REVISITING VIRTUAL MEMORY

By

Arkaprava Basu

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

2013

Date of final oral examination: 2nd December 2013.

The dissertation is approved by the following members of the Final Oral Committee:

Prof. Remzi H. Arpaci-Dusseau, Professor, Computer Sciences
Prof. Mark D. Hill (Advisor), Professor, Computer Sciences
Prof. Mikko H. Lipasti, Professor, Electrical and Computer Engineering
Prof. Michael M. Swift (Advisor), Associate Professor, Computer Sciences
Prof. David A. Wood, Professor, Computer Sciences

Dedicated to my parents Susmita and Paritosh Basu for their selfless and unconditional love and support.

Abstract

Page-based virtual memory (paging) is a crucial piece of memory management in today's computing systems. However, I find that need, purpose and design constraints of virtual memory have changed dramatically since translation lookaside buffers (TLBs) were introduced to cache recently-used address translations: (a) physical memory sizes have grown more than a million-fold, (b) workloads are often sized to avoid swapping information to and from secondary storage, and (c) energy is now a first-order design constraint. Nevertheless, level-one TLBs have remained the same size and are still accessed on every memory reference. As a result, large workloads waste considerable execution time on TLB misses and all workloads spend energy on frequent TLB accesses.

In this thesis I argue that it is *now time to reevaluate virtual memory management***.** I reexamine virtual memory subsystem considering the ever-growing latency overhead of address translation and considering energy dissipation, developing three results.

First, I proposed direct segments to reduce the latency overhead of address translation for emerging big-memory workloads. Many big-memory workloads allocate most of their memory early in execution and do not benefit from paging. Direct segments enable hardware-OS mechanisms to bypass paging for a part of a process's virtual address space, eliminating nearly 99% of TLB miss for many of these workloads.

Second, I proposed opportunistic virtual caching (OVC) to reduce the energy spent on translating addresses. Accessing TLBs on each memory reference burns significant energy, and virtual memory's page size constrains L1-cache designs to be highly associative -- burning yet more energy. OVC makes hardware-OS modifications to expose energy-efficient virtual caching as a dynamic optimization. This saves 94-99% of TLB lookup energy and 23% of L1-cache lookup energy across several workloads.

Third, large pages are likely to be more appropriate than direct segments to reduce TLB misses under frequent memory allocations/deallocations. Unfortunately, prevalent chip designs like Intel's, statically partition TLB resources among multiple page sizes, which can lead to performance pathologies for using large pages. I proposed the merged-associative TLB to avoid such pathologies and reduce TLB miss rate by up to 45% through dynamic aggregation of TLB resources across page sizes.

Acknowledgements

It is unimaginable for me to come this far to write the acknowledgements for my PhD thesis without the guidance and the support of my wonderful advisors – Prof. Mark Hill and Prof. Mike Swift.

I am deeply indebted to Mark not only for his astute technical advice, but also for his sage life-advices. He taught me how to conduct research, how to communicate research ideas to others and how to ask relevant research questions. He has been a pillar of support for me during tough times that I had to endure in the course of my graduate studies. It would not have been possible for me to earn my PhD without Mark's patient support. Beyond academics, Mark has always been a caring guardian to me for past five years. I fondly remember how Mark took me to my first football game at the Camp Randall stadium a few weeks before my thesis defense so that I do not miss out an important part of the *Wisconsin Experience*. Thanks Mark for being my advisor!

I express my deep gratitude to Mike. I have immense admiration for his deep technical knowledge across the breadth of computer science. His patience, support and guidance have been instrumental in my learning. My interest in OS-hardware coordinated design is in many ways shaped by Mike's influence. I am indebted to Mike for his diligence in helping me shape research ideas and present them for wider audience. It is hard to imagine for me to do my PhD in virtual memory management without Mike's help. Thanks Mike for being my advisor!

I consider myself lucky to be able to interact with great faculty of this department. I always found my discussions with Prof. David Wood to be great learning experiences. I will have fond memories of interactions with Prof. Remzi Arpaci-Dusseau, Prof. Shan Lu, Prof. Karu Sankaralingam, Prof. Guri Sohi.

I thank my student co-authors with whom I had opportunity to do research. I learnt a lot from my long and deep technical discussion with Jayaram Bobba, Derek Hower and Jayneel Gandhi. I have greatly benefited from bouncing ideas off them. In particular, I acknowledge Jayneel's help for a part of my thesis.

I would like to thank former and current and students of the department with whom I had many interactions, including Mathew Allen, Shoaib Altaf, Newsha Ardalani, Raghu Balasubramanian, Yasuko Eckert, Dan Gibson, Polina Dudnik, Venkataram Govindaraju, Gagan Gupta, Asim Kadav, Jai Menon, Lena Olson, Sankaralingam Panneerselvam, Jason Power, Somayeh Sardashti, Mohit Saxena, Rathijit Sen, Srinath Sridharan, Nilay Vaish, Venkatanathan Varadarajan, James Wang. They made my time at Wisconsin enjoyable.

I would like to extend special thanks to Haris Volos, with whom I shared an office for more than four years. We shared many ups and downs of the graduate student life. Haris helped me in taking my first steps in hacking Linux kernel during my PhD. I am also thankful to Haris for gifting his car to me when he graduated and left Madison!

I thank AMD Research for my internship that enabled me to learn great deal about research in the industrial setup. In particular, I want to thank Brad Beckmann and Steve Reinhardt for making the internship both an enjoyable and learning experience. I would like to thank Wisconsin Computer Architecture Affiliates for their feedbacks and suggestions on my

research works. I want to extend special thanks to Jichuan Chang with whom I had opportunity to collaborate for a part of my thesis work. Jichuan has also been great mentor to me.

I want to thank my personal friends Rahul Chatterjee, Moitree Laskar, Uttam Manna, Tumpa MannaJana, Anamitra RayChoudhury, Subarna Tripathi for their support during my graduate studies.

This work was supported in part by the US National Science Foundation (CNS-0720565, CNS-0834473, CNS-0916725, CNS-1117280, CNS-1117280, CCF-1218323, and CNS-1302260), Sandia/DOE (#MSN 123960/DOE890426), and donations from AMD and Google.

And finally, I want to thank my dear parents Paritosh and Susmita Basu – I cannot imagine a life without their selfless love and support.

Table of Contents

1

Introduction

"Virtual memory was invented at the time of scarcity. Is it still a good idea?"

--Charles Thacker, 2010 ACM Turing award lecture.

Page-based virtual memory (paging) is a crucial piece of memory management in today's computing systems. software accesses memory using a virtual address that must be translated to a physical address before the memory access can be completed. This virtual-to-physical address translation process goes through the page-based virtual memory subsystem in every current commercial general-purpose processor that I am aware of. Thus, efficient address translation mechanism is prerequisite for efficient memory access and thus ultimately for efficient computing. Notably though, virtual address translation mechanism's basic formulation remains largely unchanged since the late 1960s when translation lookaside buffers (TLB) were

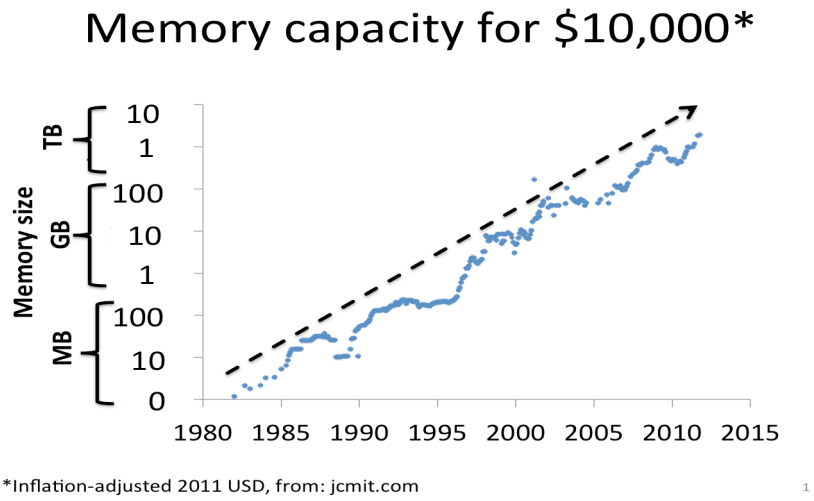


Figure 1-1. Growth of physical memory.

introduced to efficiently cache recently used address translations. However, the purpose, usage and the design constraints of virtual memory have witnessed a sea change in the last decade.

In this thesis I argue that it is *now time to reevaluate the virtual memory management***.** There are at least two key motivations behind the need to revisit virtual memory management techniques. First, there has been *significant change in the needs and the purpose of virtual memory*. For example, the amount of memory that needs address translation is a few orders of magnitude larger than a decade ago. Second, there are new key constraints on how one designs computing systems today. For example today's systems are most often power limited, unlike those from a decade ago.

Evolved Needs and Purposes: The steady decline in the cost of physical memory enabled a million-times larger physical memory in today's systems then during the inception of the page-based virtual memory. Figure 1-1 shows the amount of physical memory (DRAM) that

could be purchased in 10,000 inflation-adjusted US dollar since 1980. One can observe that physical memory has become exponentially cheaper over the years. This has enabled installed physical memory in a system to grow from few megabytes to a few gigabytes and now even to a few terabytes. Indeed, HP's DL980 server currently ships with up to 4TB of physical memory and Windows Server 2012 supports 4TB memories, up from 64GB a decade ago. Not only can modern computer systems have terabytes of physical memory but the emerging big memory workloads also need to access terabytes of memory at low latency. In the enterprise space, the size of the largest data warehouse has been increasing at a cumulative annual growth rate of 173 percent — significantly faster than Moore's law [77]. Thus modern systems need to efficiently translate addresses for terabytes of memory. This ever-growing memory sizes stretches current address-translation mechanisms to new limits.

Unfortunately, unlike the exponential growth in the installed physical memory capacity, the size of the TLB has hardly scaled over the decades. The TLB plays a critical role to enable efficient address translation by caching recently used address translation entries. A miss in the TLB can take several memory accesses (e.g., up to 4 memory access in x86-64) and may incur 100s of cycles to service. Table 1-1 shows the number of L1-DTLB (level 1 data TLB) entries per core in different Intel processors over the years. The number of TLB entries has grown from 72 entries in Intel's Pentium III (1999) processors to 100 entries in Ivy Bridge processors (2012). L1-DTLB sizes are hard to scale since L1-TLBs are accessed on each memory reference and thus need to abide by strict latency and power budgets. While modern processors have added second level TLBs (L2-TLB) to reduce performance penalty on L1-TLB misses, recent research

Table 1-1. L1-Data-TLB sizes in Intel processors over the years.

