) are to cache lines from pages with active synonyms that are cached with a different virtual address.

As we shall see in the next chapter, by leveraging these empirical observations, we propose and evaluate a practical L1 VIVT cache design for modern CPUs, which reaps most of the benefits of virtual caching.

Chapter 4

REDUCING TLB LOOKUP OVERHEAD OF CPUS WITH VIRTUAL L1 CACHING

4.1 Introduction

In this chapter, we propose a practical L1 VIVT cache design that is software transparent, and efficiently manages virtual cache complications (e.g., synonyms). By leveraging the temporal characteristics of synonyms discussed in the previous chapter, we substantiate why our proposal is a compelling design and then evaluate the effectiveness of the proposed virtual cache design as a *TLB lookup filter*.

We empirically observe five temporal characteristics of synonyms:

- **Observation 1:** The number of pages with active synonyms is quite small for small cache sizes that are typical of an L1 cache.
- **Observation 2:** The average number of virtual pages mapped to the same physical page, for pages with an active synonym, in a small cache, is quite small.
- **Observation 3:** When multiple virtual addresses map to the same physical address, always using the same virtual address to cache the page can be unnecessarily constraining.
- **Observation 4:** Typically only a small fraction of cache accesses are to cache lines from pages with active synonyms.
- **Observation 5:** In most cases, a very small percentage of cache accesses are to cache lines from pages with active synonyms that are cached with a different virtual

address.

They suggest that it might be practical to design an L1 VIVT cache in which data is cached with one *unique leading virtual address*, V_i, and synonym accesses made to the same data with a different address, V_j, remapped to use V_i instead. In particular:

- **Observations 1-2** suggest that a structure tracking remapping links [V_j, V_i] can be quite small.
- **Observation 3** suggests that this data structure needs to track the mappings dynamically, since it is desirable that the leading virtual address V_i changes during a program's execution.
- **Observations 4-5** suggest that the (re)mapping data structure may be infrequently accessed.

Using these observations as a basis, we propose a practical virtual L1 cache design, called a *Virtual Cache with Dynamic Synonym Remapping* (VC-DSR). At a very high level, its operation is as follows. Data is cached and accessed with a *(dynamic) unique leading virtual address* for a given physical page. When a virtual address is generated, hardware structures are consulted to see if the address is a leading virtual address. If so, the address is used to look up the virtual cache. Otherwise, the corresponding leading address is identified and used to access the virtual cache, as shown in Figure 4.1. That is, dynamic remappings from a non-leading (virtual) address to a leading address are needed for *select accesses*, rather than performing virtual to physical translation, via a TLB, for every access. Thus, *data duplication and relocation are not needed in the cache*.

Such remapping operations are rare because synonyms are short-lived, in smaller caches. Thus, VC-DSR can handle needed remappings in an energy and latency efficient manner, thereby achieving most of the benefits of VIVT caches without any modifications to software. In addition, the use of a unique virtual address for all the operations of a

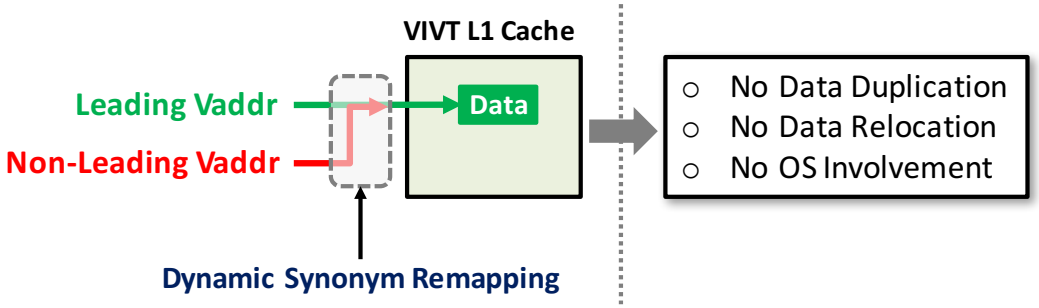


Figure 4.1: Overview of Dynamic Synonym Remapping

virtual cache greatly simplifies the design of our proposal, as well as the overall memory hierarchy.

The main contributions of this chapter are:

- We propose VC-DSR, a practical solution to a problem that has long vexed computer architects: achieving most of the benefits of virtual caches but without *any* software involvement.
- Regarding specific microarchitectural details that are critical to viable commercial implementation of any virtual cache (Section 4.2.5 and 4.2.6), we discuss issues brought about by synonyms and solutions for VC-DSR.
- Presented experimental results show that VC-DSR saves about 92% of the dynamic energy consumption for TLB lookups and also achieves most of the latency benefit (about 99.4%) of ideal (but impractical) virtual caches (Section 4.3).

4.2 Design and Implementation of VC-DSR

4.2.1 Design Overview of VC-DSR

Figure 4.2 gives an overview of the overall microarchitecture: the processor generates virtual addresses (Phase ①), the L1 cache is virtually indexed and tagged (Phase ③), and the lower-level caches are traditional physical caches (Phase ⑤). Phase ④ is a boundary

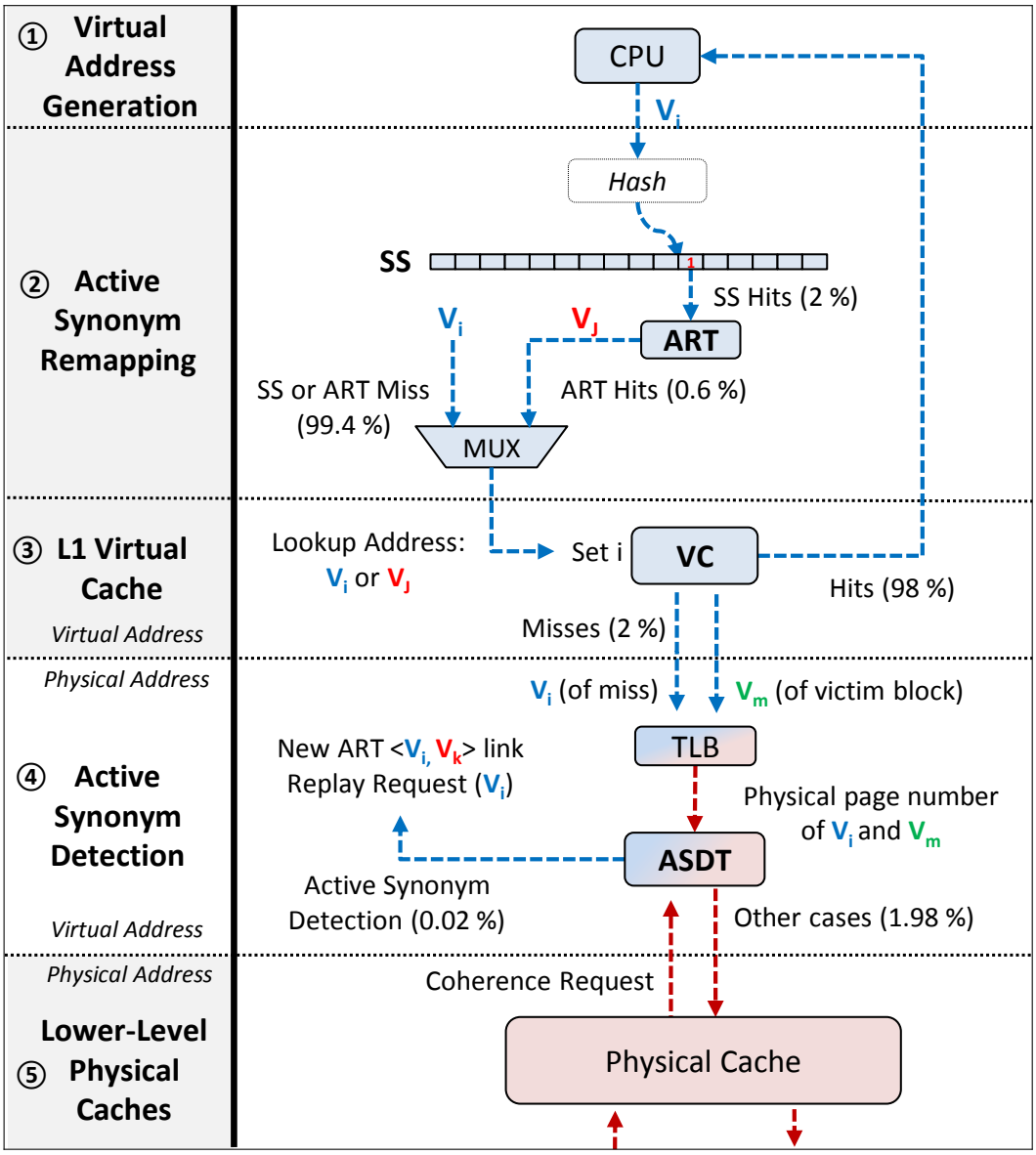


Figure 4.2: Schematic Overview of VC-DSR

between a virtual and a physical address, and thus the virtual to physical (or physical to virtual) address translation is performed via a traditional TLB for L1 virtual cache misses (or via an Active Synonym Detection Table for coherence requests from lower-level caches as we discuss later). In addition, there are other microarchitectural structures for the active synonym remapping (Phase ②) and detection (Phase ④), ensuring the correctness of overall cache operations.

Basic Operations: VC-DSR uses a unique leading virtual address for a given physical

page not only to place data of the corresponding page but also to later access the data in an L1 virtual cache. Thus, conceptually, an **Address Remapping Table** (ART) is consulted on every cache access with a virtual address (V_i)[1] generated by a CPU. It identifies if V_i is a non-leading virtual address for a given physical page. If so, the corresponding leading virtual address, V_j, is provided, and V_i is remapped to V_j for that access. Otherwise, V_i is used to look up the virtual cache.

Since temporally there are expected to be few accesses to pages with active synonyms (OB_4), the chances for finding a matching entry in the ART are low, e.g., 0.6%, and thus most accesses to the ART are wasted. To reduce the number of such ART accesses, a *Synonym Signature* (SS) could be used. The SS is a hashed bit vector based on the virtual address (V_i) generated by the CPU and conservatively tracks the possibility of a match in the ART. If the corresponding bit in the SS indicates no possibility, e.g., 98% of the time, the ART is not consulted and V_i is used to look up the cache. Otherwise, e.g., 2% of the time, the ART is consulted to determine the correct lookup address (V_i or V_j).

On a cache miss, the virtual address (V_i) is used to access the TLB, and the corresponding physical page address, P_i, is obtained. Next, an **Active Synonym Detection Table** (ASDT) is searched for an entry corresponding to P_i. A valid entry indicates that some data from that physical page resides in the virtual cache. If there is no match, one is created for the [P_i,V_i] pair, and V_i will be used as the leading virtual address. If there is a match, the corresponding leading virtual address, V_k, is obtained. If $V_i \neq V_k$, an active synonym is detected, an entry is created in the ART for the [V_i,V_k] tuple and the corresponding bit set in the SS. V_k is then used as the lookup address for references made with V_i, while the [V_i,V_k] tuple is valid in the ART.

[1] All virtual addresses include an ASID to address the issue of homonyms.

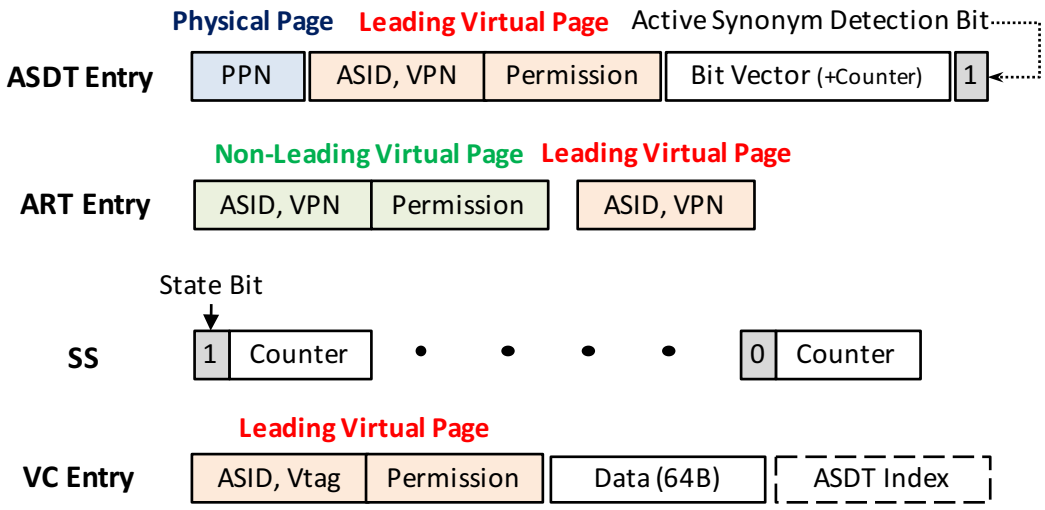


Figure 4.3: Overview of Entries for Structures Supporting VC-DSR

4.2.2 Details of Structures Supporting VC-DSR

More details of components supporting VC-DSR in each phase are described next. Figure 4.3 presents the basic organization of an entry for each component.

4.2.2.1 Active Synonym Detection

The Active Synonym Detection Table (ASDT) is a set-associative array indexed with a physical page number (PPN). A valid entry in the ASDT tracks (1) whether lines from a physical page are being cached in the virtual cache and (2) the leading virtual address being used to cache them, along with its permission bits. For the former, a counter suffices for correctness. However, for more efficient evictions (Section 4.2.3.5), employing a bit-vector to identify the individual lines from the page in the cache may be a better option.

The ASDT is consulted on every L1 cache miss to see if an active synonym access occurs by comparing the current leading virtual address and the referenced virtual address, and an active synonym bit in the entry is set if there is one. In addition, the ASDT is also consulted to perform the physical to the (leading) virtual address translation for coherence requests from lower-level caches.

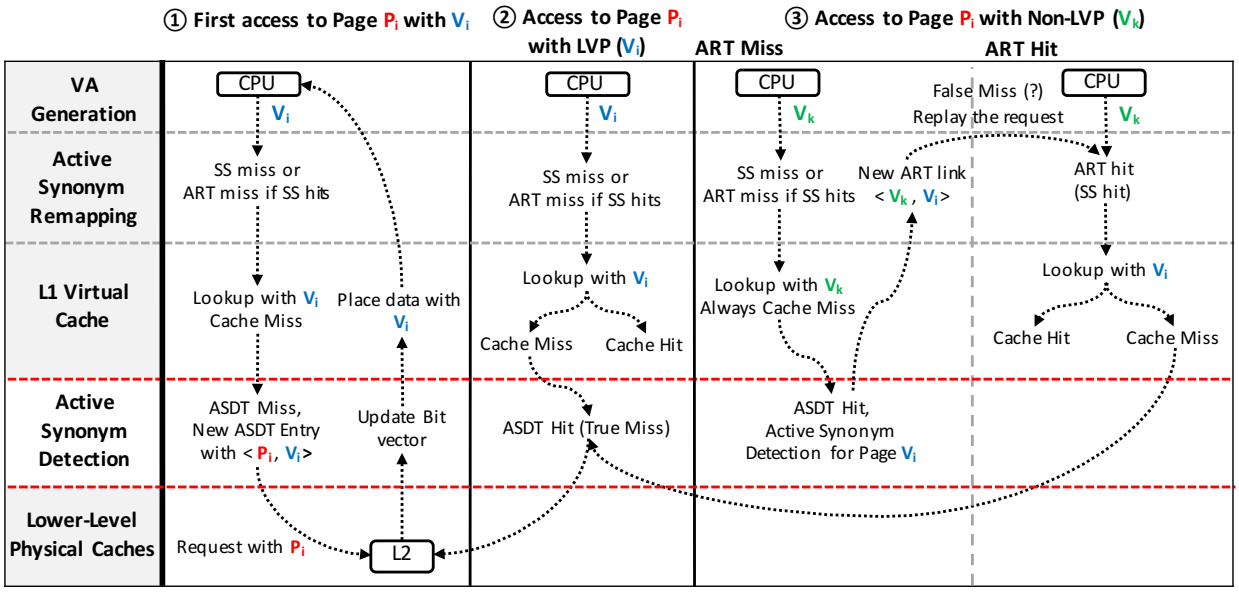


Figure 4.4: Overall Operations of VC-DSR

4.2.2.2 Active Synonym Remapping

The design employs two other structures to efficiently perform a remapping between a non-leading and the corresponding leading virtual address, when needed.

The Address Remapping Table (ART) is a small set associative cache, indexed with a virtual address generated by a CPU, whose entries are created when the ASDT detects an active synonym. A valid entry in the ART tracks a [non-leading, leading virtual page] tuple, along with the permission bits of the non-leading page, for an active synonym. On a match, the ART returns the leading virtual page and the permission bits for the requested (non-leading) virtual address. The permission check for accesses with a non-leading virtual address is performed at this point (details in Section 4.2.5) and the leading virtual address is used to look up the virtual cache for the request. The absence of a matching entry in the ART indicates the lack of an active synonym for that page, i.e., a virtual address generated by the CPU is the correct lookup address. When an active synonym no longer persists, e.g., when all the cache lines from a physical page are evicted from the virtual cache, the corresponding entry in the ART has to be invalidated (details in Section 4.2.3.5).

Accesses to the ART are unnecessary when the referenced virtual address is a leading virtual address, i.e., no match in the ART. A Synonym Signature (SS) is used to elide most unneeded ART lookups. The SS is a Bloom filter [16], which is accessed with the virtual address of a request, and a **single bit** is read to determine if the ART should be accessed. When a new ART entry is populated, a bit in the SS is set based on the hash of the non-leading virtual address for the entry. Since multiple non-leading virtual addresses can be mapped to the same bit, each bit in the SS conservatively tracks the possibility of a match in an ART, and thus false positives are possible.

To prevent the SS from being overly conservative, a counter per bit tracks how many entries in the ART are associated with it; the counter increases/decreases when the corresponding ART entry with the virtual address is populated/invalidated. The size of the counter is proportional to the number of entries in an ART. It will be quite small (e.g., 4 bits) since an ART has few (e.g., 16) entries (see OB_1 and OB_2). As we shall see in Section 4.3.2, a small SS (e.g., 256 bits) with 5-bit counters is sufficient to filter out almost all unneeded ART lookups.

4.2.2.3 L1 Virtual Cache

An L1 virtual cache entry is effectively the same as a traditional virtual cache entry, as depicted in Figure 4.3. ASIDs are employed to address *homonym* issues without flushing the cache on context switches. The leading virtual page's permission bits are stored with each entry. Each entry also keeps track of the index of the corresponding ASDT entry to avoid an additional TLB lookup on a VC eviction. The detailed operations are discussed in Section 4.2.3.5.

4.2.3 Overall Operation

We describe the overall operation of VC-DSR. Figure 4.4 illustrates how virtual addresses are used and the needed operations in each phase.

4.2.3.1 Determining a leading virtual page (LVP)

An access is made with the virtual page number (V_i). Let us assume this is the first access to data from a physical page (P_i) (Case ①). A leading virtual page (LVP) has not been determined for P_i (i.e., there is no matching ASDT entry), and thus we have an SS miss or an ART miss (if an SS hit occurs due to the false positive information). Either way, V_i is used to look up the virtual cache (VC), leading to a VC miss.

The TLB is accessed with V_i to obtain the physical page number (PPN), and then the ASDT is checked for a matching entry of the PPN. As this is the first access to P_i, no matching entry is found. An ASDT entry is chosen as a victim (e.g., an invalid entry, or a valid entry with the lowest counter value). For a valid entry victim, operations needed to invalidate the entry are carried out (Section 4.2.3.5). The entry is allocated for the requested physical page P_i. The leading virtual page field is populated with the V_i and with its permission bits.

The line is fetched from the lower-level(s).[2] The corresponding bit in the bit vector of the ASDT entry is set and the counter is incremented. Then the fetched line (and the permission bits) is placed in the VC with the leading virtual address (V_i). A victim line in the VC may need to be evicted to make place for the new line.

4.2.3.2 Accesses with a leading virtual page

Further accesses to the physical page (P_i) with the leading virtual page (V_i) (Case ②) proceed without any intervention. A VC hit proceeds as normal. A VC miss indicates that the line does not reside in the VC (i.e., it is a true miss) since V_i is the leading virtual page. Thus, the line is fetched from the lower-level caches, and then the request is satisfied as discussed above.

[2]MSHR entries keep the index of the relevant ASDT entry. Hence no additional ASDT lookups are needed when the response arrives.

4.2.3.3 Accesses with a non-leading virtual page

We now consider the first synonymous access (left side of Case ③). Data of the common physical page (P_i) is accessed with a virtual address synonym (V_k). An active synonym has not yet been detected for this page. Thus, V_k is used to look up the virtual cache. Since V_k is not a leading virtual page (LVP), the access will result in a miss. The TLB is then accessed to translate V_k (to P_i), a matching entry for the common physical page (P_i) is found in the ASDT, a new active synonym is detected, and the ART is informed to create a new entry for the [V_k, V_i] tuple (and the SS is modified accordingly). Any further accesses to virtual page V_k will be remapped with V_i via the ART (right side of Case ③). Since the data may actually reside in the VC with the leading virtual address (i.e., it is a false miss), the request is resubmitted (replayed) to the VC with V_i. In this case, the VC hit/miss is handled like accesses with a leading virtual page.

4.2.3.4 Coherence Request

A coherence event typically uses physical addresses. Thus, it is handled as normal for lower-level (PIPT) caches. For coherence activities between L2 and virtual L1 caches, the ASDT has to be consulted to translate a physical address to the corresponding leading virtual address. This may add additional latency and energy overhead. However, such coherence events between an L2 and a (relatively small) L1 are rare [13]. The ASDT can also be used to shield the VC from unnecessary VC lookups if an ASDT entry includes information identifying individual lines from the page present in the cache (the bit vector).

4.2.3.5 Entry Eviction of VC-DSR Components

Virtual cache line eviction: On evicting a VC entry, the corresponding ASDT needs to be updated. A matching entry is always found in the ASDT since at least one line (i.e., the victim line) from the page was still resident in the VC. The corresponding bit in the bit

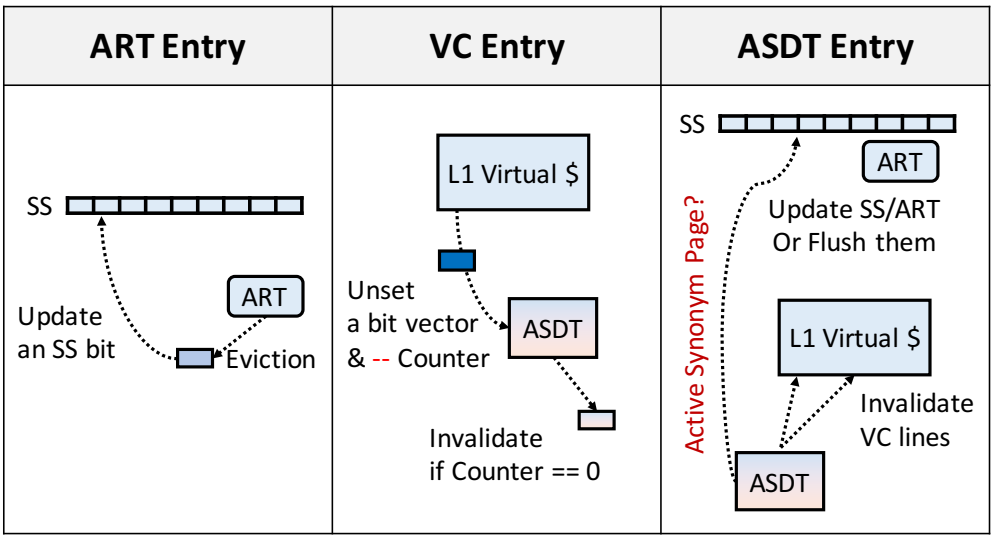


Figure 4.5: Required operations of entry evictions for VC-DSR components

vector is unset, and the counter decremented. If the counter becomes zero, the ASDT entry can be invalidated (see Figure 4.5).