Year	1999	2001	2008	2012
L1-DTLB entries	72 (Pentium III)	64 (Pentium 4)	96 (Nehalem)	100 (Ivy Bridge)

suggests that there is still considerable overhead due to misses in L1-TLB that hit in the L2-TLB [59]. Large pages that map larger amounts of memory with a single TLB entry can help reduce the number of TLB misses. However, efficient use of large pages remains challenging [69,87]. Furthermore, my experiments show that even with use of large pages, a double-digit percentage of execution cycles can still be wasted in address translation. Further, like any cache design, the TLB needs access locality to be effective. However, many emerging big data workloads like graph analytics or data streaming applications demonstrate low access locality and thus current TLB mechanisms may be less suitable for many future workloads [77].

In summary, the every-increasing size of memory, growing data footprint of workloads, slow scaling of TLBs and low access locality of emerging workloads leads to an ever-increasing address translation overhead of page-based virtual memory. For example, my experiments on an Intel Sandy Bridge machine showed that up to 51% of the execution cycles could be wasted in address translation for the graph-analytics workloads *graph500* [36].

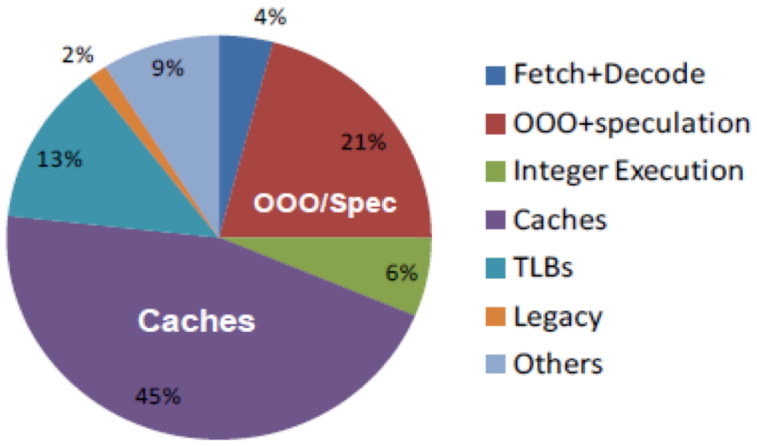


Figure 1-2. TLB power contribution to on-chip power budget. Data from Avinash Sodani's (Intel) MICRO 2011 Keynote talk.

New Design Constraint: Power dissipation is a first-class design constraint today. It was hardly the case when the virtual memory subsystems were first designed. The current focus on energy efficiency motivates reexamining processor design decisions from the previous performance-first era, including the crucial virtual memory subsystem.

Virtual memory's address translation mechanism, especially the TLB accesses, can contribute significantly to the power usage of a processor. Figure 1-2 shows the breakdown of power dissipation of a core (including caches) as reported by Intel [83]. TLBs can account up to 13% of the core's power budget. My own experiments find that 6.6-13% of on-chip cache-hierarchy's dynamic energy is attributed to TLB accesses. Further, TLBs also show up as a hotspot due to high energy density [75]. The primary reason behind the substantial energy budget of TLB's is frequent accesses to TLB. Most, if not all, commercial general-purpose processors

today access caches using physical addresses. Thus every memory reference needs to complete a TLB access before the memory access is completed. Furthermore, since a TLB is on the critical path of every memory access, fast and thus often energy-hungry transistors are used to design TLBs.

The energy dissipation is further exacerbated by the designs from performance-first era that hide TLB lookup latency from the critical path of the memory accesses. They do so by accessing the TLB in parallel to indexing into a set-associative L1 cache with page offset of the virtual address. The TLB output is used only during the tag comparison at the L1 cache. However, such a virtually indexed physically tagged cache design requires that the page offset be part of L1 cache indexing bits – forcing the L1 cache to be more highly associative than required for low cache miss rates. For example, a typical 32KB L1 cache needs to be at least 8-way set-associative to satisfy this constrain with 4KB pages. Each access to a higher-associativity structure burns more energy and thus ultimately adds to the power consumption.

In summary, many aspects of current virtual memory's address translation mechanisms warrant a fresh cost-benefit analysis considering energy dissipation as a first-class design constraint.

Proposals: In this thesis I aim to reduce the latency and the energy overheads virtual memory's address translation primarily through three pieces of work. First, I propose *direct segments* [8] to reduce TLB miss overheads for big memory workloads. Second, I propose *opportunistic virtual caching* [9] to reduce address translation energy. Finally, I also propose a *merged-associative TLB*, which aims to improve TLB designs for large page sizes by eliminating

performance unpredictability with use of large pages in commercially prevalent TLB designs. In the following, I briefly describe these three works.

1. Direct Segments (Chapter 3): I find that emerging big-memory workloads like in-memory object-caches, graph analytics, databases and some HPC workloads incur high address-translation overheads in conventional page-based virtual memory (paging) and this overhead primarily stems from TLB misses. For example, on a test machine with 96GB physical memory, graph500 [36] spends 51% of execution cycles servicing TLB misses with 4KB pages and 10% of execution cycles with 2 MB large pages. Future big-memory-workload trends like ever-growing memory footprint and low access locality are likely to worsen this further.

My memory-usage analysis of a few representative big memory workloads revealed that despite the cost of address translation, many key features of paging, such as swapping, fine-grain page protection, and external-fragmentation minimization, are not necessary for *most of their memory usage*. For example, databases carefully size their buffer pool according to the installed physical memory and thus rarely swap it. I find that only a small fraction of memory allocations, like those for memory-mapped files and executable code, benefit from page-based virtual memory. Unfortunately, current systems enforce page-based virtual memory for all memory, irrespective of its usage, and incur page-based address translation cost for all memory accesses.

To address this mismatch between the big-memory workloads' needs, what the systems support, and the high cost of address translation, I propose that processors support two types of address translation for non-overlapping regions of a process's virtual address space: 1) conventional paging using of TLB, page table walker etc., 2) a new fast translation mechanism

that uses a simple form of segmentation (without paging) called a *direct segment*. Direct segment hardware can map an arbitrarily large contiguous range of virtual addresses having uniform access permissions to a contiguous physical address range with a small, fixed hardware: *base, limit* and *offset* registers for each core (or context). If a virtual address is between the base and limit register values then the corresponding physical address is *calculated* by adding the value of the offset register to the virtual address. Since addresses translated using a direct segment need no TLB lookup; no TLB miss is possible. Virtual addresses outside the direct segment's range are mapped using conventional paging through TLBs and are useful for memory allocations that benefit from page-based virtual memory. The OS then provides a software abstraction for direct segment to the applications – called a *primary region*. The primary region captures memory usage that may not benefit from paging in a contiguous virtual address range, and thus could be mapped using direct segment.

My results show that direct segments can often eliminate 99% of TLB misses across most of the big memory workloads to reduce time wasted on TLB misses to 0.5% of execution cycles.

2. Opportunistic Virtual Caching (Chapter 4): I proposed Opportunistic Virtual Caching (OVC) to reduce energy dissipated due to address translation. I find that looking up TLBs on each memory access can account for 7-13% of the dynamic energy dissipation of whole on-chip memory hierarchy. Further, the L1 cache energy dissipation is exacerbated by designs that hides TLB lookup latency from the critical path.

OVC addresses this energy wastage by enabling energy-efficient virtual caching as *a dynamic optimization under software control*. The OVC hardware allows some of the memory

blocks be cached in the L1 cache with virtual addresses (virtual caching) to avoid energy-hungry TLB lookups on L1 cache hits and to lower the associativity of L1 cache lookup. The rest of the blocks can be cached using conventional physical addressing, if needed.

The OS, with optional hints from applications, determines which memory regions are conducive to virtual caching and uses virtual caching or conventional physical caching hardware accordingly. My analysis shows that many of challenges to efficient virtual caching, like inconsistencies due to read-write synonyms (different virtual addresses mapping to same physical address), occur rarely in practice. Thus, the vast majority of memory allocation can make use of energy-efficient virtual caching, while falling back to physical caching for the rest, as needed.

My evaluation shows that OVC can eliminate 94-99% of TLB lookup energy and 23% of L1 cache dynamic lookup energy.

3. Merged-Associative TLB (Chapter 5): While the proposed direct segments can eliminate most of DTLB misses for big memory workloads that often have fairly predictable memory usage and allocate most memory early in execution, it may be less suitable when there are frequent memory allocation/deallocations. In contrast, support for large pages is currently the most widely employed mechanism to reduce TLB misses and could be more flexible under frequent memory allocation/deallocation by enabling better mitigation of memory fragmentation.

In the third piece of work I try to improve large-page support in commercially prevalent chip designs like those from Intel (e.g., Ivy Bridge, Sandy Bridge) that support multiple page

sizes by providing separate sub-TLBs for each distinct page sizes (called a split-TLB design). Such a static allocation of TLB resources based on page sizes can lead to performance pathologies where use of larger pages can increase TLB miss rates and disallow TLB resource aggregation. A few competing commercial designs, like AMD's, instead employ a single fully associative TLB, which can hold entries for any page size. However, fully associative designs are often slower and more power-hungry than a set-associative one. Thus, a single set-associative TLB that can hold translations for any page size is desirable. Unfortunately, such a design is challenging as the correct index into a set-associative TLB for a given virtual address depends upon the page size of translation, which is unknown till the TLB lookup itself completes.

I proposed a *merged-associative TLB* to address this challenge by partitioning the abundant virtual address space of a 64-bit system among the page sizes instead of partitioning scarce hardware TLB resources. The OS divides a process's virtual address space into a fixed number of non-overlapping regions. Each of these regions contains memory mapped using a single page size. This allows the hardware to decipher page size by examining a few high-order bits of the virtual address even before the TLB lookup. In turn, this enables the hardware to *logically aggregate* the TLB resources for different page sizes into a larger set-associative TLB that can hold address translations for any page size. A merged-associative TLB can effectively achieve miss rates close to a fully associative TLB without actually having one and avoid performance pathologies of split-TLB design.