To find the matching ASDT entry, the PPN is needed, via a TLB lookup. Thus, on a VC miss, with the straightforward approach, the ASDT and TLB are consulted twice, once each for the evicted line and the newly fetched line. An alternative is to keep the index of the relevant ASDT entry with each line in the VC (e.g., with 8 bits) (see Figure 4.3) so that the corresponding ASDT entry for the victim line can be directly accessed without a TLB (and ASDT) lookup.

ASDT entry invalidation/eviction: A valid ASDT entry is normally evicted when there is no available entry for a new physical page. Page information changes or cache flushing also triggers invalidations of corresponding ASDT entries. To evict an ASDT entry, it first has to be invalidated (Case ⑤). All lines from the corresponding (victim) page that are still in the VC have to be evicted. Furthermore, if active synonyms have been observed (a detection bit was set), the mappings in the ART are now stale. Thus the relevant ART/SS entries have to be invalidated/updated (or simply flush all).

ART entry invalidation/eviction: If space is needed in the ART, a victim entry can be chosen and evicted without correctness consequences as accesses made with a non-

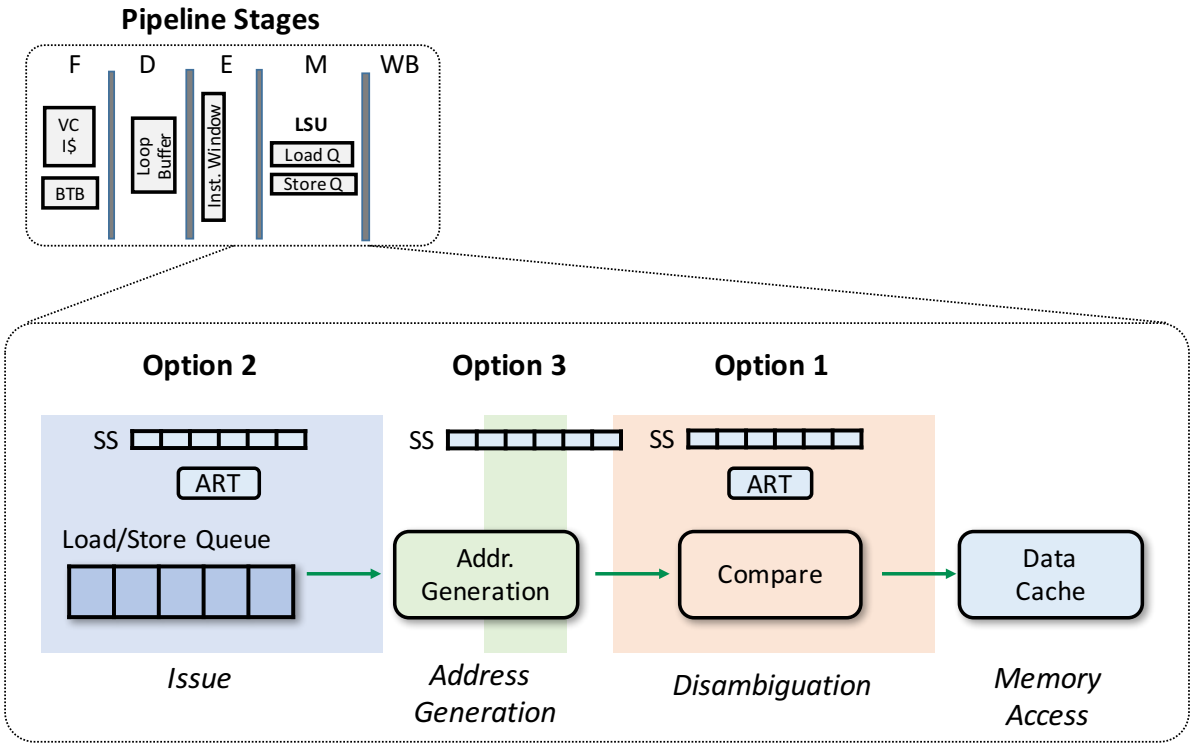


Figure 4.6: Different placement of SS and ART lookup in pipeline stages

leading virtual address will simply miss (and result in recreation of the ART entry, Case ③). However, the corresponding counter/bit in the SS should be updated so that it can retain its effectiveness. Page information change for non-leading virtual pages also triggers the invalidation of the related ART entries. This is because the information (e.g., permission bits or virtual-to-physical address mappings) of the corresponding entries is no longer valid.

4.2.4 Design Choices for SS and ART Lookups

In the previous sections, we described the overall operations of our proposal by conceptually assuming that a synonym signature (SS) is consulted prior to the phase of looking up the L1 cache. This may lengthen the cache access path, although the SS is small. We can still expect most of the power/energy benefits of using virtual caches, but perhaps not the potential latency benefits. However, the SS and ART (if needed) accesses could be

performed in a manner that may not affect the timing of the actual L1 access, depending upon the microarchitecture.

Since memory operations typically require additional steps between an address generation and submission to the L1 cache (e.g., load/store queue, disambiguation), there are several choices for where/when the SS and ART are consulted in the pipeline stages. Figure 4.6 gives an overview of different placements where the SS and ART can be consulted in terms of memory accesses. The SS and ART can be accessed in parallel with memory disambiguation (Option 1 in Figure 4.6). In practice, the SS can be accessed *before* the address generation, based upon the contents of a base (or segment) register (Option 2 in Figure 4.6). If the address generation is a multi-step process, the SS could be accessed after intermediate steps of the process (Option 3 in Figure 4.6). By consulting an SS in advance (Options 2 and 3) or in parallel with other operations in the pipeline (Option 1), the ART could be selectively accessed before L1 cache lookups (or in the pipeline). Option 2 looks appealing, since it can be applied to both I and D caches, and most of the energy and latency benefits can be achieved without an increase in design complexity. In addition, SS and ART accesses could be bypassed for store requests (e.g., 30% of data accesses) because write synonym accesses are rare [13]. In practice, this does not pose a correctness problem because a memory access with a non-leading virtual address will be detected at the ASDT and replayed to correctly make a proper memory access.

4.2.5 Supporting Virtual Memory Features with VC-DSR

In Section 2.2.2, we discussed virtual memory features and challenges of supporting them with VIVT caches. We now discuss how to efficiently support them with our proposal.

4.2.5.1 Virtual page information changes

When page information (e.g., mapping and permission) changes, all the entries corresponding to that page in all the components supporting VC-DSR need to be invalidated

(or updated) to ensure consistency (as described in Section 4.2.3.5). The event is triggered by TLB invalidations. This potentially requires multiple ASDT lookups to search for an ASDT entry that has the target virtual page as a leading virtual page because consulting an ASDT is performed with a physical address. In practice, such events can be filtered out with a simple filter based on the leading virtual page information of all ASDT entries. For example, a Bloom-filter can be employed. This filter is accessed with target virtual page numbers of TLB invalidations, and each bit indicates likelihood that there is a corresponding ASDT entry having the target virtual page as a leading virtual page. When a matching ASDT entry is found, only the corresponding lines can be selectively evicted from the VC with a bit vector. Thus, this does not have a significant impact on the effectiveness of our proposal, although such events are not rare [103]. For page mapping changes, some OSes, e.g., x86 Linux, flush whole caches (or selectively invalidate cache lines of the relevant pages depending on the ranges of the target memory region). In this case, we do not have to additionally flush all caches again.

4.2.5.2 Permission check

Depending on whether an access is to a leading or non-leading virtual page, the location of checking (keeping) the page permission bits is different. For the former case, each line in the VC tracks the permission bits, while an entry in the ART maintains the permission bits for a non-leading virtual page. The permission check is done when a matching entry is found. On virtual cache misses, the permission check is performed by consulting a traditional TLB. Once a page permission mismatch occurs at the ART or L1 VC, the request is handled like a permission mismatch at a conventional TLB, which carries out all the actions of an OS page fault handler for possible exceptions such as a store to a read-only-page or execution of data in a non-executable page.

If the active synonym remapping phase is bypassed for stores (Section 4.2.4), the ART does not necessarily need to track permission bits for non-leading virtual pages. Stores with

non-leading virtual addresses will simply cause cache misses to arise, and the permission check will be performed by consulting a TLB as normal. This does not result in any correctness issues because of the replay mechanism supported by the ASDT.

4.2.5.3 Challenges due to virtual address synonyms

We already discussed how to resolve two challenges (i.e., data consistency issue and cache coherence) in the previous section (Section 4.2.3). Our proposal does not face the data consistency issue. This is because VC-DSR uses a *unique* leading virtual address for virtual cache operations, which does not allow data duplication regardless of types of synonym accesses (i.e., read-only or read-write synonym accesses). For cache coherence requests, in addition, we can take advantage of the information from the ASDT to efficiently perform physical to leading virtual address (reverse) translation.

In this section, we will focus on discussing other issues resulting from virtual address synonyms for operations in pipeline stages that use (only) virtual addresses with L1 virtual caches (i.e., sequential semantics of a program, memory consistency, and buffered instruction consistency issues).

Supporting a virtually addressed Load-Store Unit: Using virtual addresses for a conventional write buffer (or store queue) can result in a violation of sequential semantics: a load may not identify the matching latest store due to synonyms, and vice versa. Thus, stale data could be returned from caches or from a matching older store. In the similar vein, using virtual addresses for a load queue could potentially violate memory consistency models, when a coherence event, e.g., an invalidation or eviction, occurs in L1 caches, a load that has been carried out (speculatively) for the corresponding data may need to be identified and replayed [18, 39, 76, 93, 97]. In some commercial processors, these issues are handled by finding potentially offending loads by matching the physical addresses and replaying them. For virtual caches, however, synonyms can complicate their identification.

Most of the prior literature has not discussed these issues, and the proposed solution

for the former issue may not be efficient [88]. However, VC-DSR can easily handle all of them, as follows: the key idea is to employ a leading virtual address (LVA) as an alternative to a physical address. Accesses with a non-leading virtual address always cause VC misses and are eventually replayed with the LVA via an ART (Case ③). Thus, once a load/store is (speculatively) executed, its LVA is identified and kept in the load/store queue. The unique LVA of each load/store is used to find potential violations.

Consistency issues due to self-modifying codes: Self-modifying code is used to dynamically change the instructions of a program during runtime. Examples include a just-in-time (JIT) compiler, dynamic code optimization, and use of a debugger. A processor core performs this modification through store operations, and the up-to-date instruction can reside in data caches. In this case, to prevent data consistency issues between an instruction cache and data cache, the corresponding stale instruction cache line should be invalidated (or the instruction cache flushed). We can perform this by identifying its LVA via an ASDT for an instruction cache. The use of a unified ASDT for L1 instruction and data cache could simplify the operations.

Invalidating the line from the instruction cache is not enough because the stale instruction could have already been fetched and be being buffered in the pipeline stages (e.g., a trace cache, instruction window, etc.) [1, 64, 92]. Some processors use a *unique* physical address to identify the modified instruction within the pipeline stages [48]. In a similar vein, we can keep using its *unique* LVA for that, which provides an illusion of the conventional approach.

4.2.6 Other Design Issues

We now discuss several other issues that need to be addressed for a practical virtual cache design. Most of these issues also arise in other VC designs.

Non-cacheable data access: The cacheability of accessed data is recognized when a corresponding TLB entry is looked up. VC-DSR consults a TLB only when virtual cache

(VC) misses occur. Thus an access to non-cacheable data first triggers an L1 VC miss, then, the cacheability will be identified at a TLB. The latency resulting from the VC lookup may increase the access latency. However, the overhead (e.g., 1-2 ns) is a very small portion of the overall latency (e.g., 50-100 ns for memory reference). Hence, the actual timing overhead will not be significant, even though such accesses are not rare.

Hardware page-table walk based on physical addresses: Some architectures (e.g., x86) support a hardware page-table walker [48, 50]. The page table entry (PTE) may be accessed using a virtual address by the OS, and thus may end up in the cache, whereas it is only accessed using physical addresses by the page table walker. The key to handling this case is the observation that if a line (from a physical page) resides in the L1 cache, there will be a corresponding entry in the ASDT. An access made with a physical address will consult the ASDT to obtain the leading virtual address (LVA) with which the block is being cached, and access the cache (if the block is there) with that LVA.

Supporting precise interrupts (memory exception checking): Modern general-purpose processors support precise interrupts. The oldest instruction is committed if it is completed without any exceptions. Then architectural state is updated. Regarding memory operations, recoverable memory exceptions, e.g., page faults, should be checked before committing the instruction. Conventional systems with physical caches perform the exception check *early* by consulting a TLB [87]. TLB hits indicate that the access is *exception-free* because it guarantees the *existence of a valid virtual-to-physical page mapping* (i.e., no page fault).

With virtual caches, conventional TLBs are consulted only when virtual cache misses occur. Thus, we need to check for exception conditions in a different manner, like the permission check we discussed above. For accesses with leading virtual addresses, virtual cache hits indicate that the access is exception-free. ART hits guarantee the accesses is page-fault free for non-leading virtual addresses. On virtual cache misses, the exception check will be performed as normal by consulting a TLB. In this case, however, identifying the exception is slightly delayed relative to the other two cases (i.e., ART hits and VC hits)

because VC tag comparison is needed first. This does not cause performance degradation with our proposal. In practice, the delay itself is very short: one cycle for tag comparison of an L1 VC. The actual overhead will be imposed solely when there are no available entries in instruction queues (i.e., a structural hazard) due to the delayed exception check for (rare) virtual tag misses.[3] Hence, it is highly unlikely to cause a structural hazard.

Some memory consistency models (e.g., Total Store Order (TSO)) employ a write buffer that sits between a processor and an L1 cache [97]. The write buffer holds *committed* stores. As discussed above, memory exception conditions should be checked before committing memory operations. With virtual caches, the exception check is performed at different locations. In most cases, L1 VC tag matches are required (i.e., except for ART hit cases). Regarding managing stores in a write buffer, the exception check can be efficiently used by leveraging information from the tag comparison. First, it can be considered as pre-tag comparison for a store operation if the corresponding data resides in the cache (i.e., VC tag hits). We can identify where the corresponding data is placed (i.e., a way number) in advance. The information can be kept in the matching entry of a write buffer. With the information, we can directly access/update the data of the matching cache line when store operations are actually performed from a write buffer. In addition, the exception condition check can play a role of prefetching data or write coherence permission of a store on VC misses [39, 87].

Large pages: For large pages, individually identifying lines from the page resident in the L1 virtual cache (VC) is not practical since the bit vector would be extremely large. However, keep in mind that a bit vector is simply an optimization to invalidate lines in the VC before evicting the corresponding ASDT entry. Without precise information about individual lines, the lines from a page could be invalidated by walking through the lines in the VC to search for lines from the page, and using the associated counter to track when

[3]When a load instruction reaches the head of the reorder buffer, it has already obtained data from the caches, which means it has already been checked for exception conditions. Thus, the extra delay for load misses can be easily hidden.

the necessary operation is completed (counter is zero). We can minimize the likelihood of this potentially expensive operation by not evicting an ASDT entry for a large page unless absolutely necessary (e.g., all candidate entries are for large pages—an extremely unlikely event).

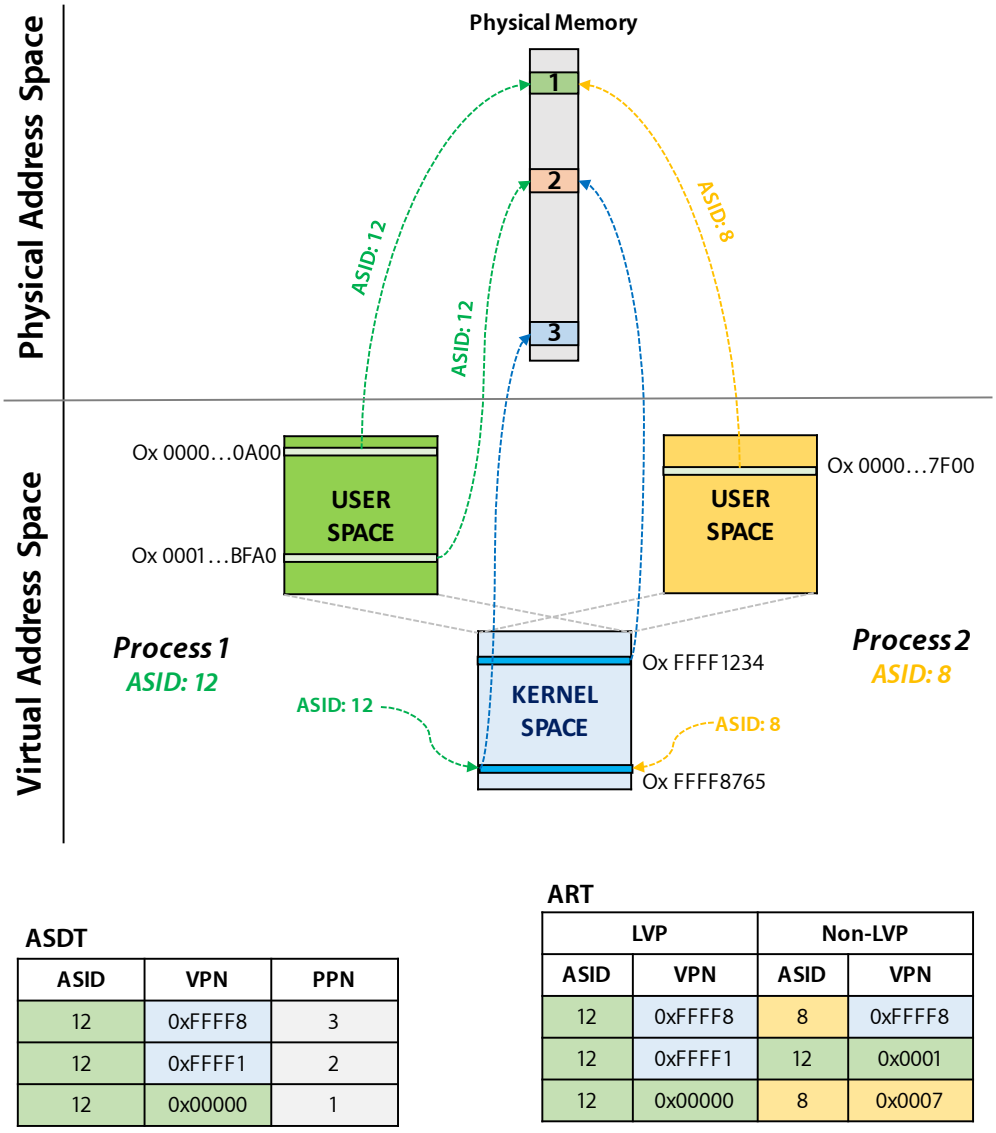
As an alternative, we can take advantage of OS memory allocation properties. Basically, synonyms are aligned on page (4KB) boundaries. Hence, a large page (> 1MB) can be viewed as a collection of small 4KB pages. By leveraging this property, the baseline ASDT, considering only 4KB pages, can be employed to handle accesses for larger pages.

Supporting H/W Prefetchers: H/W prefetchers employed by modern processors can be seamlessly integrated with our proposal. For example, a stride prefetcher and stream buffer [10, 77] can use the physical address from the translation caused by a cache miss (i.e., no additional TLB lookups). Multiple prefetches initiated in a (large) page can also hide the potential overhead of multiple TLB lookups (or ASDT lookups if prefetched data is placed in the L1 VC).

4.2.7 Optimizations

Last LVA Register: Though few, ART accesses can be reduced even further by exploiting the fact that often successive references are to the same page (especially true for instructions). Similar to the VIBA/PIBA register used for **pre-translation** for instruction TLB accesses in the VAX-11/780 [27], we can employ a **Last LVA** (LLVA) register, which maintains the leading virtual address (LVA) for the last page that was accessed, close to (or within) address generation. Thus consecutive access to the same page need not consult the ART.

Kernel Address Space Access: Each process has its own (user) virtual address space, while a kernel virtual address space is unique and globally shared by all processes (refer to Figure 3.1). To address homonym problems, a virtual address includes an ASID. By using the ASID information as a part of a virtual tag, we can distinguish accesses with the same virtual address, but from different user virtual address spaces. Different from



ASDT

ASID	VPN	PPN
12	0xFFFF8	3
12	0xFFFF1	2
12	0x00000	1

ART

LVP		Non-LVP	
ASID	VPN	ASID	VPN
12	0xFFFF8	8	0xFFFF8
12	0xFFFF1	12	0x0001
12	0x00000	8	0x0007

Figure 4.7: Example of kernel space accesses

user spaces, however, the kernel space is globally shared. Thus, no homonym issues arise among kernel space accesses from different processes. This indicates that we do not have to consider ASID information for such accesses. However, our baseline design employs ASID information regardless of the type of virtual address space to be accessed (i.e., user or kernel space). This may lead to *unnecessary* active synonyms among kernel space accesses, because their virtual tags become different solely due to the difference in the ASIDs.

Figure 4.7 depicts this scenario. The top part of this figure shows virtual-to-physical

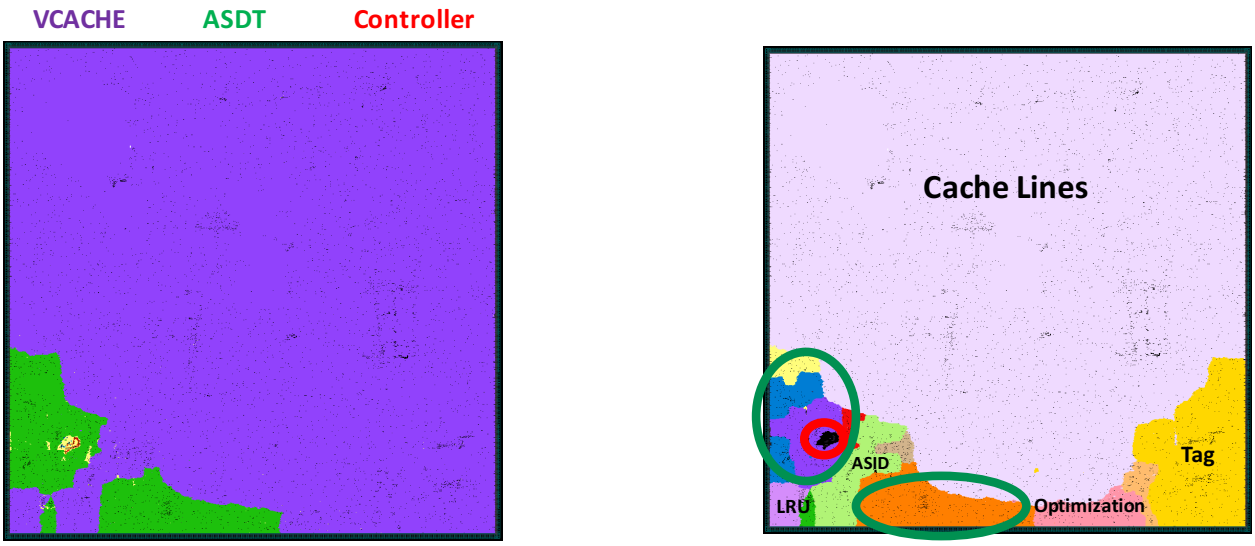
page mappings of two processes. The bottom part of the figure shows the information of ASDT and ART entries. Physical page 3 is mapped to a virtual page (VPN: `0xFFFF8`) in the shared kernel space. A virtual address (e.g., `0xFFFF8765`) in the virtual page is accessed by two processes with different ASIDs, i.e., `12` and `8` for process 1 and 2, respectively. Let's assume that the physical page 3 is accessed by process 1 for the first time. Then, an ASDT entry is allocated for the physical page 3 (see the first entry), and it has `0xFFFF8` for the leading virtual page (VPN) with ASID `12`. An access with the same kernel virtual address from process 2 will cause a virtual cache miss; ASID `8` is used although the virtual address itself is identical. For the same reason, this access is considered as an active synonym access with a non-leading virtual address at an ASDT, which allocates a new ART entry (see the first ART entry in Figure 4.7).