My experiments show that the merged-associative TLB successfully eliminates performance unpredictability possible with use of large pages in a conventional split-TLB

design. Furthermore, the merged-associative TLB could reduce the TLB miss rate by up to 45% in one of the applications studied.

Organization of the thesis: The organization of the rest of the thesis is as follows.

Chapter 2 describes the background of the virtual memory address translation mechanisms.

Chapter 3 describes the direct segments work that was published in 40th International Symposium on Computer Architecture (ISCA 2013). The content of the chapter mostly follows from the published paper but adds discussion on how direct segment could work in presence of physical page frames with permanent faults.

Chapter 4 describes the opportunistic virtual caching (OVC) work that was published in 39th International Symposium on Computer Architecture (ISCA 2012). The chapter follows the published paper for most part but adds a section on how OVC and direct segments could work together in a system.

Chapter 5 describes merged-associative TLB work, which is not yet published.

Chapter 6 concludes the thesis and describes potential future extensions to the thesis.

2

Virtual Memory Basics

In this chapter, I briefly discuss the history of evolution of virtual memory and basic mechanisms for virtual memory. While I primarily focus on the virtual memory as provided in x86-64 instruction set architecture (ISA), I also discuss virtual memory in contemporary ISAs like ARM, PowerPC, SPARC.

2.1 Before Memory Was Virtual

From the early days of electronic computing the designers recognized that fast access to large amount of storage is hard and thus computer memories must be organized hierarchically [26]. Computer memories have been commonly organized in at least two levels – "main memory" and "auxiliary memory" or storage. A program's information (code, data etc.) could be referenced only when it resides in main memory. The obvious challenge is to determine, at each moment, which information should reside in main memory and which in auxiliary memory. This problem has been widely known as *storage allocation problem*. Until late 1950s, any

program that needed to access more information than could fit in the main memory, required to contain the logic for addressing the storage allocation problem [95]. Further, to allow multiprogramming and multitasking early systems divided physical memory with special set of registers as in DEC's PDP-10 [95]. The challenges for automatic storage allocation and multiprogramming not only complicated the task of writing large programs but also made effectively sharing the main memory, a key computing resource, difficult.

2.2 Inception of Virtual Memory

As programs got more complex and more people started programming, the need to provide an *automatic management* of memory was deemed necessary to relieve programmer's burden. In 1959, researchers from University of Manchester, UK, proposed and produced first working prototype of a virtual memory system as part of Atlas system [50]. They introduced the key concept behind virtual memory – distinction between the "address" and the "memory location". The "address" would later more widely be known as virtual address, while "memory location" can be a physical or real address in the main memory or it could be a storage location as depicted in Figure 2-1. This allowed programmers to name information only by its virtual address while the system software (OS) along with the hardware is tasked to dynamically translate virtual addresses to its location in main memory or

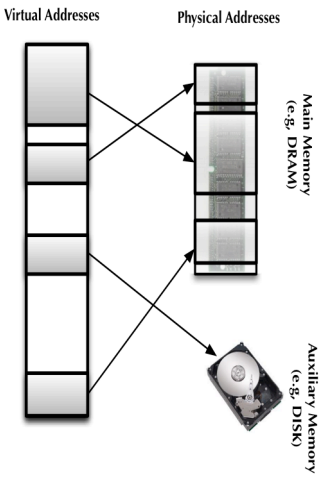


Figure 2-1. Abstract view of Virtual Memory

storage. The OS enabled automatic movement of data between the memory and storage as needed.

They key concepts and mechanisms of virtual memory were then greatly refined by the Multics project [24]. They enabled two-dimensional virtual address space that allowed ease of sharing, modularity, and protection, while efficiently managing physical memory by allocating memory in fixed sizes (called *pages*). Such a two-dimensional address space required two identifiers to uniquely identify a memory location. The working set model [25] on access locality of programs was then invented by Denning to fill in critical component in virtual memory – how to decide how much of memory to assign to a process and how to decide which information to keep in the main memory and which information in the auxiliary memory.

The rest of this chapter is organized as follows. I will first discuss a few of the important use-cases of virtual memory. I will then delve into intricacies of implementation of virtual memory management. This discussion will revolve around how virtual memory is implemented in x86-64 processors. Since these mechanisms vary considerably across different instruction-set-architectures (ISAs) I will briefly compare and contrast virtual memory management of other relevant ISAs like PowerPC, ARM-64 and UltraSPARC (T2) with that of x86-64's. Finally, I will briefly discuss what aspects of the virtual memory management this thesis addresses.

2.3 Virtual Memory Usage

While virtual memory was initially conceived to lessen the burden of the programmer to write their own storage allocation procedures, it has since enabled many use-cases. A few of them are listed below.

- *Automatic storage allocation*: Virtual memory makes it possible to run a program with memory needs in excess of the installed main memory size. This relieves the programmer from being aware of memory sizes of the system where it will execute. Modern OSes achieve this by moving data in and out between the main memory (e.g., DRAM) and the auxiliary memory (e.g., disk) to provide an illusion of larger memory than one actually available. This mechanism is commonly referred to as *swapping*.
- *Protection:* Virtual memory enables efficient fine-grain (e.g., 4KB granularity) protection of memory. The virtual-memory hardware can enforce access restriction to parts of memory by processes. This is useful for avoiding unwanted overwrite of read-only information (e.g., code). It also enables many widely used optimization like copy-on-write whereby a memory location(s) can be shared among multiple processes until at least one of the processes tries to write memory location. Modern systems, like x86-64, also enable "no-execute" protection to disallow execution of certain memory allocation to enhance security against malicious code snippets.
- *Demand paging*: Virtual memory allows bringing in information (data, code, etc.) from auxiliary memory (storage) to main memory only when they are first referenced. Demand

paging enables fast startup for processes by not requiring pre-loading of all the information that the process would require during its execution.

- *Relocation:* Virtual memory enables ease in software development by allowing the same virtual address in different programs to refer to different physical memory location during execution. This allows the software tool chains (e.g., compiler, linker) generate code using only virtual addresses without worrying about where the referred information is located and thus, be agnostic of a system's memory configuration. This in turn, enables programs to be built from separately compiled, reusable and sharable modules.
- *Metadata collection:* Most current virtual memory hardware enables efficiently storing small amount of metadata with virtual memory allocation units (e.g., per-page dirty and reference bits). This enables usage information collection. For example, these hardware-managed metadata are often useful for many OS procedures like page-frame-reclamation or even in garbage collection of many high level languages.
- *Sharing:* Virtual memory enables efficient sharing of code and data among multiple processes by mapping virtual addresses of generated by different processes to map on to same physical memory location. This is especially useful in sharing libraries across multiple processes.

2.4 Virtual Memory Internals

In this section, I discuss the mechanisms to translate software-generated virtual addresses to physical addresses – a key component of virtual memory subsystem.

The OS and the hardware coordinate to accomplish this address-translation process. There are broadly two techniques of address translation that were ever employed – paging and segmentation. Paging or page-based virtual memory translates a linear virtual address to a physical address. The address translation process is essentially hidden from user programs except from its performance implications. Page-based virtual memory translates the virtual addresses in one or few ISA-defined granularities (e.g., 4KB). On the other hand, segmentation allows mapping between the address spaces at variable (often arbitrary) granularity. Segmentation often exposes two a dimensional virtual address space to users where an address is identified by a segment identifier and an offset within the segment. One advantage of paging over segmentation is that paging can help mitigate external fragmentation. External fragmentation can occur if memory is allocated and de-allocated in various sizes leaving physical memory scattered in small holes that is unusable to satisfy later memory requests. Paging mitigates this by allocating and de-allocating memory in one or a few fixed sizes called page size. Segmentation on the other hand allows ease of sharing semantically related memory locations (segments) among multiple processes. In this section, I will primarily focus on paging since it is the primary address translation mechanism supported across almost all modern architectures. Further, current systems that allow segmentation, do so on top of paging.

2.4.1 Paging

While almost every commercial ISAs today implements paging, in this subsection I will discuss intricacies paging as implemented in x86-64 ISA. The hardware mechanism that is

responsible for translating software generated virtual addresses to physical address is called memory management unit or MMU. Below, I discuss various aspects of MMU.

2.4.1.1 Address Spaces

While x86-64 can theoretically support up to 64-bit long linear virtual address all current implementations support up to 48 bits [40]. The address bits between 63^{th} and 48^{th} are sign extended. Thus current generation of x86-64 processors can refer up to 256TB of virtual addresses. Most commercial operating systems, such as Linux, Windows, export separate virtual address spaces for each process. Linux, for example, allows 128TB of virtual address space for each user process while allows another 128TB for the kernel.

The size of the physical address space determines the maximum size of the main memory that can be installed in a given system. Currently, x86-64 ISA supports up to 52 bits of physical address but unlike virtual address, the physical address size is a micro-architectural (rather than architectural) property and varies across processor models. A more typical 40-46 bit physical addresses can support up to 64TB installed physical memory capacity. Note that there is only one physical address space for a system, while each process in a system can have its own virtual address space.

2.4.1.2 Translating Virtual Addresses

The primary function of the virtual memory subsystem is to translate an address in a virtual address space to an address in the physical address space. In the following subsection, I will describe the key components and mechanism of this address translation process.

Page table: The page table is the per-process in-memory data-structure that holds the translation information from a linear virtual address space to physical address space. The page table stores the translation information at the granularity of ISA-defined page sizes. The default page size in x86-64 is 4KB, while a couple of larger page sizes are supported (discussed later in the section). The virtual address is divided in to two parts – virtual page number (VPN) and page offset. The page offset identifies the byte address within a page. For example, with 4KB page size the lower 12 bits of the virtual address. The rest of higher address bits constitute the virtual page number. The physical memory is similarly divided into physical page frames at the page-size grain and is identified by page frame number. The page offset remains unaltered between a virtual address and its corresponding physical address. Thus the page table translates a virtual page number to a page frame number. An entry of page table or page table entry (PTE) contain information such as whether the page is in memory, corresponding page frame number or location on the auxiliary memory (valid/present bit), access rights to the page (read, read-write etc.), whether the page belongs to supervisor mode (OS), reference and dirty bit. The size of each PTE in x86-64 is 8 bytes. In x86-64, the page table structure itself is accessed using physical address only.