To avoid the associated overhead, a *unique, shared* ASID can be used for accesses to the *unique, shared* kernel space, while an original ASID is used for corresponding user space accesses. That is, we need to perform selective remapping of an ASID for kernel space accesses. There are several ways of identifying these accesses. First, we can take advantage of memory mappings of OSes. For the current design (x86-64 Linux) supporting 48-bit virtual address, the OS takes the upper half of the address space. Thus, we can dynamically identify that by reading the 48th bit of a virtual address. Second, we can also employ a global bit/flag in a page-directory or page-table entry (e.g., x86) [13] via a TLB. We need to maintain the information of the leading virtual page in the corresponding ASDT and VC entries.

4.2.8 Storage Requirements

The storage requirements for the proposal are quite modest. An over-provisioned ASDT (e.g., 128 entries) requires 2.4KB.[4] A 32-entry ART (large given the number of pages with

[4]Half of the storage is taken by a 64-bit bit vector per entry for a 4KB page with 64B lines. To further reduce the overhead, the information can be maintained in a coarse grained manner, although it may lose accuracy.



(a) Floorplan with modules

(b) Submodules of the floor-plan

Figure 4.8: Implementation Design of an L1 cache with an ASDT.

active synonyms (OB_1 and OB_2)), along with 256-bit SS (with 5-bit counters), needs approximately 550B. A 256-entry ASDT would also imply 8 extra bits with each VC line if eliminating the TLB access for cache line eviction was desired.

To estimate more accurate area overhead, we implemented a virtual cache with an ASDT in Verilog, which is synthesized in 32nm. The virtual cache module is a non-blocking design with a single port. It consists of sub-modules like an LRU module, virtual tags, cache lines (64B), and ASDT index per line as an optimization. The ASDT module is comprised of an LRU module, tags (physical page numbers), leading virtual page information, and bit vector per entry. There is a controller module with a finite state machine managing the interaction between the VC and the ASDT for cache lookups. Figure 4.8 (a) and (b) show the floor-plan of the three modules and their sub-modules, respectively. We notice that most of the area (93%) is dominated by the virtual cache (purple part). The rest of the area (7%) is for the ASDT and controller. According to Figure 4.8 (b), about 4% of the virtual cache area is the extra part to support our proposal: ASID, permission bits, additional virtual tag bits, and ASDT indexes (i.e., optimization). If we consider the overall cache hierarchy including multiple L2 caches and a shared L3 cache for modern processors, the

	1	2
CPU type	AtomicSimple	3GHz, 8-way Out-of-Order
Num. CPU	Single-Core	
L1 Virtual $	Classic $ Model	Ruby 3-level hierarchy
	Separate I and D$, 32 KB, 8-way 64B Line, 1ns access latency	
L2 Physical $	4096KB 16-way	256KB, 8-way, 3ns latency
L3 Physical $	None	4096KB, 16-way, 10ns latency
Main Memory	3GB Simple Memory Model, 50ns access latency	

Table 4.1: System Configurations

TPC-H [101]	1-21 Queries, 1GB DB on MonetDB[17]
SPECjbb2005 [98]	2 Warehouses
Memcached [36, 69]	Throughput Test, 3GB Twitter Data-set
bzip2, h264ref [44]	reference input size
Raytrace [38]	desktop input size
Stream (Copy, Add, Scale, Triad) [38]	desktop input size, sequential execution of four different Stream workloads

Table 4.2: Workloads

additional storage overhead is not considerable.

4.3 Evaluation

We now evaluate the effectiveness of VC-DSR. First, Section 4.3.1 describes the evaluation methodology and workloads. Then, we evaluate how much dynamic energy consumption can be saved in Section 4.3.2, briefly evaluate the latency benefits in Section 4.3.3 and compare VC-DSR with three other virtual cache proposals in Section 4.3.4.

4.3.1 Evaluation Methodology

We attached modules to simulate our proposal in the Gem5 x86 simulator [15]. To carry out a meaningful study of virtual caches, especially for synonyms, two essential features have to be considered for the experimental methodology. First, the infrastructure needs to support

a full-system environment so that all kinds of memory access, e.g., access to dynamically linked libraries and access to user/kernel address space, can be considered. Second, the experiments need to be run long enough to reflect all the operations of a workload and interaction among multiple processes running on the system. Detailed CPU models in architecture simulators [15, 79] supporting a full-system environment are too slow, and instrumentation tools [67], while relatively fast, provide inadequate full-system support.

For most of the evaluation, we use a functional CPU model (AtomicSimple[5]) running Linux and simulate a maximum of 100B instructions for each benchmark. The default system configurations are presented in the left column of Table 4.1. Although this does not provide accurate timing information, it provides information regarding all memory accesses and is fast enough to test real world workloads for a long period. We also evaluate potential performance impact conservatively by using more detailed CPU (Out-of-Order) and cache models. The configurations are presented in the right column of Table 4.1 and for this we simulate up to 1B instructions.

We model hardware structures supporting VC-DSR at a 32 nm process technology with CACTI 6.5 [73]. For benchmarks, we use several real world applications, e.g., DB, server, and mobile workloads, described in Table 4.2.

4.3.2 Dynamic Energy Saving

We first consider how much dynamic energy consumption for TLB lookups can be saved with VC-DSR. Ideally, the benefit will be proportional to the cache hit ratio for L1 virtual caches since a TLB is consulted only when cache misses occur. For VC-DSR, in practice, the ART is selectively looked up to obtain a leading virtual address for synonym access, and the ASDT is also referenced for several cases such as cache misses, coherence, TLB invalidations, etc. The organization of the needed structures (e.g., ASDT size) could also

[5]More sophisticated processors may show different instruction cache access patterns, to some extent, by using diverse techniques to improve the efficiency of the fetch stage, e.g., fetch/loop buffers, branch predictors, prefetchers, etc.

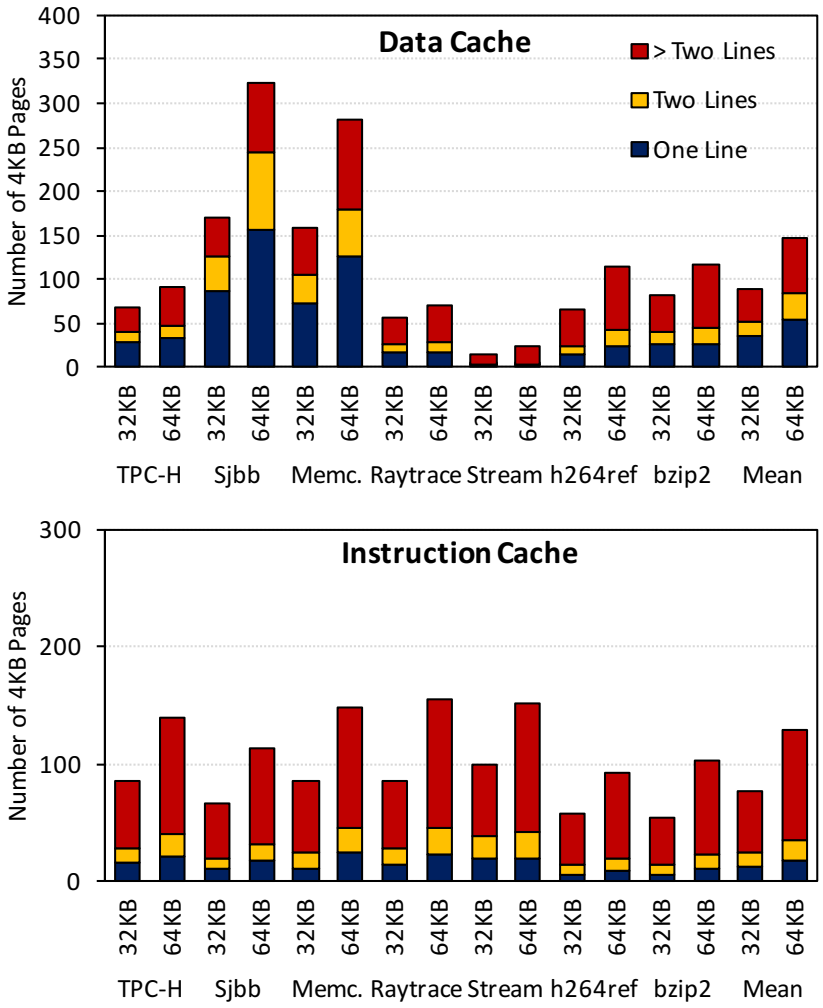


Figure 4.9: Diversity in Pages whose Data Resides in Caches

affect cache misses. These aspects have all to be accounted for when evaluating the overall benefits.

ASDT size: An ASDT with fewer entries tracks fewer pages, which could evict pages whose lines are not dead yet from the cache in order to track newly referenced data. This could increase the cache miss ratio, degrading performance. Hence, an ASDT needs to be adequately provisioned to prevent this case.

Figure 4.9 shows the average number of distinct 4KB pages from which blocks reside in 32KB (64KB) L1 physical caches with a 64B line size. Each bar has three sub-bars. They are classified according to the number of cached lines from each page. Even though 512

	W/O Op		Kernel Op		LLVA Op		Kernel+LLVA	
	Acc.	**Hits**	**Acc.**	**Hits**	**Acc.**	**Hits**	**Acc.**	**Hits**
Memcached	39.3	26.8	0.97	0.63	13.9	10.1	0.05	0.03
TPC-H	3.95	0.92	0.21	0.15	0.52	0.3	0.04	0.03
SPECjbb	0.35	0.27	0.33	0.27	0.04	0.02	0.04	0.02
bzip2	0.12	0.1	0.06	0.05	0	0	0	0
h264ref	0.01	0.01	0	0	0	0	0	0
Stream	3.02	2.02	2.86	1.91	0.8	0.5	0.79	0.44
Raytrace	5.54	3.19	5.53	3.18	0.2	0.1	0.17	0.06

Table 4.3: Analysis of ART Accesses: data accesses

(1024) different pages are possible, typically there are an average of less than 150 distinct pages at a given time in most cases. There are fewer distinct pages in instruction caches (e.g., 80) than in data caches (e.g., 100). Moreover, most instruction pages have more than 2 lines cached, whereas many data pages have only 1 or 2 lines cached. The data suggest that a middle-of-the-road sized ASDT is enough and that a smaller ASDT can be used for instruction caches: 128 and 256 entries for 32KB L1 I-cache and for D-cache respectively.

Based on this configuration, the experimental results show that an ASDT entry eviction occurs rarely (less than once per 100 L1 cache misses [6]) and that this results in at most 1 or 2 VC line evictions. The results suggest that the overhead resulting from the ASDT entry eviction is not significant and that identifying VC-resident lines individually with a bit vector is likely to be useful.

ART access: Table 4.3 and 4.4 shows how efficiently ART lookups can be managed with a 256-bit SS. We assume that the SS is consulted before the address generation, with bits 19-12 of the base register with a 4KB fixed offset (Option 2 in Section 4.2.4). The SS decision not to consult the ART is considered valid if these bits do not change as a result of address generation.[7] For these results, we consider 32KB L1 virtual caches and the ART has 32 entries (8 sets and 4 ways). The data presented in Table 4.3 and 4.4 is: 1) percentage of

[6] On average, the L1 data cache accesses show about 3% miss ratio for the tested workloads. Thus, the ASDT entry evictions arise less than once per about 3333 L1 data cache accesses.

[7] Accessing the SS with a more complex hash (e.g., using more address bits or the sum of the ASID and the address bits) further reduces the number of ART accesses.

	W/O Op		Kernel Op		LLVA Op		Kernel+LLVA	
	Acc.	**Hits**	**Acc.**	**Hits**	**Acc.**	**Hits**	**Acc.**	**Hits**
Memcached	56.9	28.1	3.41	2.13	1.61	0.9	0.08	0.06
TPC-H	3.86	1.5	1.88	0.26	0.12	0.1	0.02	0.01
SPECjbb	0	0	0	0	0	0	0	0
bzip2	0.03	0.03	0	0	0	0	0	0
h264ref	0	0	0	0	0	0	0	0
Stream	0.04	0.03	0	0	0	0	0	0
Raytrace	0.01	0.01	0	0	0	0	0	0

Table 4.4: Analysis of ART accesses: instruction accesses

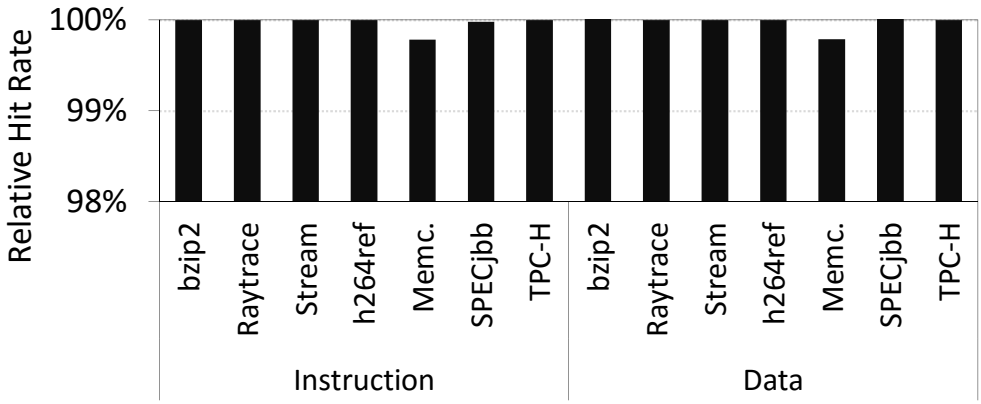


Figure 4.10: Analysis of VC-DSR L1 VC Hits

all cache accesses that consult the ART after the SS lookup (**Acc.**) and 2) percentage of all cache accesses that find a matching entry in the ART (**Hits**), without any optimizations (**W/O Op**), with an LLVA register (**LLVA Op**), and with the optimization for accesses to the kernel virtual address space (**Kernel Op**).

For most cases, the small SS can filter out a significant number of ART lookups. For example, for *tpc-h*, the SS filters out about 96% of instruction accesses, and only 1.5% of accesses hit in the ART without optimizations. For *memcached* showing most frequent active synonym accesses, notice frequent ART lookups (i.e., SS hits) in the base case. The use of the optimization for kernel virtual address space significantly reduces the number of ART accesses. This is because most of the active synonym accesses for *memcached* occur in the kernel space. The optimization completely prevents such accesses from being considered as

active synonym accesses. Thus this makes the SS less conservative, considerably reducing the number of SS hits. Employing an LLVA register is also effective. For *memcached*, notice a significant reduction in ART lookups for instruction accesses. However, due to relatively low temporal and spatial locality of data accesses, we can still see noticeable ART accesses (about 14%) with an LLVA register. This can be reduced to almost zero with the optimization for kernel virtual address space.

Since very few ART accesses end up finding a matching entry, a larger sized SS does an even better job of filtering out unnecessary ART accesses (data is not presented). To be conservative in presenting our results, we use a 256-bit SS for the rest of the evaluation.

L1 VC hits: Figure 4.10 shows the hit rate of a 32KB L1 VC-DSR, relative to a PIPT cache; the false misses due to synonymous accesses (Case ③ in Figure 4.4) are treated as a miss. Notice almost the same results across all of the workloads regardless of instruction and data accesses.

Energy saving: Putting it together, we now consider how much TLB lookup energy can be saved with VC-DSR. Figure 4.11 presents the breakdown of dynamic energy consumption accounted for by each component of VC-DSR. A 32 entry, fully associative 3-ported TLB is used as a baseline. 100% indicates the TLB lookup energy of the baseline. The without optimization (W/O) bars correspond to the baseline design that we described in Section 4.2.2 and the with optimization (W) bars include the optimization mentioned in Section 4.2.3.5 and 4.2.7. For virtual cache lookups (bars with L1 Cache label), we consider the extra overhead due to reading additional bits (e.g., ASID and extra virtual tag bits) on every access as well as the false misses.

A few points before discussing the results. First, TLBs with more than 32 entries are common in real designs, e.g., ARM Cortex [7] and AMD Opteron [42]. They consume significantly higher energy than the 32-way baseline, and thus we can expect more energy benefits when they are used as a baseline. Second, we use the same TLB organization as the baseline with VC-DSR, although VC-DSR would permit us to have a larger, slower, lower-

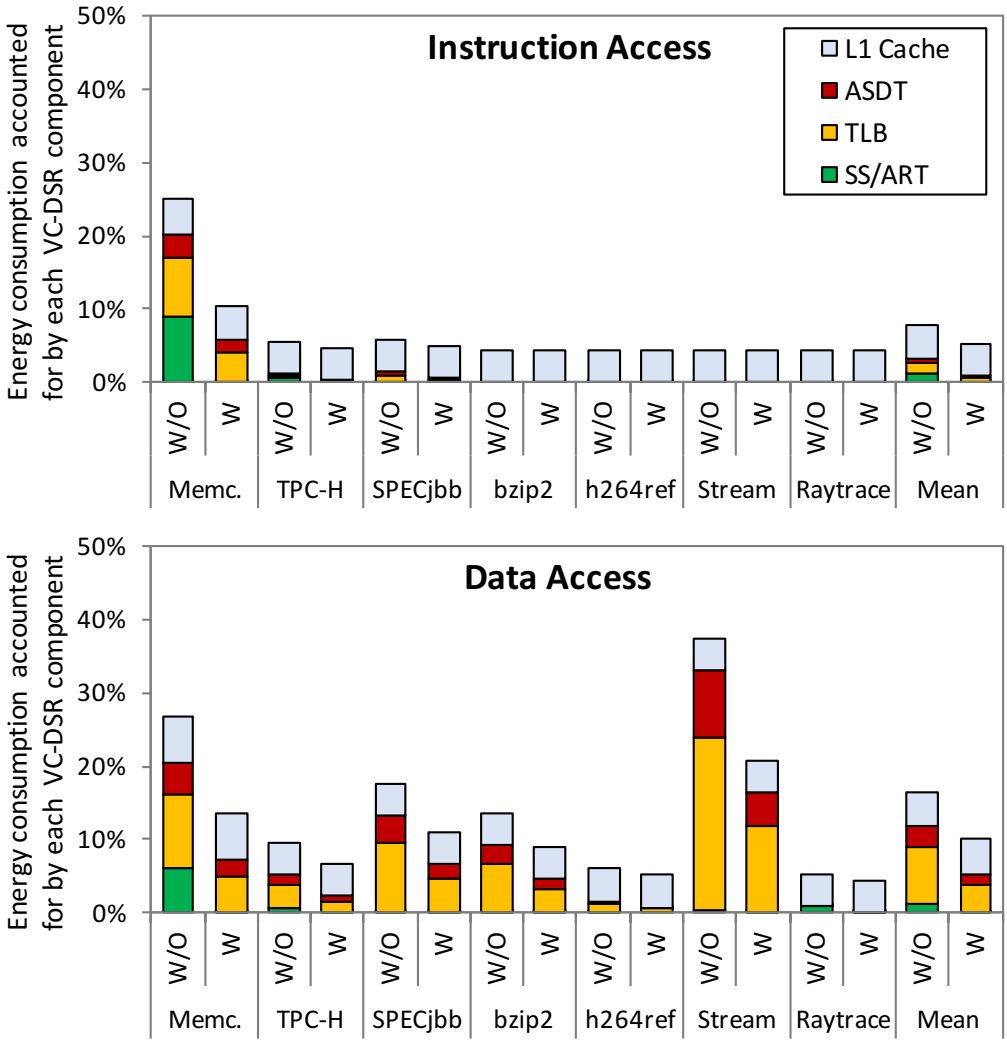


Figure 4.11: Breakdown of Dynamic Energy Consumption for VC-DSR (baseline 100%, lower is better). We model a TLB that is fully associative (32-way) with 3 ports (one read, one write, and one read/write). Its dynamic read energy per access is 0.003029 (nJ). The modeled ASDT is an 8-way set associative array with a single r/w port. It shows 0.0027 (nJ) dynamic read energy per access. For remapping operations (i.e., SS/ART bars in this figure), 0.00107 (nJ) is consumed per access. The VC consumes 0.0003 (nJ) more than the baseline physical cache per each cache lookup due to reading additional virtual tag information (i.e., L1 Cache bars).

associativity, fewer ports design, which has lower energy consumption (and miss rate). Third, our proposal removes constraints on the organization of virtually indexed caches (i.e., a larger associativity). Thus we can also expect lower power/energy consumption for L1 cache lookups [13, 49]. We do not consider these benefits from the flexible design choices of VC-DSR in our comparison, thereby disadvantaging VC-DSR.

We will first consider the results without optimization (bars with W/O label). Notice about 93% and 84% (average) energy saving for an instruction and for data TLB, respectively. We observe that a small portion of the energy overhead is accounted for by reading the additional bits on every L1 VC lookup, regardless of instruction and data accesses. The consumption is dominated by the TLB and ASDT lookups on cache misses (especially for data accesses). The energy overhead except the overhead for the L1 VC lookups can be reduced further by leveraging the proposed optimization (Section 4.2.3.5 and 4.2.7). We can save the TLB and ASDT lookup overhead for handling an evicted line on every miss by keeping the ASDT index with a cache line. The use of an LLVA register can reduce ART lookups. We observe that all the optimizations reduce the energy consumption by half (bars with W label), resulting in about 95% and 90% energy saving for an instruction and for data TLB, respectively. In comparison, an ideal virtual cache would achieve about a 99% and 96% reduction for an instruction and data TLB[8], respectively, in this situation. *These results suggest that VC-DSR achieves most of the energy benefits of employing pure, but impractical virtual caches.*

4.3.3 Latency and Timing Benefits

While improved cache access latency—the original motivation for virtual caches—and the consequent performance impact, was not our primary driver, we do mention the quantitative timing benefits we obtained in our experiments (the configuration described in the right side of Table 4.1). We assume that one cycle is consumed by consulting a TLB, one

[8]The address translation via a TLB is necessary on a virtual cache miss even when the ideal virtual cache is considered. Thus, achieving 100% reduction is not possible.