The x86-64 ISA defines the page-table data structure as a four-level radix tree with fan out of 512 at each level. Such a hierarchical page table structure allows space-efficient representation of large potentially sparse virtual address space. The last level of the page table contains PTE that contains the physical frame number.

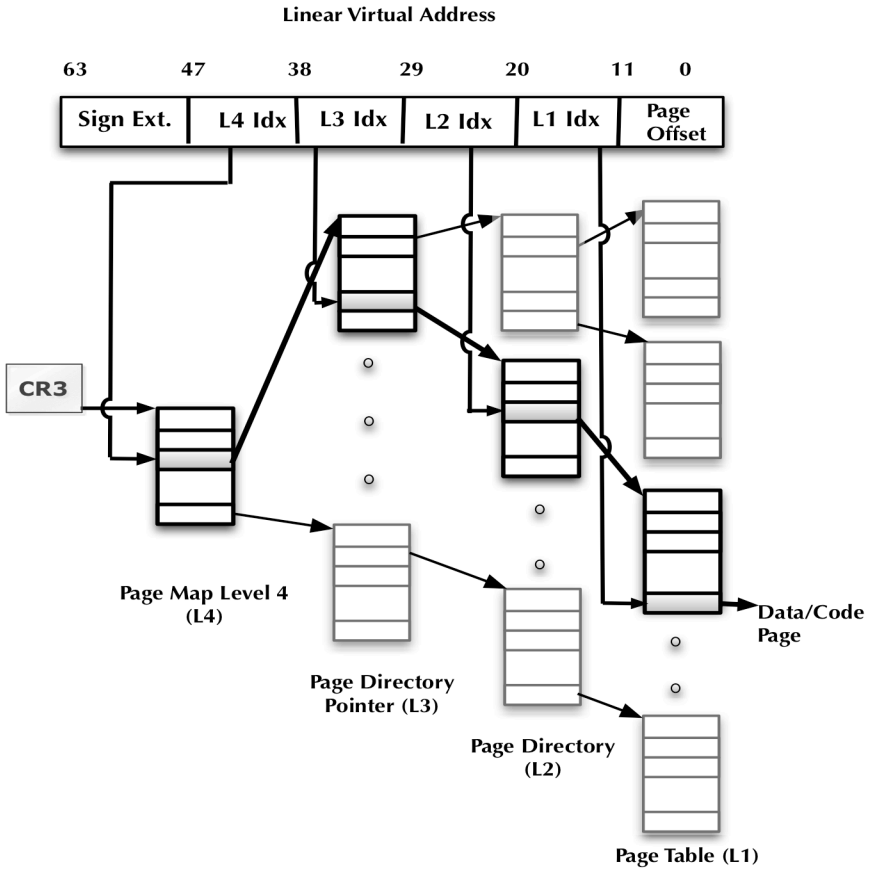


Figure 2-2. Page table walk in x86-64 with 4KB page size.

Page Table Walk: In Figure 2-2, I depict the structure of four-level page table for a process in x86-64. The figure also depicts how the page table is searched to find the desired PTE corresponding to a given linear virtual address. This process is called a page table walk. The size of each node in the tree structure is 4KB and the size of each entry in the node is 8-bytes wide. Thus, at each node of the page table there can be up to 512 entries (i.e., fan out of 512). I number the address bits such that the most significant bit is the 63^{th} bit of the virtual address and 0^{th} bit is the lowest significant bit.

In x86-64, a hardware page-table-walker walks the in-memory page table. First, the page-table-walker needs to know the location of root of the page-table in the memory. In x86-64, a register named **CR3** holds the *physical page frame number* (40 bits) of the root of the page table data structure of the currently running process [40]. The OS updates the value of CR3 when switching process contexts, as the location of the page table is part of a process's context. The least significant 12 bits of the virtual address constituting the page offset for 4KB pages remain unaltered across the virtual and physical address. The remaining higher-order 36-bits are divided in to four 9-bit indices used to select entries from four levels of the page table as shown in Figure 2-2. The four levels of page tables are named Page Map Level 4 (PML4 or L4), Page Directory Pointer (PDP or L3), Page Directory (PD or L2) and Page Table (PT or L1). In the first step, the value of CR3 register is used to locate the 4KB-aligned starting physical address of the PML4 of the page table. As depicted in the Figure 2-2, 9 bits from bit position 39^{th} to 47^{th} are extracted from the virtual address and appended to the value of CR3 (40 bits of physical address). This yields 49-bit physical address of the appropriate L4 entry. The L4 entry could contain the physical frame number of the page containing the corresponding L3 node. The process is repeated by extracting the L3 index field from virtual address and appending it to the physical page frame number obtained from the L4 node to locate the correct L3 entry. This recursive process continues until either a selected entry is invalid or the desired PTE at the L1 node is found. The PTE at L1 contains the desired physical page frame number corresponding to the given virtual page number. The page offset can then be appended to form the full physical address of the given virtual address. Such a page-table "walk" procedure can require up to four memory references to find the physical page frame number for a given virtual page number. As

the address space grows, more levels would need to be added, further increasing the number of memory references for a page table walk. A full 64-bit virtual address space could require up to six levels. However, often off-chip memory accesses are avoided as the page table entries could be cached along with normal data blocks in on-chip processor caches (e.g., L1, L2 and/or L3 caches).

Page fault: The page table can be sparsely populated. At any level if there are no valid entry corresponding to the given virtual address index, the sub-tree beneath that index is not instantiated. This yields significant memory space savings for a sparsely allocated virtual address space. An exception is raised to the OS if a valid entry is not found at any of the levels of the page table during the walk. Such a hardware exception is called a *page-fault*. The x86-64 hardware provides the linear virtual address that caused the page fault, called the faulting address, in the control register called CR2. The operating system then takes appropriate actions and may install mapping for the faulting address. OS can also raise exceptions to application if it deems necessary, e.g., a segmentation fault is raised if an unmapped address is accessed. Page-faults could also be triggered access rights violation (e.g., a write access to a read-only address) or if the desired page has been swapped out to the auxiliary memory (disk). While generating page-fault exception the hardware also provides related information such as whether the faulting address is instruction or data, whether the fault is due to access rights violation or due to invalid mapping information.

2.4.1.3 Making Address Translation Faster

TLB: Software accesses memory using virtual addresses that need to be translated to physical addresses. However, the aforementioned address translation process is too slow to be carried out on each memory access. To speed up this process, translation lookaside buffer or TLB was introduced. A TLB is a hardware cache of recently used address translation entries to avoid long-latency page-walk on every memory access. Each TLB entry contains a virtual page number and its corresponding physical page frame number. A TLB entry also contains related metadata information like access rights, super-user bits etc. On each memory access the TLB is accessed with the virtual page number. If it hits in the TLB then the physical page frame number is immediately available. No page-table-walk is necessary. However, in case of a TLB miss the hardware page-table walker looks up the page-table which could take up to four memory accesses as described in the previous sub-section.

Large Pages: The TLB structure needs to be fast and energy efficient as it is looked up on every memory access. This makes it hard to increase the number of TLB entries. To translate larger amount of memory without requiring more TLB entries, hardware designers introduced larger page sizes that helps increasing *TLB reach* (number of TLB entries × page size). Larger TLB reach can potentially lower the number of TLB misses. Currently, x86-64 supports two larger page sizes – 2MB and 1GB beyond the default 4KB page size. A single TLB entry for larger page sizes can translate 2MB or 1GB of contiguous virtual address to the corresponding amount of physical address. Thus, compared to the base page size, larger page sizes can increase the TLB reach with same number of TLB entries. Furthermore, larger page sizes shorten the

page walk procedure described in Figure 2-2. For example, an entry in the L2 node (page directory) could directly point to physical frame number of a 2MB page. Thus an address mapped using 2MB pages needs up to 3 memory references to find the corresponding physical page frame number. Similarly, page walk could stop at a L3 node (page directory pointer) for a 1GB page, requiring only up to two memory accesses. In summary, large pages can potentially lower the number of TLB misses as well as reduce the number of memory references needed for a page walk.

I observe that each page size in x86-64 is 512 times of its nearest size. This is an artifact of the x86-64's page table organization as radix tree with fan-out of 512. Radix-tree nodes at each level of the tree represent different a page size – L1 nodes represent 4KB pages, L2 nodes represents 2MB pages and L3 nodes represent 1GB pages. Going by this trend, it can be assumed that in future next larger page size could be 512GB.

A typical per-core TLB hierarchy with multiple page sizes in recent Intel's recent Ivy Bridge processors has two levels of TLBs. There are separate L1 TLBs for instruction and data. The L1 instruction TLB holds 128 entries for 4KB pages in an 8-way set-associative structure, while there are 8 entries for 2MB pages. The L1-data TLB has three sub-TLBs – 64 entry 4-way set-associative TLB for 4KB pages, 32 entry 4-way set-associative TLB for 2MB pages and 4-entry fully associative TLB for 1GB pages. The second level (L2) TLB can hold translations for both instruction and data. The L2-TLB can hold 512 4KB page translation and is a 4-way set-associative structure. There is no L2-TLB for larger page sizes. A miss in L1-TLB that hits in L2-TLB can incur a delay of 7 processor cycles [39].

Paging structure caches (PSC): A TLB miss triggers a long-latency hardware page table walk. To reduce this cost, the hardware designers have introduced a cache for holding only page table information, called a page-structure cache (Intel terminology). The key observation is that the accesses to upper levels of page table demonstrate significant temporal locality. For example, page table walks to two consecutive virtual address pages will likely to have same three upper level entries since indices selecting these entries are extracted from higher order bits of virtual address that rarely change. If the higher-level page table entries could be cached then the later page walks could be serviced faster, requiring less number of memory accesses. Thus the x86-64 vendors have developed caches to store recently used upper level page table entries [3,6,41]. However, their designs differ significantly. In this subsection, I primarily discuss Intel's designs of translation caches and briefly contrast it with AMD's design.

Like TLBs, PSCs are accessed using virtual addresses. However, while the TLBs are indexed and/or tagged by complete virtual page number, the PSCs are looked up using only a *partial prefix of the virtual page number*. More specifically, virtual address prefixes for different levels of the page table is used for accessing PSCs. For example, there can be a PSC entry for the L4 index (bits 39 to 47) of a virtual address (refer to Figure 2-3). Such an entry would provide the physical page frame number containing the corresponding node in the next level of the page table (here L3 node). Similarly, a PSC entry for bits 30 to 47 (L4 index appended by L3 index) of a virtual address would contain physical frame number containing the corresponding L2 node of the page table. An entry for bits 21 to 47 (L4 index appended by L3 index appended by L2 index) would contain physical frame number of desired PTE. I observe that for a given virtual

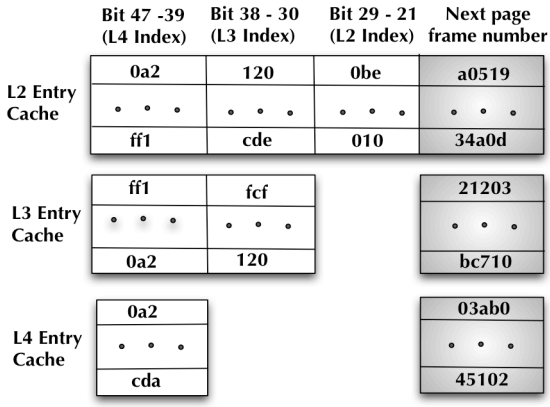


Figure 2-3. Paging-structure caches. The virtual address prefixes are un-shaded. The physical page frame number of next level in the page table is in shade.

page number there could be up to three prefixes that could match in the translation cache. Intel's page-structure caches employ a split design where different levels of page table entries corresponding to three different prefix lengths of the virtual address are cached in separate caches to avoid interference. Such a split design could be helpful since different level of page table entries demonstrate very different access locality. Figure 2-3 shows a typical page-structure cache configuration as found in Intel's recent processors. The number in the un-shaded boxes are prefixes of virtual addresses while the numbers in shaded represents physical frame number. The physical frame number represents the address of the page containing the desired PTE for a L2 entry, while it is the page frame number for node containing desired L2 node and L3 node for L3 and L4 entry, respectively.

On a TLB miss, the hardware page table walker firsts looks up the translation cache with the virtual page number of the missing address. The lookup can produce up to three hits, one each for three sub caches of three levels of the page table. The page table walker then selects an entry with longest prefix match. A hit in the sub-cache for L2 page table entries can reduce the

number memory access on TLB miss to one, instead of four as originally required. Hits for L3 and L4 entry would require two and three memory references, respectively.

Different from Intel's design, AMD's chips implement a simpler design called page-walk cache for caching higher-level page table information. The entries are cached in the page-walk cache with their physical addresses, unlike partial prefix of virtual address for Intel's page-structure caches. AMD's page-walk cache can be deemed simply as a specialized data cache that can hold only page table entries. In the ideal case, all three upper level page table entries could be cached and three sequential lookups of the page walk cache can provide with the physical page frame number of the node containing the desired PTE. Thereafter a single memory reference provides the desired address translation. Further, unlike Intel's split design, AMD's page walk caches employ a unified design where all entries from all three upper levels of page-table compete for a place in the cache.

2.4.1.4 Putting it altogether

In this section, I discuss how various components of x86-64's page based virtual management work together to translate a linear virtual address generated by the software to a physical address.

Figure 2-4 depicts various components of x86-64's virtual memory management and their interactions. The paged virtual memory management component in x86-64 is responsible for translating a software-generated 48-bit wide virtual address (VA) to 52-bit wide physical address (PA). In the following I describe the steps in that address translation process as it is done in most Intel's current processors.

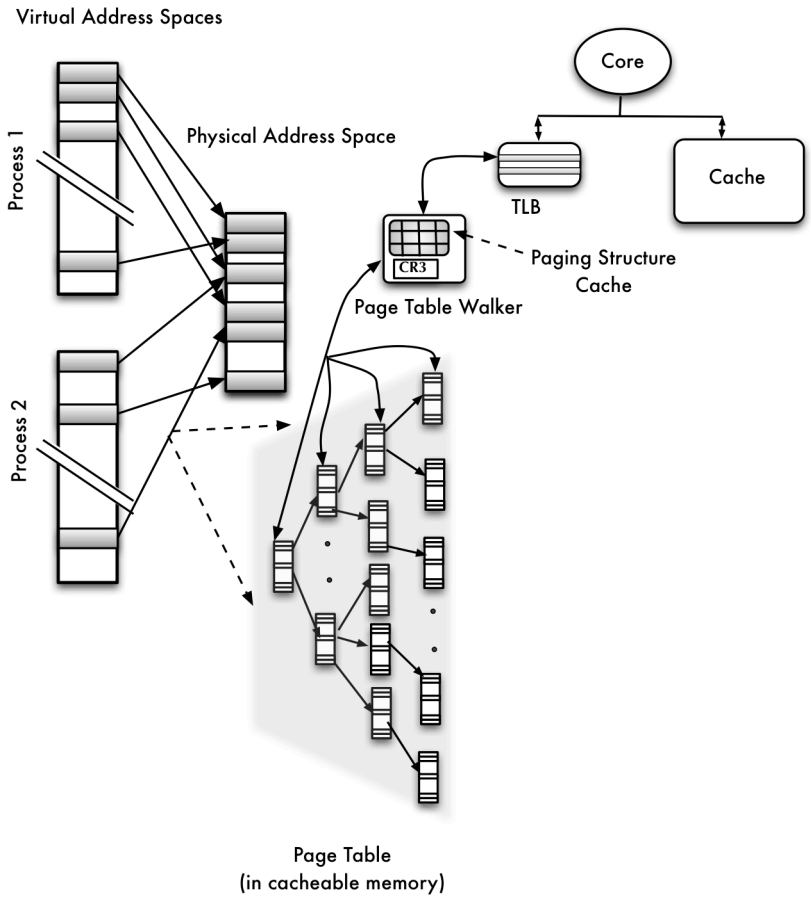


Figure 2-4. Schematic diagram of key components of virtual memory management. (Not to scale)

TLB: The virtual page number is first searched in the TLB hierarchy. A hit in the TLB yields the desired physical page frame number along with the access rights. If the access rights are sufficient for a given access then the page offset is appended to the physical page frame number to obtain the desired physical address.

Paging-structure Cache (PSC): On a miss in TLB hierarchy, the PSC accompanying the hardware page table walker is looked up using prefixes of the virtual page number. As described in Section 2.4.1.3, page-structure caches can help skipping one or more levels of the

four levels of hierarchical page table. On a hit in the paging structure cache it could require one to three memory references for the partial page walk to find the desired PTE.

Page walk: On a miss in the PSC a complete page walk of in-memory page table data-structure is initiated. The CR3 register provides the physical address of the root of the page table for the given process. The hardware page-table-walker then walks the four-level radix tree structure of the page table that could take up to four off-chip memory accesses. The page table entries reside in normal cacheable memory and thus hardware page walker first accesses the on-chip cache hierarchy before accessing main memory.

Page fault: If the hardware page-table-walker finds no valid entry in the page table corresponding to the given virtual address or if entry lacks sufficient access rights, it raises page-fault exception to the OS. On a page-fault the OS can create the missing PTE or raise exception to user program.

2.4.2 Segmentation

While previous section discussed the intricacies of address translation with paging, in this section I discuss segmentation.

Segmentation exports the virtual address space as a set of variable-sized sections or segments. Different segments usually represent distinct natural divisions of program like stack region, code region, data region, etc., and thus visible to the programmer or application. Segmentation usually exports a two-dimensional address space as opposed to linear address space provided by paging. A segment identifier identifies each segment. Thus a virtual address

in segmented address space is a pair **<*s*, *o*>**, where ***s*** is the segment identifier or segment selector and ***o*** is the offset value within the segment. The offset value starts with value 0 and is bounded by the size of the segment minus one. The segment information is stored in the segment table where each entry contains segment selector, base, limit, protection information and a valid bit. To translate a segmented address the segment table is accessed with the segment identifier to find the physical base address and the limit value. If the segment offset (***o***) of the given virtual address is within the limit of the selected segment and the protection information allows the requested operation then the offset value (***o***) is added to the base value of the segment to produce a linear address. The content of the segment table are often cached in a few ISA-defined hardware segment registers to fasten the look up of the segment table.

Modern system that implements segments (e.g., PowerPC and x86), do so on top of paging. Thus the linear address generated by segmentation is further translated to the physical address via paging hardware. However, some very early system like Burrough's B5000 [96] did not use paging and used only segmentation.

Segmentation is useful in sharing of a semantically related memory region among multiple processes. It also helps by allowing each segment to relocated and grown or shrunk independently of other segments. However, due to variability in the segment size, a pure segmented architecture (without paging) can lead to external fragmentation whereby physical memory can be scattered in small unusable holes. Thus most commercial systems lay out segmentation on top of paging. In the following I briefly describe x86's segmentation and its fate in x86-64 architecture.

2.4.2.5 Segmentation in x86 (32-bits) and x86-64

The 32-bit x86 architecture allows two modes of operation – the real mode and the protected mode. Real mode of operation is used only during early boot process to initialize hardware and allows only a 20-bit address space (limited to 1MB). In the real mode, x86 implements pure segmentation without paging and each segment is always 64KB long.

During normal operation, an x86 system runs in protected mode. The virtual address consists of 16-bit segment selector (or segment identifier) and a 32-bit of segment offset. The segment selector needs to be in one of the hardware segment registers. The x86 ISA exposes six segment registers: CS, SS, DS, ES, FS, GS. During instruction fetch, the CS or code segment register is selected by default, while SS (stack register) is selected for stack reference. DS register is selected by default for data accesses. However, data access can also use ES, FS or GS by explicit referring them in the program binary. The upper 13 bits of the 16-bit long segment selector read from the relevant segment register are called segment number and is used to index into one of the two segment descriptor tables. There is a local descriptor table (LDT) and global descriptor table (GDT), and a bit in the segment selector decides which of these two segment descriptor tables to index into. In general, segments private to a process are kept in the LDT while global segments reside in the GDT. The segment descriptor entries contain the segment's linear base address as well as limit and protection information. Segment information is cached along with the segment selector in the segment registers for fast lookup. The segment base address is then added to the segment offset to create the linear virtual address.

Table 2-1. Comparison of key virtual memory features across different architectures.