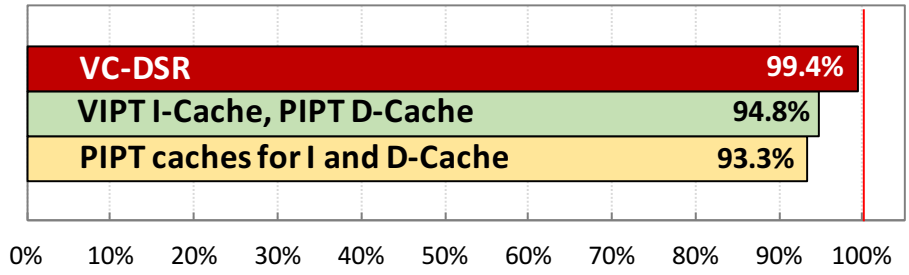


Figure 4.12: Performance Analysis of VC-DSR

cycle for consulting an ART and three cycles for active synonym detection (TLB and ASDT lookups). The optimizations discussed above and the virtually addressed Load-Store Unit discussed in Section 4.2.5 are employed.

Figure 4.12 shows relative performance of various L1 cache configurations. The baseline uses L1 VIPT caches for both instruction and data accesses, which can completely hide the one cycle latency overhead of TLB lookups. We saw a trivial (0.6%) timing overhead for VC-DSR. We can expect such overhead could be hidden by taking advantage of flexible design choices supported by the usage of virtual caches, e.g., TLB organization lowering misses [14] and virtual cache organization fully employing spatial locality of memory access [68]. *The results suggest that VC-DSR also achieves most of the latency benefits of employing virtual caches.*

In addition, some commercial designs [7, 8] use a PIPT data cache to simply avoid problems with synonyms. VC-DSR has a significant timing benefit over such designs.

4.3.4 Comparison with Other Proposals

There has been a plethora of proposals for virtual caches, each with their own pluses and minuses. We compare VC-DSR with three of them: Opportunistic Virtual Cache (OVC) [13], which is a recent proposal, Two-level Virtual-real Cache hierarchy (TVC) [104], which is an early proposal that has several pluses for a practical implementation, and Synonym Look-

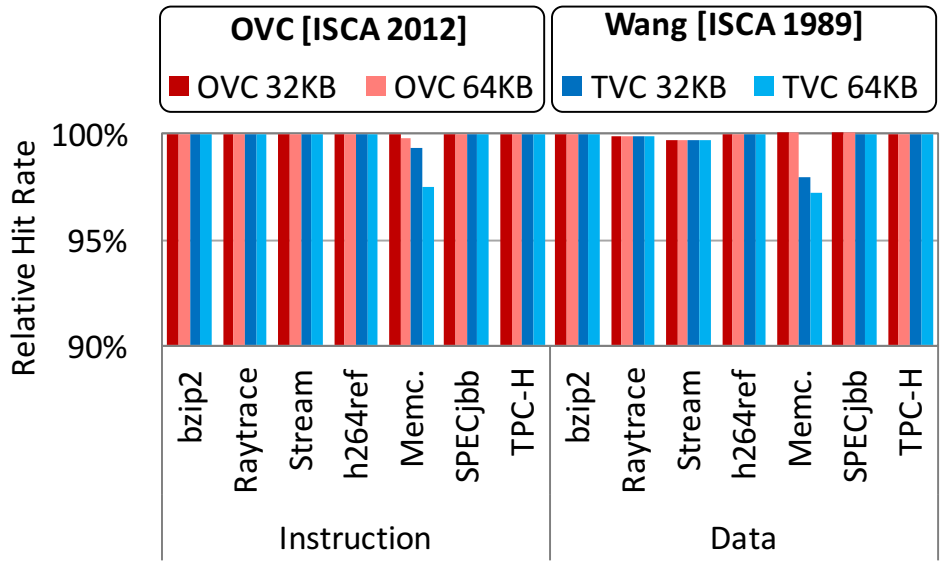
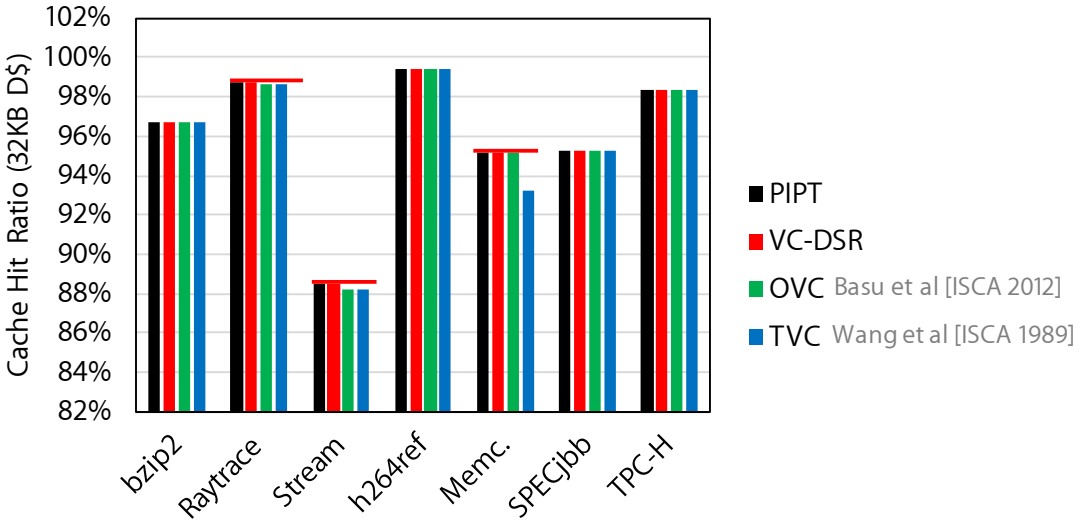


Figure 4.13: Comparison with Other Approaches (Relative cache miss ratio)

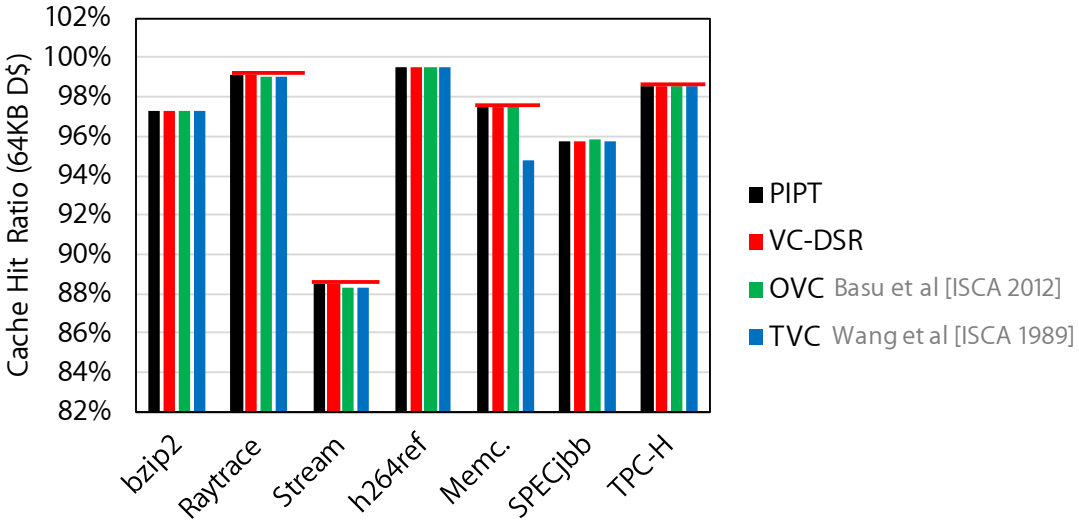
aside Buffer (SLB) [88], which takes a somewhat similar approach for handling synonyms (see detailed differences in Section 6.1.1). Note that OVC and SLB require assistance from software (i.e., OS supports), whereas VC-DSR is a *software-agnostic* approach. For OVC, we assume an optimistic case where all lines will be cached with virtual addresses, saving the energy consumption for TLB lookups as much as possible. For TVC, we assume that an ASID is employed to address the issue of homonyms, which excludes the impact of flushing virtual caches on a context switch [5].

Figure 4.13 presents the L1 cache hit rate of two approaches, OVC and TVC, relative to VC-DSR for 32KB and 64KB caches. Figure 4.14 shows (absolute) data cache miss ratios of them. Notice that VC-DSR achieves a slightly higher hit rate due to more efficient handling of active synonym accesses. OVC allows duplicate copies of data to be cached for read-only synonyms in a virtual cache, resulting in a reduction in cache capacity.[9] This overhead could be noticeable for caches with smaller associativity or in systems with kernel same-page merging dynamically allowing multiple processes to share the same physical page (e.g., virtualized environments) In addition, this approach could impose restrictions on

[9]VC-DSR does not allow data duplication for both read-only and read-write synonyms in a virtual cache, because data is cached with a unique virtual address during data residence in the virtual cache.



(a) 32KB data cache



(b) 64KB data cache

Figure 4.14: Comparison with Other Approaches (Cache miss ratio)

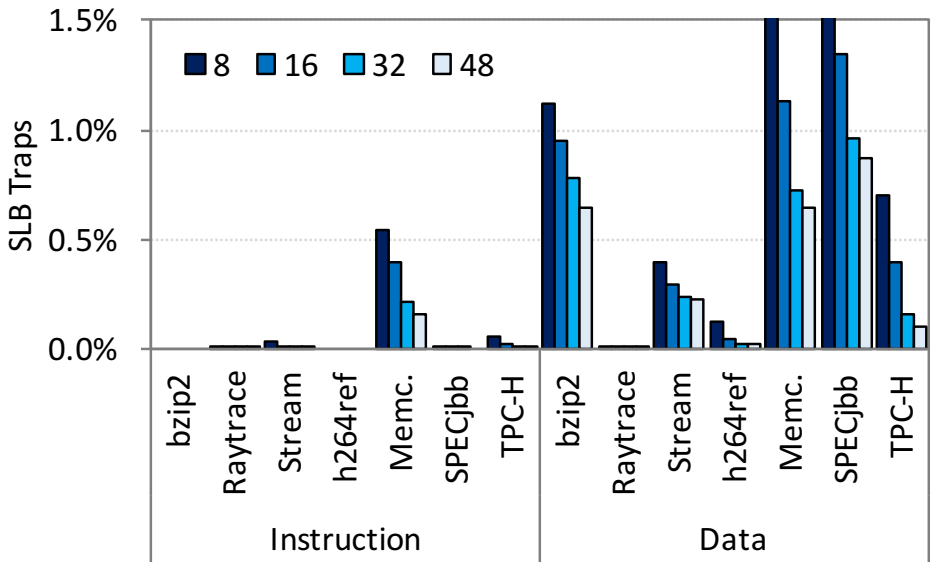


Figure 4.15: Frequency of SLB Miss Traps

designing a viable practical implementation because the issues of virtually addressed Load-Store Units (Section 4.2.5) may not be efficiently handled.

TVC does not keep duplicate copies of data in virtual caches like our proposal, but accesses with a synonymous address that is different from a virtual address used to cache data result in cache misses. This requires additional operations to cache a synonym copy with a most recently referenced virtual address. It is possible for the synonym copy to be relocated to a different set, and thus TVC also has the overhead of data replication in virtual caches. The cost of handling synonymous accesses could be more noticeable depending on the multiprogramming environment and microarchitecture design (e.g., SMT). Furthermore, TVC works only with an inclusive cache hierarchy. The management of back-pointers with L2 cache lines could complicate its design when virtual page information changes; back-pointers of all the corresponding L2 cache lines may need to be updated as well.

For SLB, performance degradation can result due to SLB traps that occur when the SLB cannot provide a corresponding leading virtual address (LVA) for synonym access. This leads to misses in all the caches in the hierarchy. Further it requires not only TLB

lookups but also OS involvement to search for the corresponding LVA (not latency efficient). Accordingly, we analyze the percentage of cache accesses that result in an SLB trap for different SLB sizes in Figure 4.15. In this analysis, we take a checkpoint after the initial boot processes (including setting up a server or loading data) is over, and start the SLB simulation from this point. This initial phase has significant OS activity and results in many synonyms and thus a significant source of the total number of potential synonyms is left out in our analysis, thereby advantaging the SLB. Regardless, we see noticeable SLB traps for data cache access although a larger SLB is employed for some cases, e.g., 0.6% for *memcached* with 48 entries. This suggests that using a smaller SLB instead of a TLB may not be a viable solution. Such traps could potentially be further reduced by profiling and dynamically changing the LVA for virtual pages that are being frequently accessed at a given time during the program's execution. However, this entails even more OS involvement.

4.4 Chapter Summary

In this chapter, we propose and evaluate a L1 VIVT cache design, called ***VC-DSR: Virtual Cache with Dynamic Synonym Remapping*** to reduce latency and power/energy overheads of TLB lookups. Actually, these opportunities are well-known to processor vendors. A plethora of techniques have been proposed and deployed. However, they fall short of reducing both the latency and energy impacts of the TLB access. We believe that VC-DSR can be a practical way to limit all of the impacts as an address translation filter.

Why VC-DSR is a Compelling Solution? VC-DSR affords key aspects of practical virtual caching as follows, which facilitates the practical use of virtual caching not only for modern processors but also for future systems.

1. *Software transparent approach:* To handle synonym issues, some VIVT cache proposals require OS involvement (e.g., a single global virtual address space or modifications of

memory allocation). Doing so could restrict not only design flexibility of OSes but also programmability. However, as we demonstrate in the paper, VC-DSR achieves the goal without any OS involvements.

2. *Simple design:* VC-DSR requires less design complexity. We can easily integrate VC-DSR into any cache architectures regardless of their features such as writing policy, coherence protocol, etc. This allows the flexibility for target systems. Accordingly, we envision that the VC-DSR is readily applicable for future systems with different design features.

3. *Efficient synonym management:* Future computing systems could exploit more data sharing, which increases the likelihood of synonym accesses. However, we believe that temporal properties of synonyms continues to be pronounced in smaller caches. Accordingly, we expect that the SS and ART will be able to be efficient, scalable filters of active synonym accesses, which reduces potential overhead of synonyms.

In sum, satisfying these key aspects of virtual caching makes VC-DSR practical, which can be exploited to achieve the benefits of VIVT caches in a practical manner. Experimental results based on real world applications show that VC-DSR can achieve about 92% of the dynamic energy savings for TLB lookups, and 99.4% of the latency benefits of ideal (but impractical) virtual caches for the configurations considered.

Chapter 5

REDUCING TLB MISS OVERHEAD OF HETEROGENEOUS COMPUTING WITH VIRTUAL CACHE HIERARCHY

5.1 Introduction

In this chapter, to reduce the virtual address translation overhead on tightly-integrated CPU-GPU heterogeneous systems, we propose a **GPU virtual cache hierarchy** that caches data based on virtual addresses instead of physical addresses. *We employ the GPU multi-level cache hierarchy as an effective bandwidth filter of TLB misses,* alleviating the bottleneck of the shared translation hardware. We take advantage of the property that no address translation is needed when valid data resides in virtual caches [110]. We empirically observe that more than 60% of references that miss in the GPU TLBs find corresponding data in the cache hierarchy (i.e., private L1s and a shared L2 cache). Filtering out the private TLB misses leads to considerable performance benefits.

Virtual caching has been proposed several times to reduce the translation overheads on CPUs [13, 40, 57, 78, 88, 104, 109]. However, to the best of our knowledge, the efficacy of virtual caching on GPUs has not been evaluated, and it is not publicly known whether current GPU products use virtual caching. Furthermore, virtual caching has not been generally adopted, even for CPU architectures. The main impediment to using virtual caches for CPUs is virtual address synonyms. In Chapter 3, we empirically demonstrate that this synonym problem is exacerbated by larger caches and complex CPU workloads (e.g., server workloads). In larger caches, there is longer data residence time and thus a higher likelihood for synonymous accesses, which imposes more overhead for synonym detection and management to ensure correct operation. This complicates the deployment

of virtual caching for the entire cache hierarchy for CPUs.

However, we leverage the fact that GPUs are most commonly used as an accelerator, not a general-purpose processor. This fact greatly simplifies virtual cache implementation, as it reduces the likelihood of virtual cache complications (i.e., synonyms and homonyms). This easily enables the scope of GPU virtual caching to be extended to the whole cache hierarchy (private L1s and a shared L2 caches), and allows us to take more advantage of virtual caching as a bandwidth filter of TLB misses. Including the shared L2 cache in the virtual cache hierarchy filters more than double the number of TLB accesses compared to virtual L1 caches (31% with L1 virtual caches and 66% with both virtual L1 and L2 caches).

These observations lead to our virtual cache hierarchy design for the GPU. In our design, all of the GPU caches—the private L1s and the shared L2—are indexed and tagged by virtual addresses. We add a new structure (called a forward-backward table, FBT) to the I/O memory management unit (IOMMU) that is fully inclusive of the GPU caches. The FBT is used to ensure correct execution of virtual caches for virtual address synonyms, TLB shootdown, and cache coherence.

We show that using virtual caching for the entire GPU cache hierarchy (both L1 and L2 caches) can be practical from the perspective of performance as well as design complexity. The experimental results show an average speedup of 1.77× over a (practical) baseline system. The proposal also shows more than 30% additional performance benefits over the L1-only GPU virtual cache design.

The main contributions of this chapter are:

1. We analyze GPU memory access patterns and identify that a major source of GPU address translation overheads is the significant bandwidth demand for the shared TLB due to frequent private TLB misses. In addition, we show that (multi-level) virtual caches can filter out a significant number of the private TLB misses.

2. We integrate our virtual cache design that is described in Chapter 4 into GPUs. We present a practical and software transparent design of virtual caching for the entire

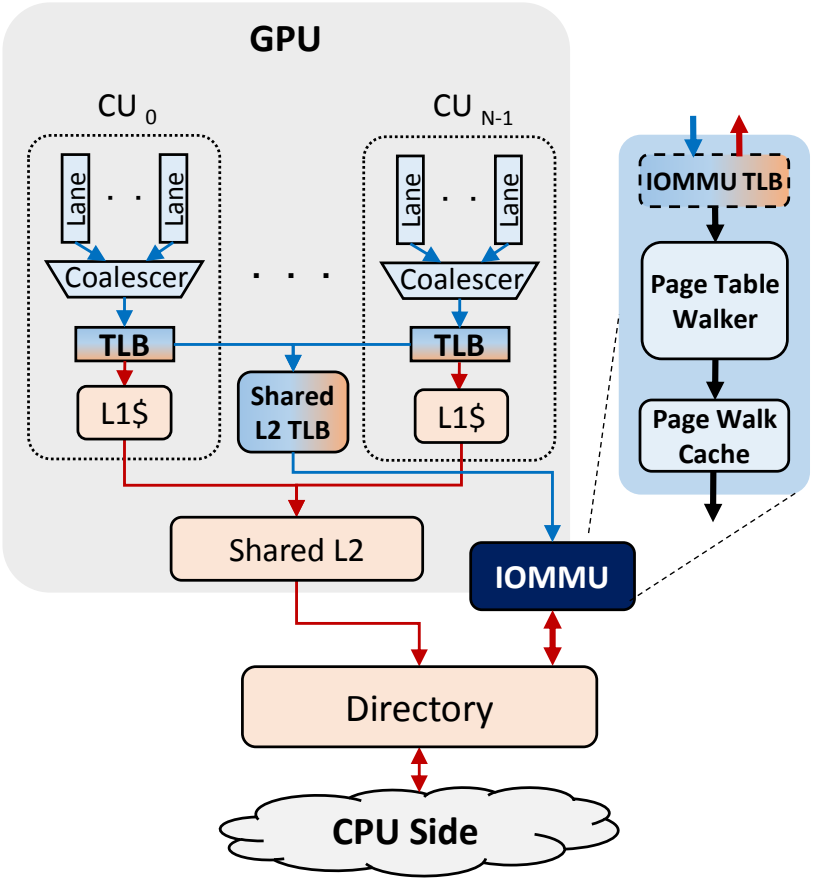


Figure 5.1: Overview of Baseline GPU MMU Design

GPU cache hierarchy. This virtual cache hierarchy filters most of the shared TLB accesses, which provides about 1.77× speedup, on average.

5.2 Background: GPU Address Translation

In this section, we first discuss the current GPU memory system design and the GPU address translation.

Figure 5.1 overviews the GPU memory system. we consider integrated CPUs and GPUs with fully unified (shared) address space support (e.g., Heterogeneous System Architecture (HSA) specification [62]). In current systems, each computational unit (CU) has a private TLB. The TLB is consulted after the per-lane accesses have been coalesced into the minimum number of memory requests; it is not consulted for scratch-pad memory

accesses. In addition to private TLBs, there is a large second-level TLB that is shared between all of the CUs. On a shared TLB miss, an address translation service request is sent to the IO Memory Management Unit (IOMMU) over the interconnection network. This interconnection network typically has a high latency. Even though integrated GPUs are not physically on the PCIe bus, IOMMU requests are still issued using the PCIe protocol, adding transfer latency to TLB miss requests [2].

The IOMMU consists of page table walker (PTW) and page walk cache; depending on microarchitecture designs, an additional IOMMU TLB may be employed for sharing across all I/O devices in a system. On a shared TLB miss, the PTW accesses the corresponding page table. The PTW is multi-threaded (e.g., supporting 16 concurrent page table walks) to reduce the high queuing delay due to frequent shared TLB misses [2, 85, 102]. Additionally, there is a page walk cache to decrease the latency of page table walks by leveraging the locality of page table accesses. Once the address translation is successfully performed (i.e., the corresponding PTE is found in the page table), a response message is sent back to the GPU. Otherwise, a GPU page fault occurs, and the exception is handled by a CPU.

5.3 Potential of Virtual Caching on GPUs

In this section, we show the potential of a GPU virtual cache hierarchy. We make the following four empirical observations:

- GPU workloads show high per-CU TLB miss ratio.
- They also show high per-CU TLB miss rates (i.e., significant bandwidth demand for the shared TLB).
- Many references that miss in per-CU TLBs find valid data in the GPU cache hierarchy (i.e., huge opportunity for caches to act as a TLB miss bandwidth filter).

CPU	1 core, 3GHz, 64KB D$ and 32KB I$, 2MB L2$
GPU	16 CUs, 32 lanes per CU, 700 MHz
L1 GPU Cache	per-CU 32KB, write-through no allocate
L2 GPU Cache	Shared 2MB, 8 banks, write-back, 128B lines, coherent with CPU
IOMMU	16 concurrent PTW, 8KB page-walk cache
DRAM	192 GB/s
Interconnect	Dance-hall topology within the GPU and Point-to-point network between the CPU-GPU

Table 5.1: Simulation Configuration

- Synonyms are less likely in GPUs than in CPUs since GPUs are used as an accelerator, not a general-purpose execution platform.

As we will establish below, these four key observations explain why a GPU virtual cache hierarchy is promising as a (private) TLB miss bandwidth filter. We first substantiate these characteristics and discuss their design implications.