	PowerPC (Power 7)	UltraSPARC (T2)	ARM-64	x86-64
Virtual Address Space	80-bit segmented-address space on top of paging	64-bits paged-address space. Lower 44 bits used.	64-bits paged-address space. Lower 48 bits used.	64-bits paged-address space. Lower 48 bits used.
Page Table	Inverted hashed page table.	OS-defined flexible structure.	Four-level hierarchical.	Four-level hierarchical.
TLB Miss Handling	H/W page table walker.	H/W looks up TSB. On TSB miss S/W handler walks page table.	H/W page table walker.	H/W page table walker.
Page Sizes	4KB, 64KB, 16MB and 16GB.	8KB, 64KB, 4MB, and 256MB.	4KB and 64KB.	4KB, 2MB and 1GB.

The x86-64 essentially uses flat model without much support for segmentation where CS, SS, DS, ES registers are hardwired to zero. FS and GS registers can still have non-zero base addresses and are sometimes used by modern systems for holding some thread-private information. Thus, in today's x86-64 systems paging is the primary address translation mechanism.

2.4.3 Virtual Memory for other ISAs

Hitherto, I have discussed the virtual memory mechanisms as in the x86-64 architecture. However, other ISAs like PowerPC, ARM-64 and SPARC have some important differences in the way they handle virtual memory. In this subsection I will provide a brief discussion on how these other ISAs differ from x86-64.

Table 2-1 provides brief contrast of key virtual memory features of various different ISA with those of x86-64. I discuss more details about these features and ISA in the following.

PowerPC: 64-bit Power-PC architecture exposes 80-bit wide segmented virtual address space, which like x86, is implemented on top of paging. Each segments in PowerPC (Power7) is 256 MB or 1TB in size [82]. It uses the top 36 bits of a virtual address to index into a segment descriptor table whose entries can be cached in hardware in a segment-lookaside-buffer.

PowerPC implements a hashed inverted page table, which is an eight-way set-associative software cache structure. On a TLB miss the hardware computes a hash of the virtual address to index into the inverted page table. Due to possible conflict in the hash table the OS also maintains the complete page table in memory. PowerPC supports 4KB, 64KB, 16MB and 16GB page sizes.

ARM-64: ARM supports a flat virtual address space without segmentation. It is very similar to x86-64 and provides a 48-bit virtual address space with kernel occupying the upper half and the user processes the lower half. It implements a four-level hierarchical page table similar to x86-64 [4]. ARM supports 4KB and 64KB page sizes in 64-bit mode. Use of 64KB page size limits the depth of hierarchical page table to two instead of four and restricts the virtual address width to 42 bits. Since use of larger page size alters the page table structure and limits address space size, it is likely that ARM-64 is unable to support mix of page sizes for a given process.

UltraSPARC T2: While SPARC supports a 64-bit address space only the lower 44-bits are used for addressing. The physical address space size is generally 41 bits. The UltraSPARC implements a software-defined page table completely under OS control. It also defines a Translation Storage Buffer (TSB) that resides in system memory. It is a virtually indexed, virtually tagged, direct-mapped software cache of the translation entry. On a TLB miss the hardware first searches the TSB. On a TSB miss an exception is raised to OS to fill the TLB. It supports 8KB, 64KB, 4MB, and 256MB page sizes.

2.5 In this Thesis

In this thesis, I focus primarily on two key problems of current virtual memory implementation: 1) excessive energy dissipation due to frequent TLB lookups and 2) performance overhead due to TLB misses. While rest of the thesis will focus around designs that are employed by x86-64 architectures, the above-mentioned two key challenges of virtual memory are universal across different architectures. For example, to the best of my knowledge all current commercial general-purpose processors looks up the TLB on each memory access and the performance overhead due to TLB misses are widely accepted challenge across architectures other than x86-64 [88,103]. Further, I believe that the techniques proposed in this thesis are applicable irrespective of ISA. In the third piece, I focus on the design trade-offs in TLB implementation if multiple page sizes are to be supported. Our proposal could benefit TLB design employed by recent commercially popular Intel chips.

3

Reducing Address Translation Latency

3.1 Introduction

In this chapter I discuss the *direct segments* proposal that strives to eliminate most of the address translation cost for emerging big memory workloads.

The primary cost of address translation in page-based virtual memory stems from indirection: on each access to virtual memory, a processor must translate the virtual address to a physical address. While address translation can be accelerated by TLB hits, misses are costly, taking up to 10s-100s of cycles [9,83].

To reevaluate the address translation cost and the benefits of decades-old page-based virtual memory in today's context, I focus on an important class of emerging *big-memory* workloads. These include memory intensive "big data" workloads such as databases, key-value stores, and graph algorithms as well as high-performance computing (HPC) workloads with large memory requirements.

My experiments reveal that these big-memory workloads incur high virtual memory overheads due to TLB misses. For example, on a test machine with 96GB physical memory,

graph500 [36] spends 51% of execution cycles servicing TLB misses with 4KB pages and 10% of execution cycles with 2 MB large pages. The combination of application trends—large memory footprint and lower reference locality [31,71]—contributes to high TLB miss rates and consequently I expect even higher address translation overheads in future. Moreover, the trends to larger physical memory sizes and byte-addressable access to storage class memory [77,91] increase address mapping pressure.

Despite these costs, I find big-memory workloads seldom use the rich features of page-based virtual memory (e.g., swapping, copy-on-write and per-page protection). These workloads typically allocate most of the memory at startup in large chunks with uniform access permission. Furthermore, latency-critical workloads also often run on servers with physical memory sized to the workload needs and thus rarely swap. For example, databases carefully size their buffer pool according to the installed physical memory. Similarly, key-value stores such as *memcached* request large amounts of memory at startup and then self-manage it for caching. I find that only a small fraction of memory uses per-page protection for mapping files, for executable code. Nevertheless, current designs apply page-based virtual memory for *all* memory regions, incurring its cost on *every* memory access.

In light of the high cost of page-based virtual memory and its significant mismatch to "big-memory" application needs, I propose mapping part of a process's linear virtual address with a *direct segment* rather than pages. A direct segment maps a large range of contiguous virtual memory addresses to contiguous physical memory addresses using small, fixed hardware: *base, limit* and *offset* registers for each core (or hardware context with multithreading). If a

virtual address V is between the base and limit ($base \leq V < limit$), it is translated to physical address $V + offset$ *without* the possibility of a TLB miss. Addresses within the segment must use the uniform access permissions and reside in physical memory. Virtual addresses outside the segment's range are translated through conventional page-based virtual memory using TLB and its supporting mechanisms (e.g., hardware page-table-walker).

The expected use of a direct segment is to map the large amount of virtual memory that big-memory workloads often allocate considering the size of physical memory. The software abstraction for this memory is called a *primary region,* and examples include database buffer pools and in-memory key-value stores.

Virtual memory outside a direct segment uses conventional paging to provide backward compatibility for swapping, copy-on-write, etc. To facilitate this, direct segments begin and end on a base-page-sized boundary (e.g., 4KB), and may dynamically shrink (to zero) or grow (to near physical memory size). While doing so may incur data-movement costs, benefits can still accrue for long-running programs.

Compared to past segmentation designs, direct segments have three important differences. It (a) retains a standard linear virtual address space, (b) is not overlaid on top of paging, and (c) co-exists with paging of other virtual addresses. Compared to large-page designs, direct segments are a one-time fixed-cost solution for any size memory. In contrast, the size of large pages and/or TLB hierarchy must grow with memory sizes and requires substantial architecture, and/or operating system and applications changes. Moreover, being a cache, TLBs

rely on access locality to be effective. In comparison, direct segments can map arbitrarily large memory sizes with a small fixed-size hardware addition.

The primary contributions of this chapter of the dissertation are:

- I analyze memory usage and execution characteristics of big-memory workloads, and show why page-based virtual memory provides little benefit and high cost for much of memory usage.
- I propose *direct-segment* hardware for efficient mapping of application-specified large *primary regions*, while retaining full compatibility with standard paging.
- I demonstrate the correctness, ease-of-use and performance/efficiency benefits of the direct segments proposal.

3.2 Big Memory Workload Analysis

I begin with a study of important workloads with big-memory footprints to characterize common usage of virtual memory and identify opportunities for efficient implementations. The study includes the following three aspects:

- *Use of virtual memory*: I study what virtual memory functionalities are used by such big-memory workloads;

Table 3-1. Test machine configuration.

	Description
Processor	Dual-socket Intel Xeon E5-2430 (Sandy Bridge), 6 cores/socket, 2 threads/core, 2.2 GHz
L1 DTLB	4KB pages: 64-entry, 4-way associative; 2MB pages: 32-entry 4-way associative; 1GB pages: 4-entry fully associative
L1 ITLB	4KB pages: 128-entry, 4-way associative; 2MB pages: 8-entry, fully associative
L2 TLB (D/I)	4 KB pages: 512-entry, 4-way associative
Memory	96 GB DDR3 1066MHz
OS	Linux (kernel version 2.6.32)

- *Cost of virtual memory*: I measure the overhead of TLB misses with conventional page-based virtual memory;
- *Execution environment*: I study common characteristics of the execution environment of big-memory workloads.

Table 3-1 describes the test machine for the experiments.

Table 3-2 describes the workloads used in the study. These applications represent important classes of emerging workloads, ranging from in-memory key-value stores (*memcached*), web-scale large relational databases (*MySQL*), graph analytics (*graph500*) and supercomputing (*NAS* parallel benchmark suite). *Memcached* is used by many popular websites like YouTube, Wikipedia, Flickr, and Craigslist [65]. *MySQL* is an open-source relational-database running TPCC workload. IDC estimates that relational database market to be worth nearly 34 billion USD in 2011 [42]. *graph500* is created by a group of fifty researchers and professionals from the industry and the academic to create representative workloads that accesses

Table 3-2. Workload descriptions.

graph500	Generation, compression and breadth-first search of large graphs. http://www.graph500.org/
memcached	In-memory key-value cache widely used by large websites (e.g., in Facebook).
MySQL	MySQL with InnoDB storage engine running TPC-C (2000 warehouses).
NPB/BT NPB/CG	HPC benchmarks from NAS Parallel Benchmark Suite. http:// nas.nasa.gov/publications/npb.html
GUPS	Random access benchmark defined by the High Performance Computing Challenge. http://www.sandia.gov/~sjplimp/algorithms.html

large graphs [36]. Graph algorithms, especially those on very large graphs, are gaining importance due to emergence of social media where social connections often represented as graphs. Further, I also studied the *GUPS* micro-benchmark designed by HPC community to stress-test random memory access in high-performance computing settings.

3.2.1 Actual Use of Virtual Memory

Swapping. A primary motivation behind the invention of page-based virtual memory was *automatic* management of *scarce* physical memory without programmer intervention [26]. This is achieved by swapping pages in and out between memory and secondary storage to provide the illusion of much more memory than is actually available.

I hypothesize that big-memory workloads do little or no swapping, because performance-critical applications cannot afford to wait for disk I/Os. For example, Google observes that a sub-second latency increase can reduce user traffic by 20% due to user dissatisfaction with higher latency [54]. This drives large websites such as Facebook, Google, Microsoft Bing, and Twitter

to keep their user-facing data (e.g., search indices) in memory [20]. Enterprise databases and in-memory object-caches similarly exploit buffer-pool memory to minimize I/O. These memory-bound workloads are therefore either sufficiently provisioned with physical memory for the entire dataset or carefully sized to match the physical memory capacity of the server.

I examine this hypothesis by measuring the amount of swapping in these workloads with the *vmstat* Linux utility. As expected, I observe *no swapping activity*, indicating little value in providing the capability to swap.

Memory allocation and fragmentation: Frequent allocation and de-allocation of memory in various sizes can leave holes in physical memory that prevent subsequent memory allocations, called external fragmentation. To mitigate such external fragmentation, paging uses fixed (page-sized) allocation units.

Big-memory workloads, on the other hand, rarely suffer from OS-visible fragmentation because they allocate most memory during startup and then manage that memory internally. For example, databases like *MySQL* allocate buffer-pool memory at startup and then use it as a cache, query execution scratchpad, or as buffers for intermediate results. Similarly, *memcached* allocates space for its in-memory object cache during startup, and sub-allocates the memory for different sized objects.

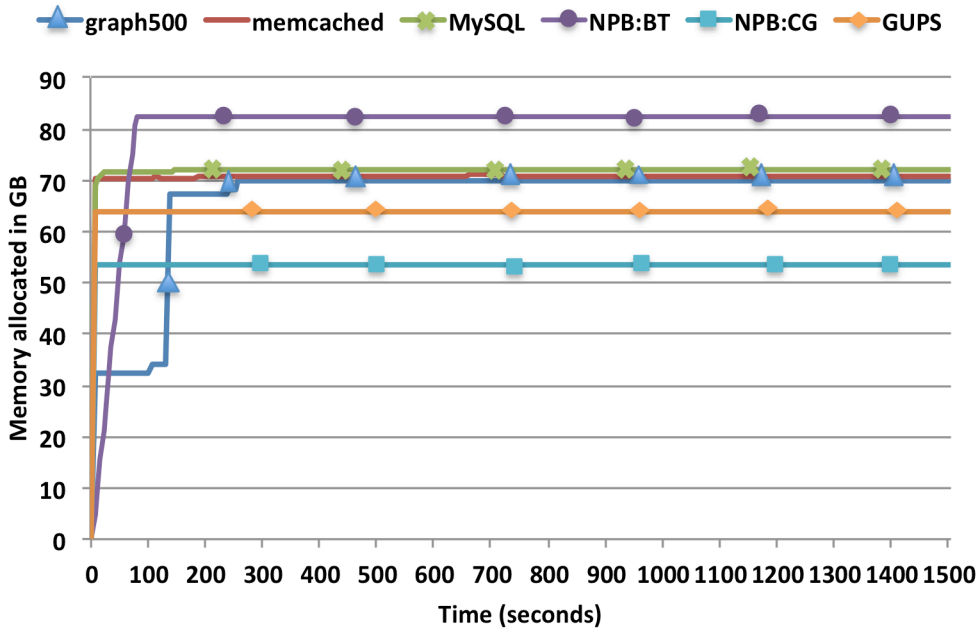


Figure 3-1. Memory allocated over time by workloads.

I corroborate this behavior by tracking the amounts of memory allocated to a workload over its runtime. I use Linux's *pmap* utility [55] to periodically collect total allocated memory size for a given process. Figure 3-1 shows the allocated memory sizes in 5-second intervals over 25 minutes of execution for each workload on the test machine described in Table 3-1.

Across the workloads, most memory is allocated early in the execution and very little variation in allocation thereafter. These data confirm that these workloads should not suffer OS-visible fragmentation. A recent trace analysis of jobs running in Google's datacenter corroborates that memory usage changes little over runtime for long-running jobs [78]. Furthermore, because memory allocation stabilizes after startup, these workloads have *predictable* memory usage.

Table 3-3. Page-grain protection statistics.

	Percentage of allocated memory with read-write permission
graph500	99.96%
memcached	99.38%
MySQL	99.94%
NPB/BT	99.97%
NPB/CG	99.97%
GUPS	99.98%

Per-page permissions. Fine-grain per-page protection is another key feature of paging. To understand how page-grain protection is used by big-memory workloads, I examine the kernel metadata for memory allocated to a given process. Specifically, I focus on one type of commonly used memory regions—*anonymous* regions (not backed by file) (excluding stack regions). Table 3-3 reports the fraction of total memory that is dynamically allocated with read-write permission over the entire runtime (averaged over measurements at 5-second intervals).

I observe that *nearly all* of the memory in these workloads is dynamically allocated memory with read-write permission. While unsurprising given that most memory comes from large dynamic allocations at program startup (e.g., *MySQL's* buffer pool, in-memory object cache), this data confirms that fine-grain per-page permissions are not necessary for more than 99% of the memory used by these workloads.

There are, however, important features enabled by page-grain protection that preclude its complete removal. Memory regions used for inter-process communication use page-grain

protection to share data/code between processes. Code regions are protected by per-page protection to avoid overwrite. Copy-on-write uses page-grain protection for efficient implementation of the *fork()* system call to lazily allocate memory when a page is modified. Invalid pages (called guard pages) are used at the end of thread stacks to protect against stack overflow. However, the targeted big-memory workloads do not require these features for *most* of the memory they allocate.

> ***Observation 1:*** For the majority of their address space, big-memory workloads do not require, swapping, fragmentation mitigation, or fine-grained protection afforded by current virtual memory implementations. They allocate memory early and have stable memory usage.

3.2.2 Cost of Virtual Memory

In this subsection, I quantify the overhead of page-based virtual memory for the big-memory workloads on *real hardware*.

Modern systems enforce page-based virtual memory for *all* memory through virtual-to-physical address translation on every memory access. To accelerate table-based address translation, processors employ hardware to cache recently translated entries in TLBs. *TLB reach* is the total memory size mapped by a TLB (number of entries times their page sizes). Large TLB reach tends to reduce the likelihood of misses. TLB reach can be expanded by increasing the number of TLB entries or by increasing page size.

However, since TLB lookup is on the critical path of each memory access, it is very challenging to increase the number of TLB entries without adding extra latency and energy overheads. Modern ISAs instead provide additional larger page sizes to increase TLB reach. For example, x86-64 supports 2MB pages and 1GB pages in addition to the default 4KB pages. Table 3-1 describes the TLB hierarchy in the 32 nm Intel Sandy Bridge processors used in this paper. The per-core TLB reach is a small fraction of the multi-TB physical memory available in current and future servers. The aggregate TLB reach of all cores is somewhat larger but still much less than a terabyte, and summing per-core TLB reaches only helps if memory is perfectly partitioned among cores.

To investigate the performance impact of TLB misses, I use hardware performance counters to measure the processor cycles spent by the hardware page-table-walker in servicing TLB misses. In my Intel's Sandy Bridge test machine the specific performance counter numbers used for measuring cycles spent on TLB load-misses and TLB store-misses are 0x08 (mask 0x04) and 0x49 (mask 0x04), respectively. In x86-64, a hardware page-table-walker locates the missing page-table entry on a TLB miss and loads it into the TLB by traversing a four-level page table. A single page-table walk here may cause up to four memory accesses. The estimate for TLB-miss latency is conservative, as I do not account for L1 TLB misses that hit in L2 TLB (for 4KB pages), which can take around 7 cycles [59]. I run the experiments with base (4KB), large (2MB) and huge page (1GB).

I report TLB miss latency as a fraction of total execution cycles to estimate its impact on the execution time. Table 3-4 lists my findings (the micro-benchmark GUPS is separated at the

Table 3-4. TLB miss cost.

	Percentage of execution cycles servicing TLB misses			
	Base Pages (4KB)		Large Pages (2MB)	Huge Pages (1GB)
	D-TLB	I-TLB	D-TLB	D-TLB
graph500	51.1	0	9.9	1.5
memcached	10.3	0.1	6.4	4.1
MySQL	6.0	2.5	4.9	4.3
NPB:BT	5.1	0.0	1.2	0.06
NPB:CG	30.2	0.0	1.4	7.1
GUPS	83.1	0.0	53.2	18.3

bottom). First, I observe that TLB misses on data accesses (D-TLB misses), account for significant percentage of execution cycles with 4KB pages (e.g., 51% of execution cycles for *graph500*). The TLB misses on instruction fetches (I-TLB misses), however, are mostly insignificant and thus are ignored in the rest of the study. With 2MB pages, the effect of D-TLB miss moderates across all workloads as expected: for *NPB:CG* cost of D-TLB misses drops from 30% to 1.45%. However, across most of the workloads (*graph500, memcached, MySQL*) D-TLB misses still incur a non-negligible cost of 4.9% - 9.9%.

Use of 1GB pages reveals more interesting behavior. While most of the workloads observe a reduction in the fraction of time spent servicing TLB misses, *NPB:CG* observes a significant increase compared to 2MB pages. This stems from almost 3X increase in TLB miss rate likely due to the smaller number of TLB entries available for 1GB pages (4 entries)

compared to 2MB pages (32 entries). A sparse memory access pattern can result in more misses with fewer TLB entries. This possibility has been observed by *VMware*, which warns users of possible performance degradation with large pages in their ESX server [90].