5.3.1 Evaluation Methodology

To get the empirical data, we used a full-system heterogeneous simulator, gem5-gpu [84]. We use full-system simulation to ensure that we model all system-level virtual memory operations. We consider high-performance integrated CPUs and GPUs with fully unified shared address space and full coherence among GPU and CPU caches. Details of our simulation parameters are in Table 5.1.

As a baseline, we use a recent work proposed by Power *et al* [85]. For the address translation hardware, we use 32 entries for the per-CU TLBs, and a large shared L2 TLB in the GPU. We assume this shared TLB can process up to one request per cycle.[1] We use 16 page table walkers to handle misses from the shared TLB, which reflects current hardware design [102]. We also have an 8 KB physical cache for the page table walkers, as prior work

[1]Power *et al* [85] considered infinite bandwidth of the shared TLB, which is unrealistic.

found this is important for high-performance translation [85]. We assume the IOMMU TLB has the same translations as GPU TLBs and do not model this structure.

We evaluate workloads from two different benchmarks suites. The Rodinia workloads [24] represent traditional GPU workloads and are mostly scientific computing algorithms. We also use Pannotia [23] to evaluate emerging GPU workloads. The Pannotia workloads are graph-based and show less locality than traditional GPU workloads. The Pannotia benchmarks suite comes with multiple versions of algorithms. We present data for each version of the algorithm separately. We run all workloads to completion. They execute from about 2 million to 2 billion cycles.

Observation 1: *GPUs show high per-CU TLB miss ratio.*

Figure 5.2 presents per-CU TLB miss ratio for 4KB pages for each benchmark by varying the TLB size. The results for the infinite size of TLBs indicate demand per-CU TLB misses.[2] The total height of each bar shows the average miss ratio for the entire execution of the application. We discuss the breakdown of each bar in Observation 3, below.

We notice that per-CU TLB miss ratio is very high, *much* higher than that of typical CPU workloads. Figure 5.2 shows about 56% (average) of misses with a 32-entry TLB per CU, and it can be above 80% for some workloads such as `pagerank`, `pagerank_spmv`, and `nw`. Even with large TLBs of 64 and 128 entries, we see frequent per-CU TLB misses.

These workloads have high TLB miss ratios for two reasons. First, GPUs are highly multi-threaded with up to 10's of unique execution streams per CU. Each of these streams may be accessing different pages in memory, limiting the spatial and temporal locality. Second, in addition to the many execution streams, each execution stream may result in multiple memory accesses per instruction due to memory divergence. For instance, `fw`

[2] "Demand misses", in this case, include all accesses that experience a non-zero translation latency when using infinite capacity per-CU TLBs. This includes multiple TLB misses to the same page in close temporal proximity as our TLB model does not combine multiple misses to the same page into a single request. Thus, we see demand misses that hit in L1 and L2 caches. In addition, some demand per-CU TLB misses hit in the shared L2 cache as data is shared between CUs. pathfinder experiences many such requests.

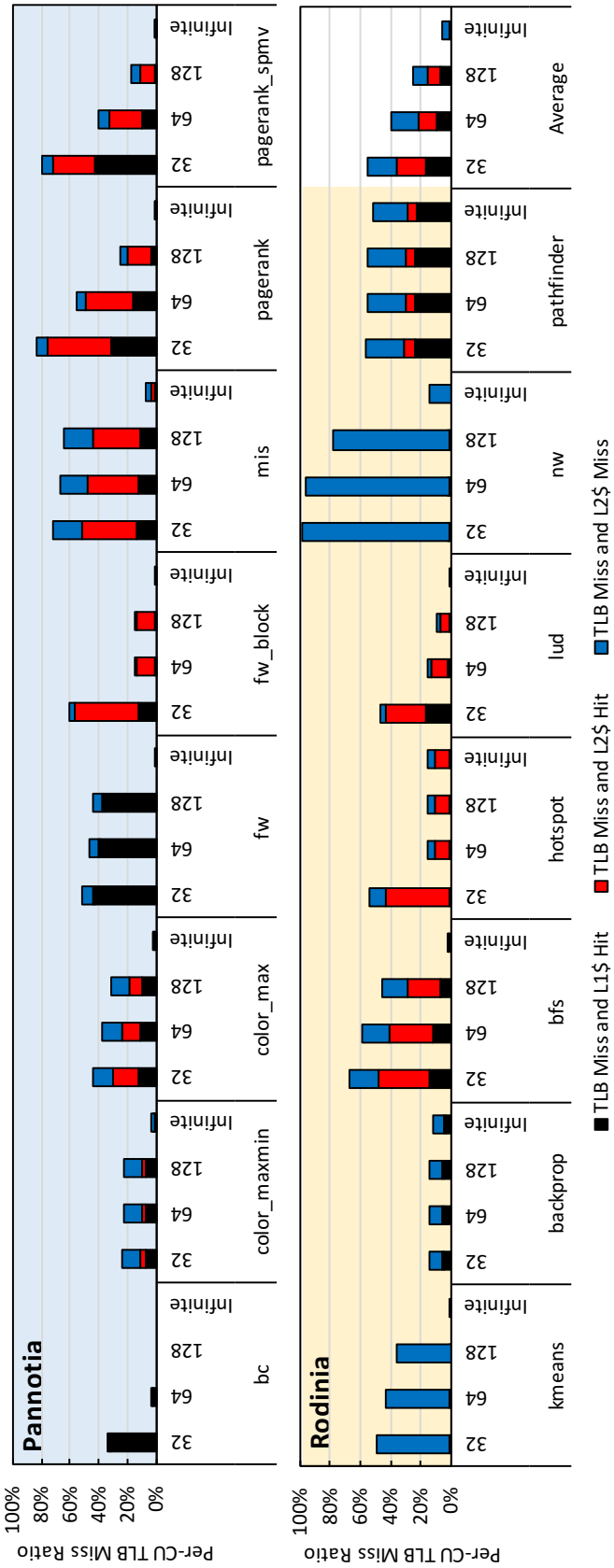


Figure 5.2: Breakdown of Per-CU TLB Miss Accesses

averages 9.3 memory accesses per dynamic memory instruction. Our observations are in line with the results of a recent study on real hardware [102].

Design Implication: *The performance impact of the address translation is much higher for GPUs than CPUs.*

Observation 2: *GPU workloads show a high rate of both shared TLB accesses and misses.*

Figure 5.3 presents an average of events per one nanosecond for shared TLB accesses (i.e., per-CU TLB misses for all CUs) and shared TLB misses. We used periodic sampling to obtain this data. The results indicate the average of events across all sampling periods, and each bar has a one standard deviation band. For the results, 32-entry per-CU TLBs and a 512-entry shared TLB are used. The experiment for this figure assumes the shared TLB can be accessed any number of times per cycle, which is impractical.

We notice both frequent shared TLB accesses and frequent shared TLB misses. There are about 0.7 shared TLB accesses and 0.19 misses per nanosecond, on average. Some workloads (e.g., `color_max`, `fw`, `mis`, `pagerank`, `bfs`, and `lud`) show numerous sample periods with more than one shared TLB access per nanosecond. For example, `color_max` shows about 20% of sample periods with more than one shared TLB accesses per nanosecond. We also observe that, emerging GPU workloads, especially graph-based workloads like Pannotia, show much more frequent per-CU TLB misses than the traditional workloads because of high memory divergence [23]. The results show there are impractical bandwidth requirements at the shared TLB and the PTW (more than two accesses per nanosecond in some cases).

Figure 5.4 breaks down the performance overhead of address translation on the GPU, compared to an "IDEAL MMU" that has infinite bandwidth at the shared TLB and infinite capacity per-CU TLBs. This figure presents only the average performance across all of the workloads evaluated. For a practical design that has 32-entry private TLBs and a 512-entry

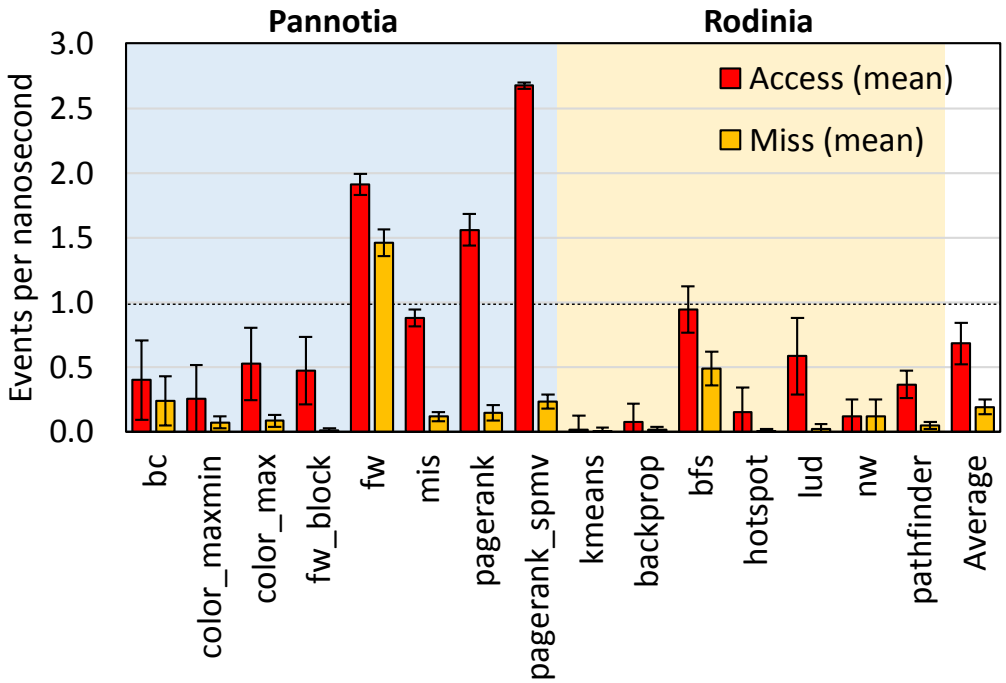


Figure 5.3: Analysis of Shared TLB Accesses/Misses

shared TLB (i.e., "BASELINE" bar), we see an average of 77% performance overhead over the "IDEAL MMU".

This overhead can be broken down into the overhead for page table walks (PTW) due to shared TLB misses, and the overhead due to serialization from limited bandwidth for the shared TLB. For the "PTW Overhead", we consider a system with an infinite capacity shared TLB, but the shared TLB bandwidth is limited to one access per cycle (0.7GHz). With an infinite capacity shared TLB, each per-CU TLB miss is immediately translated by the shared TLB, removing the PTW overhead. We can see that **performance benefits with infinite capacity of the shared TLB are small**, because the multi-threaded page table walker with a large page walk cache effectively hides shared TLB miss latency [85]. For the "Serialization Overhead", we consider a baseline system with infinite bandwidth for the shared TLB. We see that **most of the address translation overheads are due to serialization at the shared TLB, not the size of the shared TLB**.

The results suggest that the primary challenge of GPU address translation is to provide

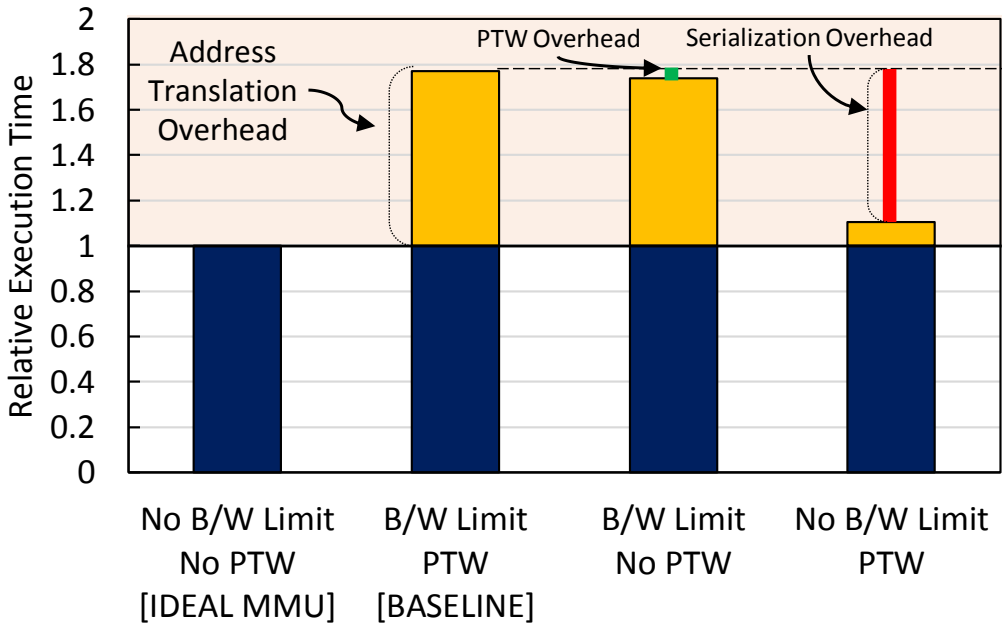


Figure 5.4: Analysis of GPU Address Translation Overhead

high bandwidth at the shared TLB. However, it is impractical to use very large per-CU TLBs or to use highly multi-ported large shared TLB. Also, it is unrealistic to provide/manage huge buffers between the CUs and the shared TLB due to frequent shared TLB accesses.

Multi-banking the shared TLB, while costly in terms of complex pipeline and arbitration logic, could increase the bandwidth if bank conflicts are rare [9, 89]. However, bank conflicts may be more common on a banked TLB than a banked cache due to using higher order address bits for bank mapping. In fact, some of our workloads (e.g., `mis`, `color_max`, etc.) show frequent conflicts when analyzing the address stream to the shared TLB. For a 16-bank shared TLB, about 50% of shared TLB accesses see more than 4 prior requests mapped to the same bank in most recent 16 prior accesses for these workloads. The frequent bank conflicts limit the bandwidth of multi-banked designs.

The bandwidth demands of the shared TLB will continue to increase as future GPUs integrate more CUs. The recently released Playstation 4 Pro console has 36 CUs, and XBOX Scorpio is speculated to have even more 60 CUs [4]. Furthermore, we believe that the overall address translation overheads will increase with emerging applications [12, 23]. Thus,

we cannot depend solely on unscalable conventional address translation optimizations, e.g., multi-banking, large per-CU TLBs (also evaluated in Figure 5.10), and large pages (Section 5.4.3). **Accordingly, we need a scalable, efficient way of filtering out the accesses to the shared TLB.**

Design Implication: *Filtering traffic to the shared TLB will improve performance.*

Observation 3: *Data resides in the caches much longer than its translation in the TLB.*

We now consider how many per-CU TLB misses can be filtered with a virtual cache hierarchy on a GPU. Figure 5.2 shows the breakdown of the TLB misses according to where the valid data is located in the GPU cache hierarchy. The results indicate that many references that miss in the per-CU TLB hit in the caches (black and red bars) and that only 34% of references that miss in the per-CU TLB are also L2 cache misses and access main memory (blue bars).

Many references that miss in per-CU TLBs actually hit in the caches. We notice that an average of 31% of total per-CU TLB misses find the corresponding data in private L1 caches (black bars), and an additional 35% of the total misses hit in a shared L2 virtual cache (red bars). These hits occur for two reasons. First, for L2 hits under per-CU TLB misses, data may be shared by multiple CUs. There are requests that miss in the per-CU TLB that hit in the shared L2 cache when other CUs have previously accessed the cache block. Second, for both L1 and L2 hits under per-CU TLB misses, blocks in the cache hierarchy are likely to reside longer than the lifetime of the corresponding per-CU TLB entries.

Figure 5.5 shows the relative lifetime of pages in each level of the cache hierarchy and the TLB. It shows three CDFs of the aspects for `bfs` workload; other workloads show similar patterns. The black line indicates the residence time of TLB entries. The other two lines show the active lifetime of data in L1 caches (blue line) and L2 shared cache (red line), respectively; the active lifetime is defined as the period between when data is cached

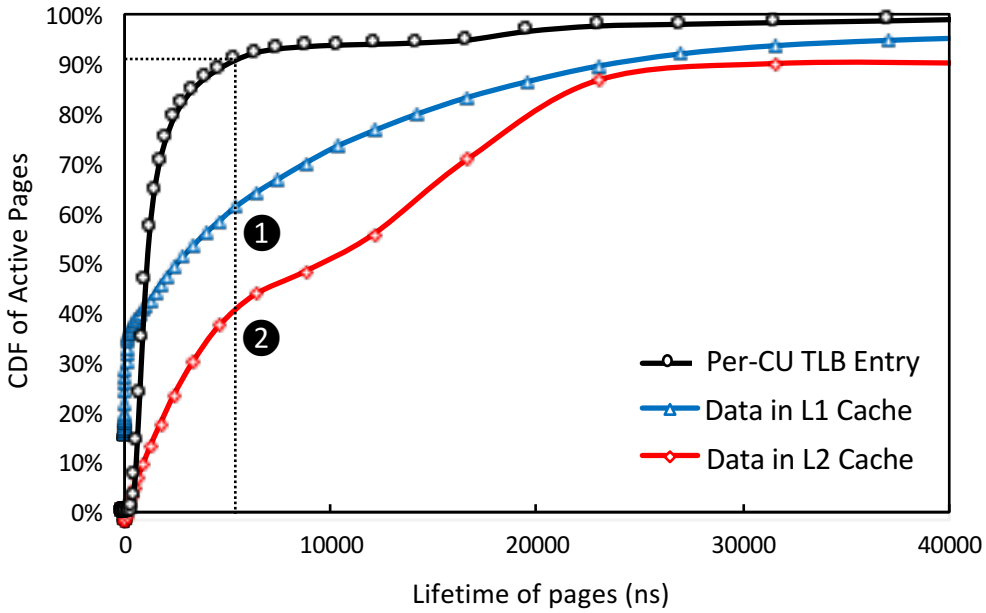


Figure 5.5: CDF Analysis of Active Pages: *bfs* Workload

and when it is last accessed. We notice that 90% of TLB entries are evicted after 5000 ns. However, 40% of data in L1 caches (①) and 60% of data in a shared L2 cache (②) are still actively used after that. Thus, accesses to such data hit in the cache hierarchy but are highly likely to occur misses in the TLB. In these cases, a virtual cache hierarchy is an effective TLB miss bandwidth filter.

Also there is a noticeable gap between two lines for caches, suggesting that data not in L1 caches is frequently found in a larger L2 cache. This supports what we observed in Figure 5.2; the red bar is larger than the black bar for many workloads such as `color_max`, `fw_block`, `mis`, `pagerank`, `pagerank_spmv`, `bfs`, `hotspot`, `lud`. Thus, extending the scope of virtual caches to the L2 shared cache helps to filter out more TLB misses.

Design Implication: *A large fraction of TLB misses can be filtered by employing deeper virtual cache hierarchy (i.e., including a shared L2 cache) for GPUs.*

Observation 4: ***GPU's accelerator nature makes synonyms much less likely than in CPUs.***

We observed a larger fraction of cache accesses to cache lines with synonyms when larger virtual caches are considered in the CPU cache hierarchy (Section 3.3). However, GPUs are not fully general-purpose compute devices. There are three key differences between CPUs and GPUs which lessen the impact of synonyms in a GPU cache hierarchy.

First, as an accelerator, *current* GPUs usually execute a single application at a time. Thus, there is usually one active virtual address space. Second, GPUs rarely access I/O devices. This is likely to continue to be true since GPUs are useful for applications with data parallelism, which I/O device drivers rarely exhibit. Third, GPUs never execute OS kernel code; they only execute user-level applications. This eliminates most causes of synonym accesses, making the occurrence of active synonym accesses less likely in a GPU cache hierarchy than in a CPU cache hierarchy. For our workloads, we never observed synonyms. We discuss the implications on future system running multiple processes in Section 5.4.3.

Design Implication: *GPU virtual caches can be practical even for large second-level cache.*

5.4 Design of GPU Virtual Cache Hierarchy

We first set two key requirements to design an efficient GPU virtual cache hierarchy:

1. The GPU virtual cache hierarchy needs to be software transparent. The proposed virtual cache design should be able to efficiently deal with potential issues from virtual memory idiosyncrasies such as potential virtual address synonyms, TLB shootdowns, etc.

2. The proposed design should be seamlessly integrated into a modern GPU cache hierarchy, which is somewhat different from a traditional CPU cache hierarchy.

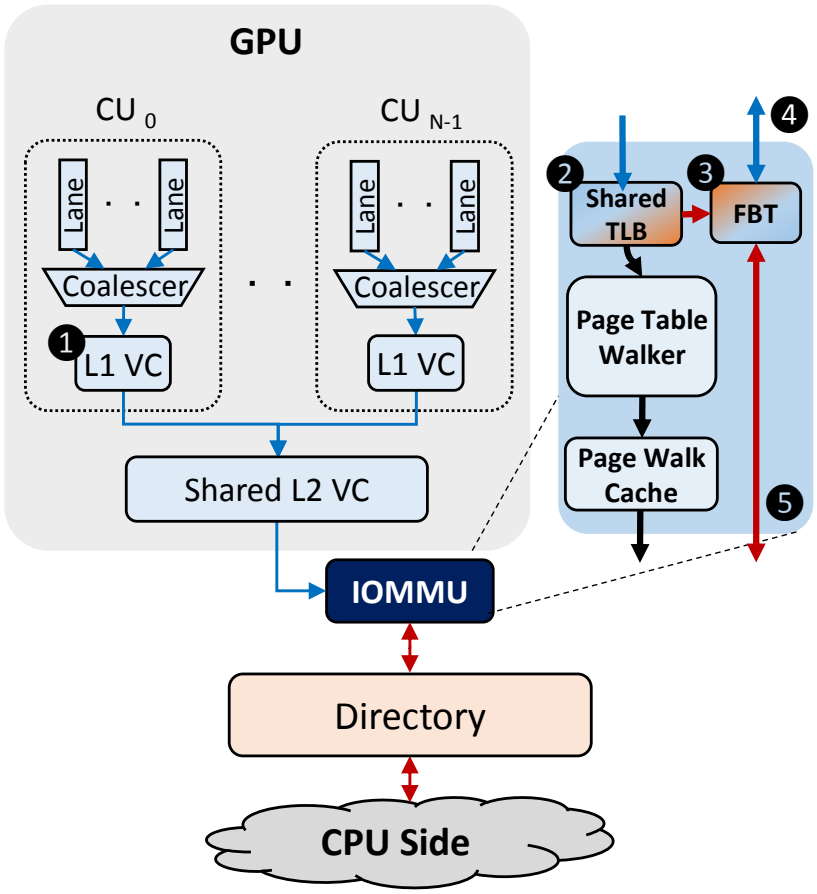


Figure 5.6: Schematic of Proposed GPU Virtual Cache Hierarchy

Figure 5.6 illustrates the conceptual design of the proposed virtual cache hierarchy. Different from the baseline system (Figure 5.1), there are no per-CU TLBs or shared L2 TLB in the GPU, and the L1 and L2 caches at the GPU side are accessed with virtual addresses. Address translation is performed via a shared TLB in the IOMMU only when no corresponding data is found in the virtual caches. That is, the address translation point is completely decoupled from the GPU cache hierarchy.

To correctly and efficiently support virtual memory, we add a *forward-backward table* (FBT) to the IOMMU. The FBT consists of two parts: *backward table* and *forward table* (see Figure 5.7). First, the backward table (BT) is primarily a reverse translation table which provides reverse translations from physical addresses to virtual addresses. We take advantage of important features of an active synonym detection table (ASDT) from the virtual

cache proposal for CPUs (refer to Section 4.2). The backward table is fully inclusive of the virtual caches with an entry for every page that is currently cached. The BT tracks mappings from a physical address to a *unique* leading virtual address that is used to place the data in the virtual caches at a page level granularity. We use this information to carry out operations of virtual caches such as detection of synonymous accesses and handling TLB shootdowns. Also, the BT is employed for reverse address translations on coherence requests from outside the virtual cache using physical addresses.

The FBT also contains a small forward table (FT). The FT tracks mappings from a leading virtual address to an index of a corresponding backward table entry, which allows the FBT to be indexed by virtual address as well as physical address. Forward translation information is needed for several operations including when the virtual caches respond to coherence requests which use physical addresses, on virtual cache evictions, and TLB shootdowns. As we shall see, the extra index structure of the FT allows us to perform these operations without a shared TLB miss and subsequent page table walk. Additionally, with the forward translation information, the FBT can be used as a large second-level TLB structure.

In the following sections, we explain how the proposed design satisfies the two key requirements (Section 5.4.1-5.4.2). Then, Section 5.4.3 describes several design choices to improve the effectiveness not only of our proposal but also of the overall heterogeneous system by taking advantage of the forward-backward table.

5.4.1 Supporting Virtual Memory without OS Involvement

To handle synonym issues, some virtual cache proposals [13, 21, 22, 60, 78, 88, 110] require OS involvement (e.g., a single global virtual address space). Doing so restricts not only design flexibility of OSes but also programmability. Hence, we believe that virtual cache hierarchy for a GPU needs to be a *software transparent technique*.

Our proposal supports virtual memory requirements without any OS involvement.

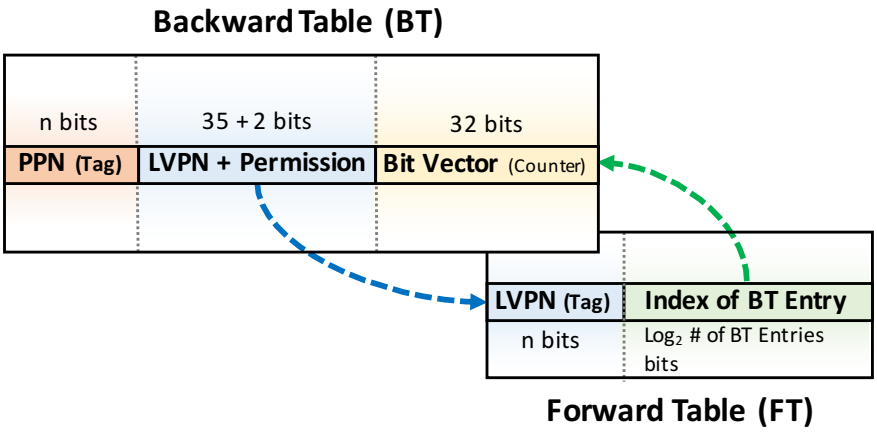


Figure 5.7: Overview of a Forward-Backward Table (FBT)

The proposal allows a GPU to access any virtual address that can be accessed by a CPU. We also effectively and transparently manage the other operations (e.g., TLB shootdown, synonyms, homonyms, etc.).

Below we describe the details of operations and the extended structures to support virtual memory. For the sake of simplicity, we initially assume a system with an inclusive two-level virtual cache hierarchy. In Section 5.4.2, we will discuss some challenges and solutions for modern GPU cache designs that support a non-inclusive, two-level cache hierarchy.

Virtual Cache Access: For memory requests, a set of *lanes* (called shader processors or CUDA cores by NVIDIA) generate virtual addresses. After the coalescer, memory requests are sent to the L1 virtual cache without accessing a TLB (① in Figure 5.6). Different from a physical cache design, the permission bits of a corresponding *leading* virtual page are maintained with each cache line due to deferred TLB lookups, and the permission check is performed when the virtual cache is accessed.

On an L1 virtual cache miss, the L2 virtual cache is accessed with the given virtual address. On a hit in the virtual cache hierarchy, the valid data is provided to the corresponding CU as long as the permissions match. Thus, no address translation overhead is imposed.

Address Translation: On an L2 virtual cache miss, the request is sent to the IOMMU. The virtual to physical address translation is performed via the shared TLB (② in Figure 5.6). The translation is required because the rest of system is indexed with physical addresses. The permission check for the cache miss is also performed at this point by consulting the shared TLB.

Interestingly, we observe that, for most of shared TLB misses (e.g., 74% on average), a matching entry is found in the FBT.[3] Thus, the FBT can be a second-level TLB. We can search for the match in the FBT by consulting the FT on the misses. If there is no matching address translation, a page table walker (PTW) executes the corresponding page table walk. To reduce the long latency of consulting multiple DRAM accesses, a page walk cache that caches non-leaf nodes of the corresponding page table is accessed first. If the PTW fails to find a matching PTE, a page fault exception occurs; this exception is handled by the CPU. These actions taken on a shared TLB and FBT miss are the same as the baseline IOMMU design.

Synonym Detection and Management: To guarantee the correctness of a program, we need to check whether the cache miss occurs due to a virtual address synonym. Multiple virtual addresses (i.e., synonyms) can be mapped to the same physical address. Hence, it is possible that valid data has been cached with a different virtual address, and in this case, the data cached with the other virtual address should be provided.

To check for synonyms, the BT is consulted with the physical page number (PPN) (③ in Figure 5.6) obtained by the shared TLB lookup or the PTW. The BT is a set-associative cache (Figure 5.7). Each entry has a physical page number (PPN) as a tag and a unique (leading) virtual page number (LVPN). *Only this current leading virtual address is allowed to place and look up data from the physical page in the virtual caches as long as the entry is valid*. Thus, no data duplications are allowed in the virtual cache hierarchy.

If a valid BT entry is identified for the cache miss, it indicates that some data from the

[3]The FBT entries correspond to all of physical pages with cached data (private L1s and a shared L2 cache) We consider an over-provisioned FBT with 16K entries, which has the reach of 64MB (Section 5.4.3).

physical page resides in the virtual cache, and has been cached with the corresponding leading virtual address. Thus, if the given virtual address for the cache miss is different from the current leading virtual address, it is a synonym access. To ensure the correct data, we simply replay the virtual cache access with the current leading virtual address (④ in Figure 5.6). Each BT entry has a bit vector indicating which lines from a physical page are cached in the virtual caches. Thus, only addresses that will hit are replayed. If the bit in the bit vector is clear, the directory is accessed. Once the valid data is received from the memory, it is cached with the current leading virtual address, and its permission bits.

For L2 virtual cache misses, if there is no match in the BT, a new entry is created for the mapping between its PPN and the given virtual page, and the given virtual page will be the leading virtual page for the physical page until it is evicted. At the same time, a corresponding FT entry is populated.

Modern GPUs are unlikely to access virtual address synonyms because they do not run OS kernel codes and usually only execute a code from a single process (refer to Observation 4 in Section 5.3), which makes our design amenable without the dynamic synonym remapping via an Address Remapping Table (ART). However, future GPU systems are likely to have more multiprogramming support like modern CPUs. Regarding this, we discuss the design aspects in Section 5.4.3.

Cache Coherence between a GPU and a CPU: Cache coherence requests from a directory or CPU caches use physical addresses. Hence, a reverse translation (i.e., a physical address to a current leading virtual address) is performed via the BT (⑤ in Figure 5.6). Then, the request is forwarded to corresponding GPU caches with the leading virtual address. When the cache responds with a leading virtual address, it is translated to the matching physical address via the FT.

Including the BT in the IOMMU has the benefit of providing an efficient coherence filter for the GPU caches. Power *et al*. proposed using region-based coherence to filter most coherence messages sent to the GPU [83]. The BT is fully inclusive of the GPU caches.

Therefore, when a coherence request is sent to the GPU, the BT can filter any requests to lines that are not cached in the GPU caches. The BT plays a similar role to the region buffer in the heterogeneous system coherence protocol [83].

TLB Shootdown and Eviction of FBT Entry: If information of a leading virtual page changes (e.g., permission or page mapping), the leading virtual page is no longer valid. Thus, the corresponding FBT entry should be invalidated. This leads to invalidations of data cached with the leading virtual address from all virtual caches for correct operations; without the invalidations, cache accesses may not respect the up-to-date permission information in the page table or may obtain stale data based on the previous mapping. Similarly, when an FBT entry is evicted (e.g., due to a conflict miss), the same operations need to be performed. By employing the information of a bit vector, only the cached data can be selectively evicted. This reduces invalidation traffic.

On a single-entry TLB shootdown, we use the FT with virtual addresses. Once a match is found, we directly access the corresponding BT entry with the index pointer. The FT filters TLB invalidation requests if no match is found. On an all-entry TLB shootdown, a cache flush is required.[4] The shootdowns resulting from page mapping or permission changes are infrequent [13]. Thus, the performance impact is likely minor.

Eviction of Virtual Cache Lines: When a line is evicted from the virtual cache, the bit vector information of the corresponding BT entry needs to be updated in order for the BT to maintain up-to-date inclusive information of the virtual cache. On every cache line eviction, the FT is consulted to identify the corresponding BT entry. The matching bit in the bit vector of the BT entry is unset.

[4]For swapping pages, some OSes flush whole caches (or selectively invalidate cache lines of the relevant pages depending on the ranges of target memory region). Thus, we do not have to additionally flush all caches again.

5.4.2 Integration with Modern GPU Cache Hierarchy

We now discuss several other design aspects to be considered for a practical GPU virtual cache hierarchy.

Read-Write Synonyms: With a virtual cache hierarchy, it is possible for the sequential semantics of a program to be violated if there is a proximate read-write synonym access [88, 109]. For example, a load may not identify the corresponding older store since their virtual addresses differ, even though their corresponding physical address is the same. This load could get the stale value from a virtual cache earlier than the completion of the store.

For CPUs, this violation is detected with a leading virtual address in the load/store unit and recoverable as they provide support for precise exceptions and a recovery mechanism (refer to Section 4.2.5). However, GPUs support relaxed consistency memory model without precise exceptions or precise recovery mechanisms [46, 58, 70]. Thus, resources for younger memory operations in the load/store unit can be deallocated before older memory operations are complete [94]. It could be too late to identify the violating instruction from the memory pipeline.

In practice, we believe that read-write synonyms will rarely cause problems in a GPU virtual cache hierarchy due to the following reasons:

1. GPUs do not execute any OS kernel operations that are the major source of read-write synonym accesses.

2. Although it is possible to have user-mode read-write synonyms, there are very few situations where they are used.

3. Even with read-write synonyms, correctness issues occur only if the aliases are in close temporal proximity.

Thus, we use a simple and viable solution and conservatively cause a fault when a read-write synonym access is detected at the FBT.[5] Although our design simply detects

[5]Read-write synonym accesses are detected at the BT. Since the FBT forwards all coherence requests

read-write synonyms and raises an error, if future GPU hardware supports recovery, it is possible to detect and recover from the violation of sequential semantics. We can use a similar mechanism discussed in Section 4.2.5; we can replay the violating synonymous access with the leading virtual address obtained from the FBT. Given GPUs move toward general-purpose computing, it is possible that recovery and replay hardware may be included in future GPU architectures [70]. Regardless of the issue of read-write synonyms, we fully support the much more common read-only synonym accesses.

Multi-Level Cache with Non-Inclusion Property: In the previous section, we conceptually explain the operations with a two-level inclusive cache hierarchy. In practice, however, modern GPUs employ a non-inclusive hierarchy. Furthermore, the designs of modern GPU cache hierarchy show different features compared to that of CPUs with respect to writing policy, inclusion, and coherence protocols [25, 95].

The baseline GPU cache hierarchy has write-through L1 caches without write-data allocation. The shared L2 cache is non-inclusive, and it supports write-back with write-data allocation. On an L2 cache eviction, invalidations of L1 copies are not required. Thus, it is possible for private L1 caches to have data that does not reside in the shared L2 cache. This complicates tracking inclusive information of data currently residing in the private L1 virtual caches in the BT. A naive solution is that private L1 caches send messages to the BT in the IOMMU on every cache line refill and eviction to keep track of the precise information; we may need to count the number of data copies in a virtual cache hierarchy with a counter per line, instead of simply using a bit vector (a bit per line) in each BT entry. This is not practical because of potential latency, energy and bandwidth overhead.

In practice, modern GPU L1 caches are not coherent, and thus we do not have to track the precise information. To keep the private L1 caches consistent, we take a conservative approach. The BT tracks the inclusive information of data currently cached only in the shared L2 cache with bit vectors. When a leading virtual page is no longer valid (e.g., an

from a GPU to a directory, we track whether any writes have occurred to a cached physical page. The fault is raised when there is a synonymous access to a physical page that has previously been written, and vice versa.

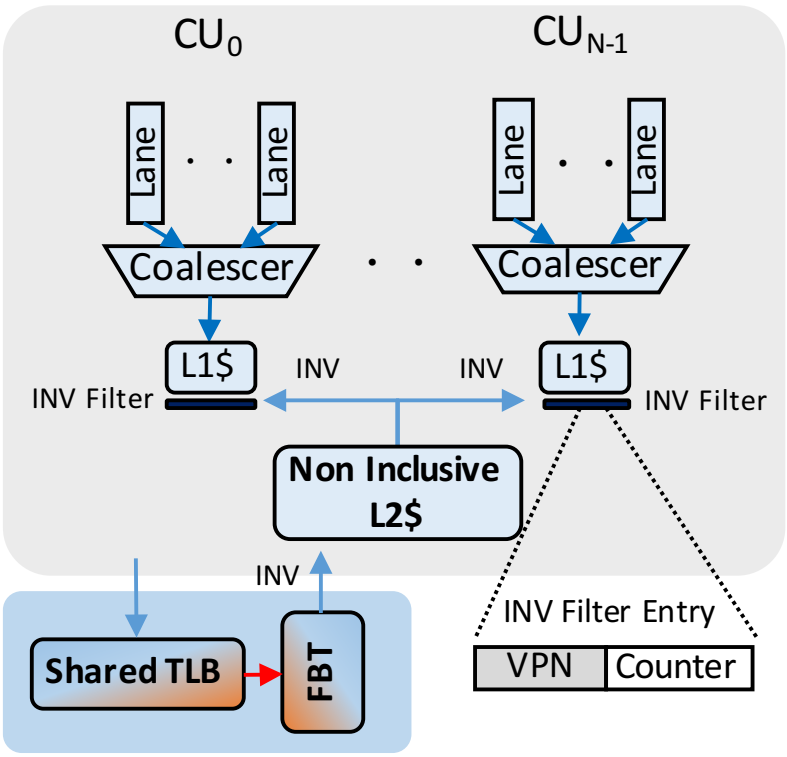


Figure 5.8: Supporting Non-Inclusive Cache Hierarchy

eviction of FBT entry or TLB shootdown), however, an invalidation message is sent to all L1 caches to invalidate all lines with a matching leading virtual page address. To avoid walking the L1 cache tags, we add a small invalidation filter at the L1 caches (shown in Figure 5.8). Each entry has the leading virtual page number as a tag and a counter tracking how many lines from the corresponding physical page currently reside in the cache. If a match is found in a filter for the invalidation request, the entire L1 cache is invalidated.

This approach does not require write backs since L1 caches have no dirty data[6] (i.e., write-through without allocation). Additionally, this approach has only a minor impact in terms of performance. This is because (i) GPU L1 cache hit ratio is usually low (less than 60% for most of our workloads), and (ii) the events triggering the flushes are highly unlikely, as TLB shootdowns are rare. Additionally, it is likely that the corresponding data are already evicted from L1 virtual caches at the point of an eviction of a corresponding

[6]For that reason, flushing L1 caches can be easily supported by the cache controller without affecting the baseline coherence protocol (e.g., state transitions).

FBT entry with an adequately provisioned FBT. Thus, the invalidation filter can filter the invalidation requests due to most FBT evictions.

5.4.3 Other Design Aspects of Proposed Design

In this section, we discuss other design aspects of the proposed virtual cache hierarchy.

Future GPU System Support: Future GPU systems will have more multiprocess support like modern CPUs, and *synonyms and homonyms* may be more common. To mitigate the overhead of handling synonym requests further, the concepts of *dynamic synonym remapping* can be easily integrated to the proposed GPU virtual cache hierarchy. That is, like our CPU virtual cache proposal, we can simply consult the synonym signature (SS) and address remapping table (ART) prior to GPU L1 VC lookups. For active synonym accesses, that is, a remapping from a non-leading virtual address to the corresponding leading virtual address can be performed prior to L1 virtual cache lookups. This reduces virtual cache misses and the latency/power overheads due to synonyms. *Homonym* issues can be easily managed by including address space identifier (ASID) information in a given virtual address; each cache line also need to track the corresponding ASID information. This prevents cache flushes on context switches.

Large Page Support: Using large pages is an effective way of reducing TLB miss overhead [82, 85, 102]. However, it has its own constraints and challenges [12, 74]. Increasing memory footprints will put additional pressure even on large page structures, requiring OSes and TLB organization modifications. In addition, large pages do not help workloads with poor locality (e.g., random access). Thus, we cannot depend on only large pages, but we need to support them to maintain compatibility with the conventional system.

In our design, supporting larger pages does not cause any correctness issues. However, it is impractical to use a large bit vector for each BT entry[7] to track lines from a large page that are resident in the cache hierarchy. In fact, the use of a bit vector is just an optimization

[7]For example, 160 bits (20B) are required for a 2MB page with 128B line size. It takes about 72% of the size of a BT entry while a bit vector for a 4KB page takes about 34% of that.

to selectively identify (or invalidate) lines of a physical page from virtual caches. Instead of maintaining the precise information, we can simply use a counter for the number of cached lines from a physical page. In this case, multiple invalidations could be sent until no more lines remain in the caches (i.e., the corresponding counter becomes zero). We can reduce the chances of the costly operations by deferring evicting FBT entries for a large page as much as possible until the counter becomes zero.

Area Requirements: To support the proposed virtual cache hierarchy, each cache line entry needs to be extended to track some extra information (e.g., virtual tag bits, permission bits), and we also need to support additional structures, i.e., invalidation filters for non-inclusive cache hierarchy and an FBT.

The size of the per private L1 invalidation filter is modest, relative to a private L1 cache. For instance, a 32KB L1 cache with 128B lines requires 1KB storage, which is less than 3% of the L1 cache size.

The FBT should be sufficiently provisioned to avoid potential overhead of cache line invalidations due to FBT entry evictions. We observe that there are about 6000 different 4KB pages whose data reside in a 2MB L2 cache on average for our workloads, and few workloads show more than 12K at maximum. We use an over-provisioned BT structure with 16K entries which requires about 190KB, and the relevant FT requires 80KB, totaling 270KB. This is about 7.5% overhead to the total GPU cache hierarchy. In practice, however, an adequately provisioned structure (e.g., with 8K entries) can mostly eliminate the invalidation overhead for our workloads.

Security Filter: Since the CUs are not allowed to issue physical addresses, a virtual cache hierarchy provides the same security benefits as a border control unit [75]. Similar to the border control unit, the BT tracks the permission information of each page cached by the GPUs. Additionally, every GPU memory access is checked at the IOMMU to ensure it is an allowed request. This protects the system from potentially buggy GPU hardware and buggy GPU applications.

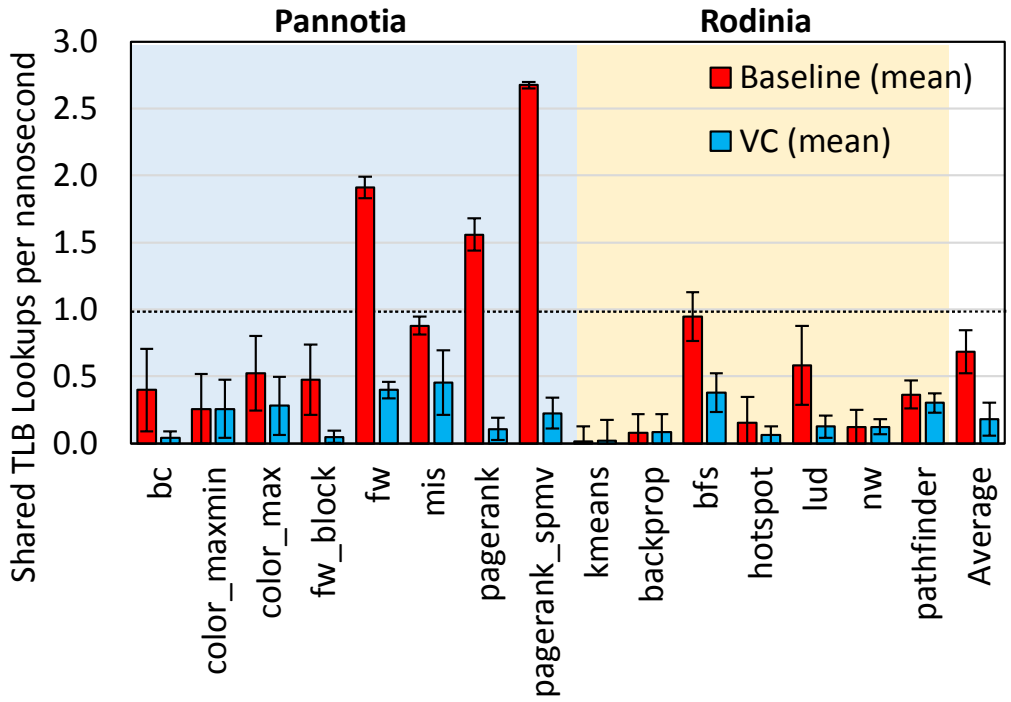


Figure 5.9: Bandwidth Requirements of shared TLB Lookups

5.5 Evaluation

Our experimental evaluation presents the effectiveness of the proposed GPU virtual cache hierarchy. We run both CPU and GPU parts in the full system mode of gem5-gpu [84]. We described the details of the evaluation methodology and a set of simulated workloads earlier in Section 5.3.1. As a baseline, we choose a 512-entry shared L2 TLB for simulation purposes. A larger TLB would fit the entire working sets of some of our workloads, which is unrealistic for future integrated CPU-GPU systems with potentially 100's of GBs of DRAM via expandable DIMMs. We model 10 cycle interconnect latency between a GPU L2 cache and FBT, and 5 cycles for FBT lookups. Since we are focusing on the performance of the GPU address translation, we only report the time that the application executes on the GPU. We analyze the filtering effects of our proposal in Section 5.5.1. Then we evaluate the performance benefits in Section 5.5.2. Finally, we will discuss why a virtual L1 only cache design (i.e., L2 cache is physically addressed) is less effective in terms of performance and design complexity.

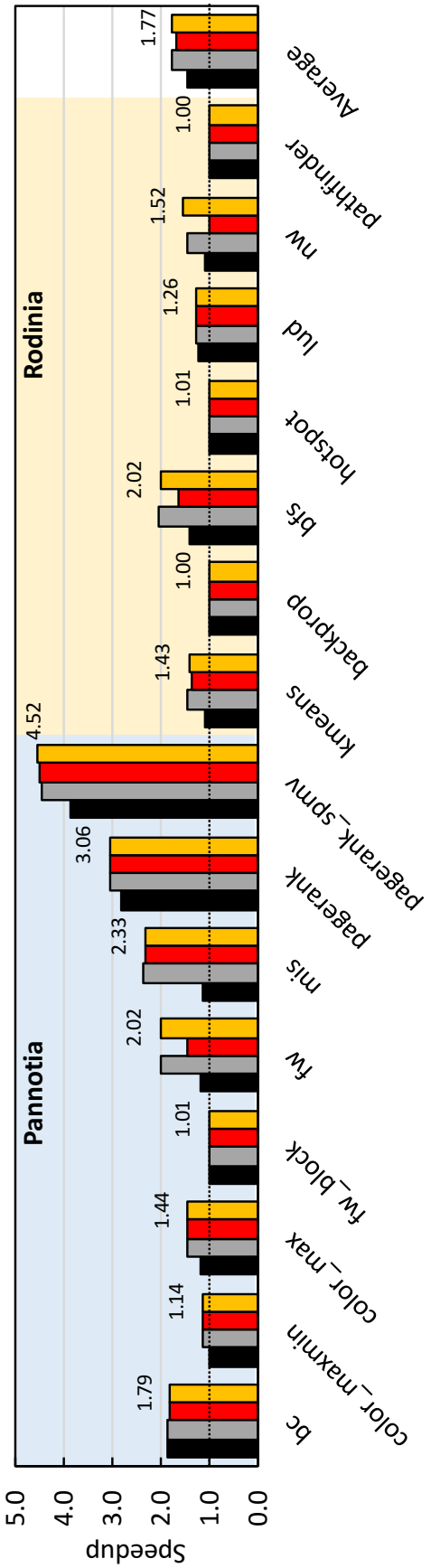


Figure 5.10: Performance Analysis: Speedup relative to 32-entry per-CU TLB (higher than 1.0 is better)

5.5.1 Virtual Cache Hierarchy's Filtering

As discussed in Section 5.3, a major source of GPU address translation overheads is the significant serialization at the shared TLB ("Serialization Overhead" bar in Figure 5.4) due to high per-CU TLB miss rate (Figure 5.3). Accordingly, the performance benefits will be directly affected by how effectively the GPU virtual cache hierarchy filters out the shared TLB accesses. Figure 5.9 presents an average of shared TLB lookups per one nanosecond for the baseline and for our proposal, respectively. Each bar has a one standard deviation band for all sampling periods. The red bars for the baseline in this figure correspond to red bars in Figure 5.3

For most of workloads, we notice significant reductions compared to the baseline system. We observe less than 0.2 events per one nanosecond on average with our proposal. Some workloads show slightly more than one event per one nanosecond, however, they are rare events (e.g., less than 0.5% of sample periods). The results suggest that the virtual cache hierarchy is effective in reducing the load on the shared TLB causing a significant serialization overhead.

5.5.2 Execution Time Benefits

Figure 5.10 shows relative speedup compared to the baseline design with 32-entry per-CU TLB and 512-entry shared TLB, for a physical cache hierarchy with a 128-entry per-CU TLB (black bars), an IDEAL MMU with infinite capacity per-CU TLBs and infinite bandwidth at the shared TLB (gray bars), a virtual cache design (VC 512, red bars) with a 512 entry shared TLB, and a virtual cache design (VC 512 with FBT, orange bars) that employs the forward-backward table (FBT) as a second-level TLB.

Our proposal (VC 512) shows considerable speedup for most of the workloads that are sensitive to address translations (e.g., `bc`, `color_max`, `fw`, `mis`, `pagerank_spmv`, `pagerank`, `kmeans`, `bfs`, and `lud`). The average speedup is about 1.6×, with a maximum speedup of more than 4.5×. Importantly, we do not see any performance degradation.

As described in the previous sections (Section 5.3 and 5.5.1), the main benefits come from reducing the serialization overhead on the shared TLB by filtering out frequent per-CU TLB misses through virtual caches. We also find that the emerging GPU workloads from Pannotia, which is graph-based workloads, show a higher speedup than the traditional workloads from Rodinia due to high memory divergence and greater shared TLB load.

Some workloads (e.g., `fw`, `bfs`, and `nw`) do not achieve the same performance as the IDEAL MMU. This is because of the restricted reach of the 512 entry shared TLB. As described in Section 5.4.1 (see Address Translation), the FBT can be employed as a second-level shared TLB (VC 512 with FBT). By consulting the FBT on a shared TLB miss we can reduce the overhead and achieve the same performance as the IDEAL MMU: 1.77× speedup on average.

Black bars in Figure 5.10 also show the speedup using very large (128-entry) fully associative per-CU TLBs. The results show that large per-CU TLBs can hide some of the address translation overhead when using physical caches. However, using the virtual cache hierarchy still shows about 1.21× speedup over the large per-CU TLBs. In addition, virtual caches also have the benefits of reducing the power, energy, and thermal issues of consulting per-CU TLBs [86, 96].

In addition, we can expect some other benefits by using a virtual cache hierarchy. Actually, some workloads (e.g., `pagerank` and `pagerank_spmv`) slightly outperform the IDEAL MMU. We see better cache locality in virtual caches in some cases, as performance of physical caches is sensitive to memory allocation [20, 68]. For example, `page-rank_spmv` shows more than 3% additional hits in the virtual cache hierarchy than in a physical cache hierarchy.

5.5.3 L1-only Virtual Caches

We also evaluate the effectiveness of a virtual cache design with only virtual L1 caches and a physical L2 cache. This system is similar to previous CPU virtual cache designs.

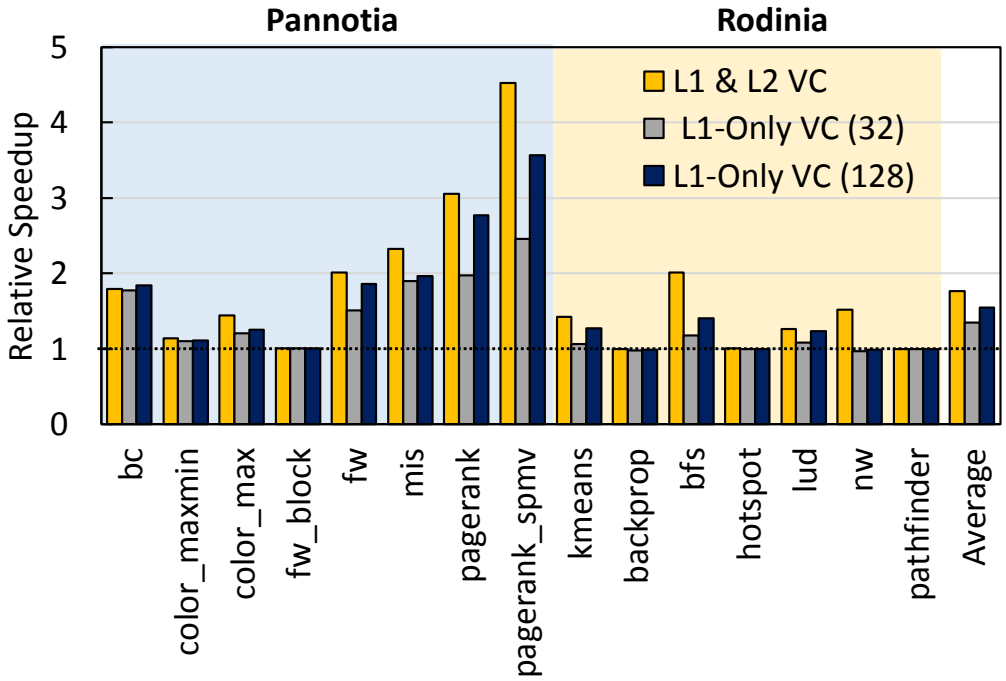


Figure 5.11: Comparison with L1 Virtual Cache Design

Figure 5.11 shows speedup comparison between the whole GPU virtual cache hierarchy (L1 and L2 VC) and two virtual L1 cache designs (L1-only VC). Gray bars show the speedup of the virtual L1-only design that has a 32 per-CU TLB and a 512 entry shared TLB. We also consider a larger (128-entry) per-CU TLB (black bars).

Supporting virtual caches only for GPU L1 cache can also obtain some performance benefits for the address translation sensitive workloads (more than 1.35× speedup on average). This is expected as we found many accesses that miss in per-CU TLBs but hit in the L1 cache (black bars, Figure 5.2).

However, we can still see noticeable gaps compared to the whole virtual cache hierarchy. Because of higher hit ratio, the larger virtual L2 cache plays a vital role as a much larger TLB miss filter; 35% more per-CU TLB misses are filtered out through the L2 virtual cache (see red bars in Figure 5.2). By extending the scope of virtual caches to the whole GPU cache hierarchy, we expect more latency and energy benefits. In addition, we can reduce the design complexity of the GPU cache hierarchy by completely removing per-CU TLB structures from a GPU.

5.6 Chapter Summary

Address translation support on GPUs is important to support flexible programmability. However, especially for emerging GPU workloads, highly divergent memory accesses put considerable pressure on address translation hardware. We show that many of these shared TLB accesses can be eliminated by using a *virtual cache hierarchy*. We make several empirical observations advocating for GPU virtual caches: (1) mirroring CPU-style memory management unit in GPUs is not effective, because GPU workloads show very high Translation Lookaside Buffer (TLB) miss ratio and high miss bandwidth, (2) many requests that miss in TLBs find corresponding valid data in the GPU cache hierarchy, (3) The GPU's accelerator nature simplifies implementing a deep virtual cache hierarchy (i.e., fewer virtual address synonyms and homonyms). We present a practical and software transparent design of virtual caching for the entire GPU cache hierarchy. This virtual cache hierarchy filters most of the shared TLB accesses, which provides about 1.77× speedup on average.

Although we focused solely on GPUs in this work, we believe that other on-die accelerators that use virtual addresses may want to consider using a virtual cache hierarchy. Other accelerators will likely have similar characteristics to GPUs (e.g., unlikely to access virtual address synonyms, usually only executing a code from a single process), which makes them amenable to virtual caching. We believe these GPUs, and potentially these other co-processing units, finally provide an environment where virtual caches are both practical and provide significant benefits.

Chapter 6

RELATED WORK

In this chapter, we review related work. First, we discuss diverse virtual cache proposals by classifying them in terms of how to manage issues of virtual address synonyms (Section 6.1), we also show how they are similar and different, compared to our design (Section 6.1.1). In addition, we study prior research works for GPU address translation (Section 6.2).

6.1 Virtual Caching

A variety of virtual cache designs have been proposed to achieve the benefits of virtual caches for over four decades; prior literature [19, 20, 52] summarized the problems and some proposed solutions. Before proceeding further, we will classify diverse virtual cache designs according to how to handle synonym accesses.

- **Approaches allowing data duplications:** Some approaches [13, 78] allow data duplications in VIVT caches; the same data can be cached with different virtual addresses at the same time. The data duplication could reduce the capacity of the cache. In addition, to prevent the consistency issues due to writes (i.e., stores) with synonyms, additional operations (e.g., identifying and updating the duplicated data in a cache) are required.

  Other proposals [57] allow data duplication while they eliminate data inconsistency issues by employing self-downgrade and self-invalidation for the data potentially causing the consistency issues due to synonyms.

- **Approaches avoiding data duplications:** Other techniques [40, 104] keep a single copy of data (i.e., no data duplication). On accesses with virtual addresses which

are not the same as the virtual address used to place the data, cache misses occur. Furthermore, the current cache line is invalidated and moved to the new location (i.e., data relocation), which also has a data duplication impact.

To reduce the overhead, other approaches [88, 109] use a unique virtual address (one of synonyms) to cache data, which does not allow data duplication. For accesses with other synonymous addresses, virtual address remapping is used to use the correct lookup address instead. They provides an illusion of the use of physical caches, but efficient, scalable remapping approaches are needed.

- **Eliminating virtual address synonyms:** Some other proposals eliminate the occurrence of synonyms (e.g., a single global virtual address space or modifications of memory allocation) [21, 22, 60, 106, 110], which requires operating system involvement. Doing so could restrict not only the design flexibility of OSes but also the programmability.

6.1.1 Details of Prior Approaches

We discuss some of the most relevant work below.

Qui *et al*. [88] proposed a *synonym look-aside buffer* (SLB) for virtual caches. The high level approach of VC-DSR is similar to that of the SLB for enforcing a unique (primary) virtual address among synonyms. The SLB is focused on efficiently resolving the scalability issue of the address translation via a TLB by using a relatively smaller SLB structure. On the other hand, VC-DSR aims for a software-transparent latency and energy efficient L1 virtual cache access by exploiting the temporal behavior of *active synonyms*. Significant differences in how SLB and VC-DSR achieve their goals, and their impact, are discussed below.

First, VC-DSR is a software-transparent virtual cache. For the SLB, considerable OS involvement is required to manage synonyms. It requires additional page table like software

structures to track a primary virtual address and other synonyms. Further, all synonyms need to be identified while the related data resides in a large main memory even though many such pages do not actually face synonym issues during their lifetime in smaller caches. In addition, the primary virtual address can change frequently in many cases (OB_3), thus restricting caching to a single, static (OS-determined) primary address can be unnecessarily constraining. These aspects further complicate the OS memory management.

Second, VC-DSR seamlessly provides a synonym remapping only for accesses with active synonyms. The SLB has to be consulted on every cache access to decide the leading virtual address (not power/energy efficient). Although it could be smaller and more scalable than a normal TLB, the misses for synonyms, i.e., SLB miss traps, are expensive, akin to a page table walk (not latency efficient: Figure 4.15), and all the mappings are shared across different caches in the system. Extra operations are needed to guarantee the consistency among them (like TLB shootdown) although it rarely occurs.

Several others [41, 59, 104] have proposed solutions that employ a variant of the (physical to virtual) reverse map to identify and track the data with synonyms in virtual caches. Goodman [41] proposed one of the earliest hardware solutions by using dual tags in virtual caches as the reverse maps. Wang *et al.* [104] augmented a back-pointer per line in a physical L2 cache to point to the matching data in L1 virtual caches.

Other proposals are supported by the OS to attain the benefits of virtual caches. Some software-based approaches [21, 22, 60, 106] employ a single global virtual address space OS that can eliminate the occurrence of synonyms itself. Zhang *et al.* [110] proposed Enigma using an additional indirection to efficiently avoid synonym issues. It uses a unique intermediate address (IA) space across the entire system for cases where data sharing is not expected. Recently, Basu *et al.* [13] proposed Opportunistic Virtual Caching (OVC) based on minor OS modifications. OVC caches a block either with a virtual address or with a physical address depending on the access pattern in a page. A virtual address will be used when caching data with it is safe (i.e., no read-write synonyms occur) or

efficient (i.e., few permission changes occur), which could result in multiple duplicates in virtual caches. In a similar vein, Park *et al*. proposed a hybrid virtual cache [78]. This also selectively switches between virtual and physical caching like the OVC, while virtual caching is used for the entire cache hierarchy. Like the VC-DSR, this approach also performs synonym detections with Bloom filters prior to L1 cache lookups, while the information of the filters is managed by the OS.

Recent work [57] takes a different approach to simplify virtual cache coherence. It eliminates the reverse translation by employing a simple request-response protocol (e.g., self-invalidation/downgrade). Woo *et al.* [105] employ a Bloom filter to reduce serial cache lookups searching for possible locations of synonyms in virtually indexed caches.

6.2 GPU Address Translation

There have been studies that investigate address translation for the GPU MMU. Power *et al*. [85] discussed diverse design aspects for the integration of a CPU-style MMU into a GPU. Bharath *et al.* [82] also studied how to lower the overhead of GPU address translation, but they more focused on the impact of warp schedule on the address translation. These previous works mainly focus on how to support GPU address translation, while we aim to reduce the overhead of the address translation. Others studied the overheads of virtual address translation on real GPU hardware [102]. Kumar *et al.* proposed Fusion [61], a coherent virtual cache hierarchy for specialized fixed-functional accelerators to reduce data transfer overhead between the accelerators in a tile and the host CPU.

Efficient warp schedulings [91] increase hits in a physical cache hierarchy. If it is applied with virtual caching, more translation hardware accesses can be filtered, which increases the effectiveness of a full virtual cache hierarchy.

Chapter 7

CONCLUSION AND FUTURE WORK

7.1 Thesis Summary and Key Contribution

Supporting virtual memory has been one of the key elements of computers the last several decades. However, two recent trends, i.e., the breakdown of Dennard scaling and the proliferation of heterogeneous computing, pose several challenges in supporting virtual memory, specifically, virtual-to-physical address translation overheads (i.e., performance, power, and energy) due to TLB accesses.

These problems continue to fundamentally exist with physical caches, where a physical address is used for cache access. To resolve the issues, this dissertation research revisits virtual caches (i.e., virtually indexed, virtually tagged caches). We observe temporal characteristics of synonym accesses that are the crux of problems deploying virtual caches. By leveraging the observations, this dissertation proposes a *practical virtual cache design with dynamic synonym remapping* (VC-DSR). VC-DSR satisfies key aspects of practical virtual caches: (1) a software transparent approach, (2) a simple design, and (3) efficient synonym management.

The thesis of this dissertation is that (1) our proposal efficiently manages virtual cache complications (e.g., synonyms), and that (2) it achieves most of the benefits of canonical (but impractical) virtual caches. To substantiate this thesis, we presented the following key contributions:

1. **Analysis of issues of virtual cache integration:** This thesis describes challenges of deployment of virtual caches with respect to each aspect of virtual memory: (1) changes in virtual-to-physical address mapping, (2) access permission checking, and

(3) virtual address synonyms (due to data sharing). Especially for issues resulting from virtual address synonyms, we also discuss what prior literature has not studied, such as potential memory consistency issues and other consistency issues of buffered instructions in pipeline stages due to self-modifying codes (in Section 2.2).

2. **Anatomy of sources of synonyms:** We characterize and classify sources of virtual address synonyms according to types of interactions between user and kernel operations (in Section 3.1).

3. **Analysis of temporal behavior of synonyms:** Synonym accesses are common over the execution of a program, while the (active) synonyms are short-lived in smaller caches like an L1 cache. Furthermore, a very small percentage of active synonym accesses are to the same data that is cached with a different virtual address. We present the novel empirical observations on the temporal behavior of synonyms, based on real world server and mobile workloads (in Section 3.3).

4. **L1 virtual cache as a TLB Lookup filter for a CPU:** By leveraging the temporal properties of (active) synonyms, this thesis proposes a practical L1 virtual cache with dynamic synonym remapping and evaluates the efficacy. The experimental results show that our proposal achieves most of the latency, power, and energy benefits of virtual caches in a software transparent manner (in Section 4).

5. **Analysis of address translation overheads on GPU memory accesses:** GPU memory access patterns are analyzed based on diverse GPU applications. We identify that a major source of GPU address translation overheads results from the significant bandwidth demand for the shared TLB due to frequent private TLB misses (in Section 5.2).

6. **Virtual cache hierarchy as a TLB miss bandwidth filter for GPUs:** We study the expected benefits by using virtual caches as a private TLB miss bandwidth filter. We

consider both an L1-only virtual cache and an entire virtual cache hierarchy (L1 and L2 caches). Based on the empirical observations, we propose the first pure virtual cache hierarchy for GPUs (in Section 5.4).

7.2 Future Opportunities and Directions

This dissertation also made diverse contributions for researchers and industry. As we discussed above, the use of VC-DSR by itself provides performance, power, and energy benefits as not only a *TLB lookup filter* but also as a *TLB miss bandwidth filter*. However, integrating VC-DSR into modern processors opens several opportunities for optimizing other features of microarchitecture. Along with future trends of computing environments, in addition, we need to answer questions about the effectiveness of our proposal.

7.2.1 Design Flexibility of Virtual Caching.

We can expect additional opportunities by exploiting the design flexibility of virtual caching in several ways. As we discussed, virtual caches (VCs) filter TLB lookups on cache hits, and thus TLBs are relatively rarely consulted. Hence, VCs would permit us to have a larger, slower, lower-associativity, fewer ports TLB design, having lower energy consumption and TLB miss rate. Additionally, the access pattern of TLBs will be different from that of the conventional TLBs because the VCs can act as a TLB miss filter. This also enables us to design more efficient TLB management (e.g., prefetching). In practice, the use of VCs is orthogonal to the organization of TLBs. We can apply any prior TLB optimization techniques. Also, VCs allows different cache geometry, e.g., by removing the large associativity (i.e., high power/energy) constraint of a virtually indexed, physically tagged cache design.

7.2.2 Supporting Virtualized Systems.

TLB miss overheads are expected to be much worse due to new computing environments. Recently, cloud computing has become widespread. The virtualized environment provides several benefits (e.g., security, resource consolidation, and isolation) to build up the infrastructure. Due to the two-level address translations between guest and host systems under the virtualized environment, however, the overhead of virtualizing memory operations especially for TLB misses is considerable [37]. Regarding future big memory workloads, some processor vendors and the OS community have started to discuss supporting 5-level paging (e.g., previously 4-level paging on x86) [47]. Thus, the overhead will be much worse. In addition, it is highly likely to observe more frequent synonym accesses across independent virtual machines. The temporal characteristics of active synonyms for such workloads are of interest. Virtual caching can open new opportunities to reduce the overhead. We need to evaluate the potential benefits of virtual caching as a TLB lookup filter and a TLB miss filter.

Appendix A

DETAILS OF COHERENCE PROTOCOL

We described high-level operations of each structures supporting the proposed virtual cache design in Section 4.2 and 5.4. In the appendix, we discuss more details of the coherence operations of the structures, especially for the Active Synonym Detection Table (ASDT). The ASDT sits on the border between virtual and physical caches.[1] This structure deals with active synonym accesses by interacting with a Translation Lookaside Buffer (TLB) and also forwards all messages between virtual caches and lower-level physical caches (or a directory) to support appropriate data accesses and coherence operations. Figure A.1 and A.2 show the details of the operations. Each row indicates a `state`, while each column indicates an `event` triggered by internal operations or coherence operations from other structures in the cache hierarchy. Each entry associated with one of the states and one of the events lists required `actions` and the `next state`.

Description of States. First, we describe the specification of each state.

- **I state:** indicates that there are no corresponding entries for a physical page in both an ASDT and a TLB. For example, the first access to a physical page could be one scenario leading to this case; (1) address translations for the page have never occurred and thus (2) any data from the page has not been cached in virtual caches.

- **I_Cached:** indicates a page table walk is outstanding in order to fetch the corresponding translation. This state is from I state on TLB misses.

- **(Both) Cached:** indicates that the corresponding TLB entry is cached and the ASDT entry is created.

[1] For the CPU virtual cache design (Section 4.2), it is consulted on L1 VC misses. For the GPU virtual cache hierarchy (Section 5.4), the structure is named as *forward table* and accessed on shared L2 VC misses.

- **NoTLB:** indicates that there is no corresponding TLB entry while we have the matching ASDT entry. Actually, it is possible for a TLB entry to be evicted earlier than an ASDT entry for the same physical page.

- **NoTLB_Cached:** indicates that we currently has an outstanding page table walk from NoTLB state. This state is from NoTLB state on TLB misses.

Description of Events. Here, we describe the specification of each event.[2] We use red text to indicates events caused by a message from a virtual cache. We use blue text to indicate events caused by a message from a physical cache (or a directory) with a physical address. We use green text to indicates events caused by interactions with a TLB.

- **ForwardUnblockToDir:** We simply forward an unblock message from a virtual cache to a directory (or a physical cache).

- **ForwardResponseToDir:** We simply forward a coherence response message from a virtual cache to a directory (or a physical cache).

- **ForwardRequestToDir:** We simply forward a coherence request (e.g., read/write permission) message from a virtual cache to a directory (or a physical cache).

- **CleanPut:** On an eviction of clean data from a virtual cache, we need to update information of a corresponding ASDT entry such as a bit vector or a counter value, which helps to keep track of inclusive information of cache data with an ASDT.

- **ForwardPUTXToDir:** On an eviction of dirty data from a virtual cache, as described above, the information of a matching ASDT entry needs to be updated. In addition, the dirty data should be written back to lower level caches for caches with a write back property.

[2]For simplicity, we consider a large ASDT which is enough to track all distinct physical pages whose data reside in virtual caches. Thus, we do not need to consider an eviction of an ASDT entry.

- **ForwardForwardToCache:** We simply forward a coherence request message from a directory (or a physical cache) to a virtual cache.
- **ForwardResponseToCache:** We simply forward a response message from a directory (or a physical cache) to a virtual cache.
- **WalFinishedRequest:** indicates a page table walk on a TLB miss is done.
- **TLBEviction:** evict a victim TLB entry due to lack of entries.

<table>
<tr><th></th><th>ForwardUnblockToDir</th><th>ForwardResponseToDir</th><th>ForwardRequestToDir</th><th>CleanPut</th><th>ForwardPUTXToDir</th><th></th></tr>
<tr><td>I</td><td></td><td></td><td>- Allocate a TLB entry
- Perform PTW on a TLB miss
Next State: I Cached</td><td></td><td></td><td>I</td></tr>
<tr><td>I Cached</td><td></td><td>- Stall the request and wait</td><td>- Stall the request and wait</td><td></td><td></td><td>I Cached</td></tr>
<tr><td>Cached</td><td colspan="2">- Perform VtoP addr translation via a TLB
- Forward the msg</td><td>- Perform VtoP addr translation via a TLB
- Lookup the corresponding ASDT
- Check active synonym access: Replay or Fwd the msg</td><td rowspan="3">- Lookup a matching ASDT entry
- Update inclusive information
- No msg forwarding</td><td rowspan="3">- Perform VtoP addr trans. via an ASDT or a Backward Table
- Forward the msg</td><td>Cached</td></tr>
<tr><td>NoTLB</td><td colspan="2" rowspan="2">- Perform VtoP addr translation via an ASDT (using the ASDT index) or a Backward Table
- Forward the msg</td><td>- Allocate a TLB entry
- Perform PTW on a TLB miss
Next State: NoTLB Cached</td><td>NoTLB</td></tr>
<tr><td>NoTLB Cached</td><td>- Stall the request and wait</td><td>NoTLB Cached</td></tr>
<tr><td></td><td>ForwardUnblockToDir</td><td>ForwardResponseToDir</td><td>ForwardRequestToDir</td><td>CleanPut</td><td>ForwardPUTXToDir</td><td></td></tr>
</table>

Figure A.1: Transition table for events caused by a message from virtual caches

	ForwardForwardToCache	ForwardResponseToCache	WalkFinishedRequest	TLBEviction	
I	- Send Nack to the sender				**I**
I Cached	- Stall the request and wait		- Update the physical page of the TLB entry - Allocate the corresponding ASDT entry - Wakeup all pending requests Next State: Cached	- Stall the request and wait	**I Cached**
Cached	- Perform PtoV addr trans. via an ASDT - Forward the msg to a VC			- Deallocate a TLB entry Next State: NoTLB	**Cached**
NoTLB					**NoTLB**
NoTLB Cached			- Update the physical page of the TLB entry - Wakeup all pending requests Next State: Cached	- Stall the request and wait	**NoTLB Cached**
	ForwardForwardToCache	ForwardResponseToCache	WalkFinishedRequest	TLBEviction	

Figure A.2: Transition table for events caused by a message from physical caches (or a directory) and by the interactions between a TLB and an ASDT

REFERENCES

[1] Inside Intel Next Generation Nehalem Microarchitecture. http://www.realworldtech.com/nehalem/4/.

[2] IOMMU: Virtualizing IO through IO Memory Management Unit (IOMMU). http://pages.cs.wisc.edu/ basu/isca_iommu_tutorial/index.htm.

[3] Kernel Samepage Merging. https://www.kernel.org/doc/Documentation/vm/ksm.txt.

[4] PS4 Pro Specs: How Does It Fare Against Xbox Project Scorpio? Which One Is Better? http://www.itechpost.com/articles/50922/20161107/ps4-pro-specs-fare-against-xbox-project-scorpio-one-better.htm.

[5] Agarwal, Anant, John Hennessy, and Mark Horowitz. 1988. Cache Performance of Operating System and Multiprogramming Workloads. *ACM Trans. Comput. Syst.* 6(4):393–431.

[6] ARM. . ARM Cortex-A Series Programmer's Guide for ARMv8-A Version: 1.0. https://static.docs.arm.com/den0024/a/DEN0024.pdf.

[7] ———. . ARM Cortex-A72 MPCore Processor Technical Reference Manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.100095_0001_02_en/index.html.

[8] ———. . Migrating a Software Application from ARMv5 to ARMv7-A/R Application Note 425. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0425/index.html.

[9] Austin, Todd M., and Gurindar S. Sohi. 1996. High-bandwidth Address Translation for Multiple-issue Processors. In *Proceedings of the 23rd annual international symposium on computer architecture*, 158–167. ISCA '96, New York, NY, USA: ACM.

[10] Baer, Jean-Loup, and Tien-Fu Chen. 1991. An Effective On-chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of the 1991 acm/ieee conference on supercomputing*, 176–186. Supercomputing '91, New York, NY, USA: ACM.

[11] Barr, Thomas W., Alan L. Cox, and Scott Rixner. 2011. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the 38th annual international symposium on computer architecture*, 307–318. ISCA '11, New York, NY, USA: ACM.

[12] Basu, Arkaprava, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 237–248. ISCA '13, New York, NY, USA: ACM.

[13] Basu, Arkaprava, Mark D. Hill, and Michael M. Swift. 2012. Reducing Memory Reference Energy with Opportunistic Virtual Caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 297–308. ISCA '12, Washington, DC, USA: IEEE Computer Society.

[14] Bhattacharjee, Abhishek, Daniel Lustig, and Margaret Martonosi. 2011. Shared Last-level TLBs for Chip Multiprocessors. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 62–63. HPCA '11, Washington, DC, USA: IEEE Computer Society.

[15] Binkert, Nathan, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and

David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39(2): 1–7.

[16] Bloom, Burton H. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13(7):422–426.

[17] Boncz, Peter A., Martin L. Kersten, and Stefan Manegold. 2008. Breaking the Memory Wall in MonetDB. *Commun. ACM* 51(12):77–85.

[18] Cain, Harold W., and Mikko H. Lipasti. 2004. Memory Ordering: A Value-Based Approach. In *Proceedings of the 31st annual international symposium on computer architecture*, 90–. ISCA '04, Washington, DC, USA: IEEE Computer Society.

[19] Cekleov, Michel, and Michel Dubois. 1997. Virtual-Address Caches Part 1: Problems and Solutions in Uniprocessors. *IEEE Micro* 17(5):64–71.

[20] ———. 1997. Virtual-Address Caches, Part 2: Multiprocessor Issues. *IEEE Micro* 17(6):69–74.

[21] Chase, Jeffrey S., Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. 1994. Sharing and Protection in a Single-address-space Operating System. *ACM Trans. Comput. Syst.* 12(4):271–307.

[22] Chase, Jeffrey S., Henry M. Levy, Edward D. Lazowska, and Miche Baker-Harvey. 1992. Lightweight Shared Objects in a 64-bit Operating System. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, 397–413. OOPSLA '92, New York, NY, USA: ACM.

[23] Che, Shuai, Bradford M Beckmann, Steven K Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), 2013 IEEE International Symposium on*, 185–195. IEEE.

[24] Che, Shuai, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 44–54. IEEE.

[25] Chen, Xuhao, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. 2014. Adaptive Cache Management for Energy-Efficient GPU Computing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 343–355. MICRO-47, Washington, DC, USA: IEEE Computer Society.

[26] Choi, Jin-Hyuck, Jung-Hoon Lee, Seh-Woong Jeong, Shin-Dug Kim, and Charles Weems. 2002. A Low Power TLB Structure for Embedded Systems. *IEEE Comput. Archit. Lett.* 1(1):3–3.

[27] Clark, Douglas W., and Joel S. Emer. 1985. Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement. *ACM Trans. Comput. Syst.* 3(1):31–62.

[28] Couleur, John. 1995. The Core of the Black Canyon Computer Corporation. *IEEE Ann. Hist. Comput.* 17(4):56–60.

[29] Council, National Research, et al. 2011. *The Future of Computing Performance: Game Over or Next Level?* National Academies Press.

[30] Dennard, Robert H., Fritz H. Gaensslen, Hwa nien Yu, V. Leo Rideout, Ernest Bassous, Andre, and R. Leblanc. 1974. Design of ion-implanted MOSFETs with very small physical dimensions. *IEEE J. Solid-State Circuits* 256.

[31] Denning, Peter J. 2005. The Locality Principle. *Commun. ACM* 48(7):19–24.

[32] Denning, Peter J, and Peter J. Denning. 1997. Before Memory Was Virtual.

[33] Ekman, Magnus, Per Stenström, and Fredrik Dahlgren. 2002. TLB and Snoop Energy-reduction Using Virtual Caches in Low-power Chip-multiprocessors. In *Proceedings*

of the 2002 International Symposium on Low Power Electronics and Design (ISLPED), 243–246.

[34] F, C.J., and G.E. L. 1968. Shared-access data processing system. US Patent 3,412,382.

[35] Fan, Dongrui, Zhimin Tang, Hailin Huang, and Guang R. Gao. 2005. An Energy Efficient TLB Design Methodology. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design (ISLPED)*, 351–356.

[36] Ferdman, Michael, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 37–48.

[37] Gandhi, Jayneel, Mark D. Hill, and Michael M. Swift. 2016. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *Proceedings of the 43rd international symposium on computer architecture*, 707–718. ISCA '16, Piscataway, NJ, USA: IEEE Press.

[38] Geekbench. http://www.primatelabs.com/geekbench/.

[39] Gharachorloo, Kourosh, Anoop Gupta, and John Hennessy. 1991. Two Techniques to Enhance the Performance of Memory Consistency Models. In *In Proceedings of the 1991 International Conference on Parallel Processing*, 355–364.

[40] Goodman, James R. 1987. Coherency for Multiprocessor Virtual Address Caches. *SIGPLAN Not.* 22(10):72–81.

[41] ———. 1987. Coherency for Multiprocessor Virtual Address Caches. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS)*, 72–81.

[42] Hennessy, John L., and David A. Patterson. 2006. *Computer architecture, fourth edition: A quantitative approach*. Morgan Kaufmann Publishers Inc.

[43] ———. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[44] Henning, John L. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34(4):1–17.

[45] Horowitz, M, E Alon, D Patil, S Naffziger, R Kumar, and K Bernstein. 2005. Scaling, power, and the future of CMOS. In *Ieee internationalelectron devices meeting, 2005. iedm technical digest.*, 9–15. IEEE.

[46] Hower, Derek R., Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. Heterogeneous-race-free Memory Models. In *Proceedings of the 19th international conference on architectural support for programming languages and operating systems*, 427–440. ASPLOS '14, New York, NY, USA: ACM.

[47] Intel. 2016. 5-Level Paging and 5-Level EPT. https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf.

[48] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, Part1, Chapter 2. http://www.intel.com/Assets/en_US/PDF/manual/253668.pdf.

[49] Jacob, Bruce. 2009. *The Memory System: You Can'T Avoid It, You Can'T Ignore It, You Can'T Fake It*. Morgan and Claypool Publishers.

[50] Jacob, Bruce, and Trevor Mudge. 1998. Virtual Memory in Contemporary Microprocessors. *IEEE Micro* 18(4):60–75.

[51] ———. 2001. Uniprocessor Virtual Memory Without TLBs. *IEEE Trans. Comput.* 50(5):482–499.

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

[53] Juan, Toni, Tomas Lang, and Juan J. Navarro. 1997. Reducing TLB Power Requirements. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design (ISLPED)*, 196–201.

[54] Kadayif, I., P. Nath, M. Kandemir, and A. Sivasubramaniam. 2007. Reducing Data TLB Power via Compiler-Directed Address Generation. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 26(2):312–324.

[55] Kadayif, I., A. Sivasubramaniam, M. Kandemir, G. Kandiraju, and G. Chen. 2002. Generating Physical Addresses Directly for Saving Instruction TLB Energy. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 185–196.

[56] Karakostas, V., J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal. 2016. Energy-efficient address translation. In *2016 ieee international symposium on high performance computer architecture (hpca)*, 631–643.

[57] Kaxiras, Stefanos, and Alberto Ros. 2013. A New Perspective for Efficient Virtual-cache Coherence. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 535–546. ISCA '13, New York, NY, USA: ACM.

[58] Kim, Hyesoon. 2012. Supporting Virtual Memory in GPGPU Without Supporting Precise Exceptions. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 70–71. MSPC '12, New York, NY, USA: ACM.

[59] Kim, Jesung, Sang Lyul Min, Sanghoon Jeon, Byoungchu Ahn, Deog-Kyoon Jeong, and Chong Sang Kim. 1995. U-cache: A Cost-effective Solution to Synonym Problem.

In *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 243–252.

[60] Koldinger, Eric J., Jeffrey S. Chase, and Susan J. Eggers. 1992. Architecture Support for Single Address Space Operating Systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 175–186. ASPLOS V, New York, NY, USA: ACM.

[61] Kumar, Snehasish, Arrvindh Shriraman, and Naveen Vedula. 2015. Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, 733–745. ISCA '15, New York, NY, USA: ACM.

[62] Kyriazis, George. 2012. Heterogeneous system architecture: A technical review. *AMD Fusion Developer Summit*.

[63] Lee, Hsien-Hsin S., and Chinnakrishnan S. Ballapuram. 2003. Energy Efficient D-TLB and Data Cache Using Semantic-aware Multilateral Partitioning. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design (ISLPED)*, 306–311.

[64] Lee, Lea Hwang, Bill Moyer, and John Arends. 1999. Instruction Fetch Energy Reduction Using Loop Caches for Embedded Applications with Small Tight Loops. In *Proceedings of the 1999 international symposium on low power electronics and design*, 267–269. ISLPED '99, New York, NY, USA: ACM.

[65] Liptay, J. S. 1968. Structural aspects of the System/360 Model 85, II: The cache. *IBM Systems Journal* 7(1):15–21.

[66] Luk, Chi-Keung, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Pro-*

ceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, 190–200. PLDI '05, New York, NY, USA: ACM.

[67] ———. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 acm sigplan conference on programming language design and implementation (pldi)*, 190–200.

[68] Lynch, William L. 1993. The Interaction of Virtual Memory and Cache Memory. Tech. Rep. CSL-TR-93-587, Stanford University.

[69] Memcached. http://memcached.org.

[70] Menon, Jaikrishnan, Marc De Kruijf, and Karthikeyan Sankaralingam. 2012. iGPU: Exception Support and Speculative Execution on GPUs. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 72–83. ISCA '12, Washington, DC, USA: IEEE Computer Society.

[71] Moore, G. E. 2006. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter* 11(5):33–35.

[72] Mudge, T. 2001. Power: A first-class architectural design constraint. *Computer* 34(4): 52–58.

[73] Muralimanohar, Naveen, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP Laboratories*.

[74] Navarro, Juan, Sitararn Iyer, Peter Druschel, and Alan Cox. 2002. Practical, Transparent Operating System Support for Superpages. *SIGOPS Oper. Syst. Rev.* 36(SI): 89–104.

[75] Olson, Lena E., Jason Power, Mark D. Hill, and David A. Wood. 2015. Border Control: Sandboxing Accelerators. In *Proceedings of the 48th International Symposium on Microarchitecture*, 470–481. MICRO-48, New York, NY, USA: ACM.

[76] Pai, Vijay S., Parthasarathy Ranganathan, Sarita V. Adve, and Tracy Harton. 1996. An Evaluation of Memory Consistency Models for Shared-memory Systems with ILP Processors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 12–23. ASPLOS VII, New York, NY, USA: ACM.

[77] Palacharla, S., and R. E. Kessler. 1994. Evaluating Stream Buffers As a Secondary Cache Replacement. In *Proceedings of the 21st annual international symposium on computer architecture*, 24–33. ISCA '94, Los Alamitos, CA, USA: IEEE Computer Society Press.

[78] Park, Chang Hyun, Taekyung Heo, and Jaehyuk Huh. 2016. Efficient Synonym Filtering and Scalable Delayed Translation for Hybrid Virtual Caching. In *Proceedings of the 43th Annual International Symposium on Computer Architecture*. ISCA '16, Washington, DC, USA: IEEE Computer Society.

[79] Patel, Avadh, Furat Afram, Shunfei Chen, and Kanad Ghose. 2011. MARSS: A Full System Simulator for Multicore x86 CPUs. In *Proceedings of the 48th design automation conference (dac)*, 1050–1055.

[80] Patterson, David A., and John L. Hennessy. 2007. *Computer Organization and Design: The Hardware/Software Interface*. 3rd ed. Morgan Kaufmann Publishers Inc.

[81] Pham, Binh, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 258–269. MICRO-45, Washington, DC, USA: IEEE Computer Society.

[82] Pichai, Bharath, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 743–758. ASPLOS '14, New York, NY, USA: ACM.

[83] Power, Jason, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M Beckmann, Mark D Hill, Steven K Reinhardt, and David A Wood. 2013. Heterogeneous system coherence for integrated CPU-GPU systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 457–467. ACM.

[84] Power, Jason, Joel Hestness, Marc Orr, Mark Hill, and David Wood. 2014. gem5-gpu: A Heterogeneous CPU-GPU Simulator. *Computer Architecture Letters* 13(1).

[85] Power, Jason, Mark D Hill, and David A Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 568–578. IEEE.

[86] Puttaswamy, Kiran, and Gabriel H. Loh. 2006. Thermal Analysis of a 3D Die-stacked High-performance Microprocessor. In *Proceedings of the 16th ACM Great Lakes Symposium on VLSI*, 19–24. GLSVLSI '06, New York, NY, USA: ACM.

[87] Qiu, Xiaogang, and Michel Dubois. 1999. Tolerating Late Memory Traps in ILP Processors. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 76–87. ISCA '99, Washington, DC, USA: IEEE Computer Society.

[88] ———. 2008. The Synonym Lookaside Buffer: A Solution to the Synonym Problem in Virtual Caches. *IEEE Trans. Comput.* 57(12):1585–1599.

[89] Rivers, Jude A., Gary S. Tyson, Edward S. Davidson, and Todd M. Austin. 1997. On High-bandwidth Data Cache Design for Multi-issue Processors. In *Proceedings of*

the 30th annual acm/ieee international symposium on microarchitecture, 46–56. MICRO 30, Washington, DC, USA: IEEE Computer Society.

[90] Roberts, D, T Layman, and G Taylor. 1990. An ECL RISC microprocessor designed for two level cache. In *Digest of Papers Compcon Spring '90. Thirty-Fifth IEEE Computer Society International Conference on Intellectual Leverage*, 228–231. IEEE Comput. Soc.

[91] Rogers, Timothy G., Mike O'Connor, and Tor M. Aamodt. 2012. Cache-Conscious Wavefront Scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 72–83. MICRO-45, Washington, DC, USA: IEEE Computer Society.

[92] Rotenberg, Eric, Steve Bennett, and James E. Smith. 1996. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *Proceedings of the 29th annual acm/ieee international symposium on microarchitecture*, 24–35. MICRO 29, Washington, DC, USA: IEEE Computer Society.

[93] Roth, Amir. 2005. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *Proceedings of the 32nd annual international symposium on computer architecture*, 458–468. ISCA '05, Washington, DC, USA: IEEE Computer Society.

[94] Singh, Abhayendra, Shaizeen Aga, and Satish Narayanasamy. 2015. Efficiently Enforcing Strong Memory Ordering in GPUs. In *Proceedings of the 48th International Symposium on Microarchitecture*, 699–712. MICRO-48, New York, NY, USA: ACM.

[95] Singh, Inderpreet, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. 2013. Cache Coherence for GPU Architectures. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 578–590. HPCA '13, Washington, DC, USA: IEEE Computer Society.

[96] Sodani, Avinash. 2011. Race to Exascale: Opportunities and Challenges. MICRO 2011 Keynote talk.

[97] Sorin, Daniel J., Mark D. Hill, and David A. Wood. 2011. *A Primer on Memory Consistency and Cache Coherence*. 1st ed. Morgan & Claypool Publishers.

[98] SPECjbb2005. http://www.spec.org/jbb2005.

[99] Talluri, Madhusudhan, Shing Kong, Mark D. Hill, and David A. Patterson. 1992. Tradeoffs in Supporting Two Page Sizes. In *Proceedings of the 19th annual international symposium on computer architecture*, 415–424. ISCA '92, New York, NY, USA: ACM.

[100] Taylor, George, Peter Davies, Michael Farmwald, George Taylor, Peter Davies, and Michael Farmwald. 1990. The TLB slice—a low-cost high-speed address translation mechanism. *ACM SIGARCH Computer Architecture News* 18(2SI):355–363.

[101] TPC-H. http://www.tpc.org/tpch/default.asp.

[102] Vesely, J., A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee. 2016. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 161–171.

[103] Villavieja, Carlos, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrián Cristal, and Osman S Unsal. 2011. DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory. In *2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 340–349. IEEE.

[104] Wang, W. H., J.-L. Baer, and H. M. Levy. 1989. Organization and Performance of a Two-level Virtual-real Cache Hierarchy. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 140–148. ISCA '89, New York, NY, USA: ACM.

[105] Woo, Dong Hyuk, Mrinmoy Ghosh, Emre Özer, Stuart Biles, and Hsien-Hsin S. Lee. 2006. 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 (cases)*, 179–189.

[106] Wood, D. A., S. J. Eggers, G. Gibson, M. D. Hill, and J. M. Pendleton. 1986. An In-cache Address Translation Mechanism. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, 358–365. ISCA '86, Los Alamitos, CA, USA: IEEE Computer Society Press.

[107] Wu, C. E., Y. Hsu, and Y. H. Liu. 1993. A Quantitative Evaluation of Cache Types for High-Performance Computer Systems. *IEEE Trans. Comput.* 42(10):1154–1162.

[108] Yeager, Kenneth C. 1996. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro* 16(2):28–40.

[109] Yoon, Hongil, and Gurindar S. Sohi. 2016. Revisiting virtual L1 caches: A practical design using dynamic synonym remapping. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 212–224.

[110] Zhang, Lixin, Evan Speight, Ram Rajamony, and Jiang Lin. 2010. Enigma: Architectural and Operating System Support for Reducing the Impact of Address Translation. In *Proceedings of the 24th ACM International Conference on Supercomputing*, 159–168. ICS '10, New York, NY, USA: ACM.