In summary, across most workloads (graph500, memcached, MySQL) I observe substantial overhead for servicing TLB misses on the 96 GB test machine (4.3% to 9.9%), even using large pages. Results will likely worsen for larger memory sizes. While the latency cost of TLB misses suffice to show the significant overhead of paging, there are several other costs that are beyond the scope of this analysis: the dynamic energy cost of L1 TLB hit [83], the energy cost of page table walk on TLB miss, and the memory and cache space for page tables.

> ***Observation 2:*** Big-memory workloads pay a cost of page-based virtual memory: substantial performance lost to TLB misses.

3.2.3 Application Execution Environment

Finally, I qualitatively summarize other properties of big-memory workloads that the rest of this paper exploits.

First, many big-memory workloads are long-running programs that provide 24x7 services (e.g., web search and databases). Such services receive little benefit from virtual memory optimizations whose primary goal is to allow quick program startup, such as demand paging.

Second, services such as in-memory caches and databases typically configure their resource use to match the resources available (e.g., physical memory size).

Third, many big-memory workloads provide a service where predictable, low latency operation is desired. Thus, they often run either *exclusively* or with a few non-interfering tasks to guarantee high performance [62], low latency and performance predictability. A recent study by Reiss et al. [78] finds that in Google's datacenters, a small fraction of long-running jobs use more than 90% of system resources. Consequently, machines running big-memory workloads often have one or a few *primary processes* that are the most important processes running on a machine and consume most of the memory.

Observation 3: Many big-memory workloads:

a) Are long running,

b) Are sized to match memory capacity,

c) Have one (or a few) primary process(es).

3.3 Efficient Virtual Memory Design

Inspired by the observations in Section 3.2.2, I propose a more efficient virtual memory mechanism that enables fast and minimalist address translation through segmentation *where possible*, while defaulting to conventional page-based virtual memory *where needed*. In effect, I develop a hardware-software co-design that exploits big-memory workload characteristics to significantly reduce the virtual memory cost for *most* of its memory usage that does not benefit from rich features of page-based virtual memory. Specifically, I propose *direct-segment* hardware (Section 3.3.1) that is used via a software *primary region* (Section 3.3.2).

3.3.1 Hardware Support: Direct Segment

The goal is to enable fast and efficient address translation for a part of process's address space that does not benefit from page-based virtual memory, while allowing conventional paging for the rest. To do this, the proposed scheme translates a contiguous virtual address range directly onto a contiguous physical address range through hardware support called *a direct segment—without* the possibility of a TLB miss. This contiguous virtual address range can be arbitrarily large (limited only by the physical memory size of the system) and is mapped using a small fixed-sized hardware. Any virtual address outside the aforementioned virtual address range is mapped through conventional paging. Thus, *any amount of physical memory* can be mapped completely through a direct segment, while allowing rich features of paging where needed (e.g., for copy-on-write, guard pages). It also ensures that the background (*non-primary*) processes are unaffected and full backward compatibility is maintained.

The proposed direct-segment hardware adds modest, fixed-sized hardware to each core (or to each hardware thread context with hardware multithreading). Figure 3-2, provides a logical view with the new hardware shaded. The figure is *not to scale***,** as the D-TLB hardware is much larger. For example, for Intel's Sandy Bridge, the L1 D-TLB (per-core) has 100 entries divided across three sub-TLBs and backed by a 512-entry L2 TLB.

As shown in Figure 3-2, direct segments add three registers per core as follows:

- BASE holds the start address of the contiguous virtual address range mapped through direct segment,

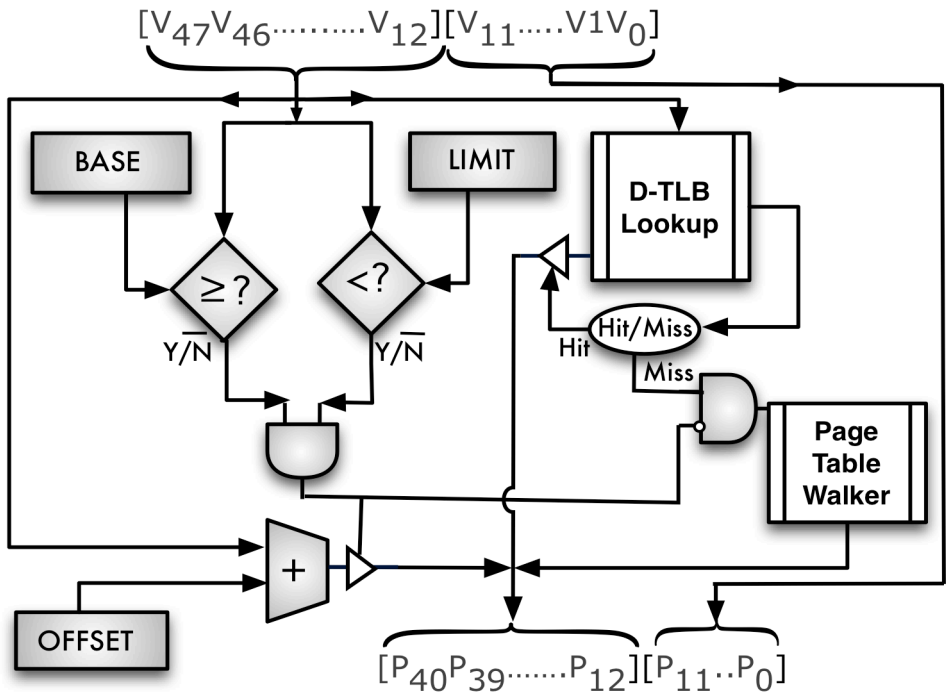


Figure 3-2. Logical view of address translation with direct segment. Added hardware is shaded. (Not to scale)

- LIMIT holds the end address of the virtual address range mapped through direct segment, and,
- OFFSET holds the start address of direct segment's backing contiguous physical memory minus the value in BASE.

Direct segments are aligned to the base page size, so page offset bits are omitted from these registers (e.g., 12 bits for 4KB pages).

Address translation: As depicted in Figure 3-2 on each data memory reference, data virtual address *V* is presented to both the new direct-segment hardware and the D-TLB. If virtual

address V falls within the contiguous virtual address range demarcated by the direct segment's base and limit register values (i.e., BASE $\leq V <$ LIMIT), the new hardware provides the translated physical address as V + OFFSET and suppresses the D-TLB translation process. Notably, addresses translated using direct segments never suffer from TLB misses. Direct-segment hardware permits read-write access only.

A given virtual address for a process is translated either through direct segment or through conventional page-based virtual memory but never both. Thus, both direct segment and D-TLB translation can proceed in parallel. A virtual address outside the direct segment may hit or miss in L1 TLB, L2 TLB, etc., and is translated conventionally. This simplifies the logic to decide when to trigger hardware page-table walk and only requires that the delay to compare the virtual address against BASE and LIMIT be less than the delay to complete the entire D-TLB lookup process (which involves looking up multiple set-associative structures).

The OS is responsible for loading proper register values, which are accessible only in privileged mode. Setting LIMIT equal to BASE disables the direct segment and causes all memory accesses for the current process to be translated with paging. I describe how the OS calculates and handles the value of these registers in the next section.

Unlike some prior segment-based translation mechanisms [24,97] direct segments are also notable for what they do *not* do. Direct segments:

(a) Do not export two-dimensional address space to applications, but retain a standard linear address space.

(b) Do not replace paging: addresses outside the segment use paging.

(c) Do not operate on top of paging: direct segments are not paged.

3.3.2 Software Support: Primary Region

System software has two basic responsibilities in the proposed design. First, the OS provides a *primary region* abstraction to let applications specify which portion of their memory does not benefit from paging. Second, the OS provisions physical memory for a primary region and maps all or part of the primary region through a direct segment by configuring the direct-segment registers.

3.3.2.1 Primary Regions

A primary region is a contiguous range of virtual addresses in a process's address space with uniform read-write access permission. Functionalities of conventional page-based virtual memory like fine-grain protection, sparse allocation, swapping, and demand paging are not guaranteed for memory allocated within the primary region. It provides only the functionality described in Section 3.2.1 as necessary for the majority of a big-memory workload's memory usage, such as MySQL's buffer cache or memcached's cache. Eschewing other features enables the primary region of a process to be mapped using a direct segment.

The software support for primary regions is simple: (i) provision a range of contiguous virtual addresses for primary region; and (ii) enable memory requests from an application to be mapped to its primary region.

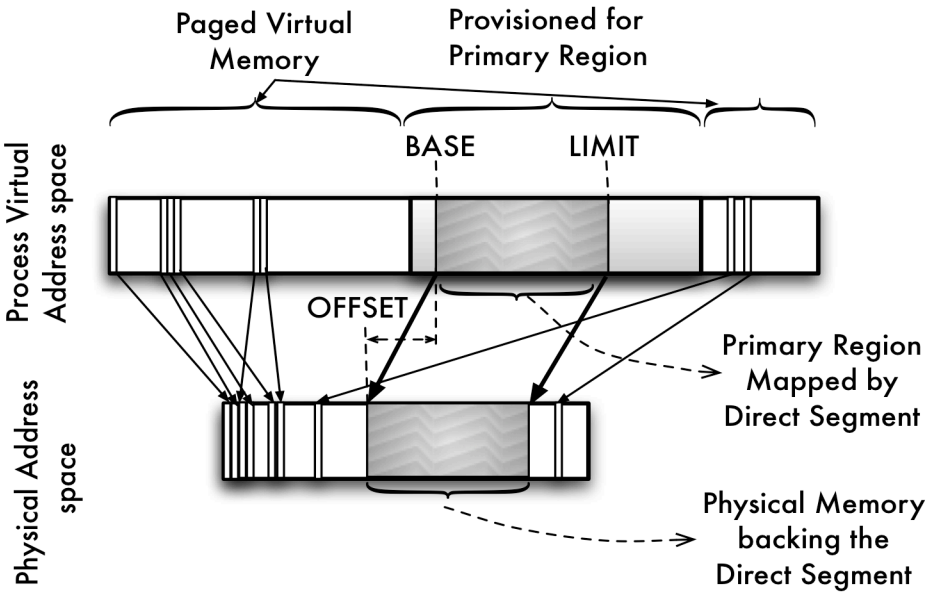


Figure 3-3. Virtual Address and Physical Address layout with a primary region. Narrow rectangles represent pages.

Provisioning virtual addresses: Primary regions require a contiguous virtual address range in a process's address space. During creation of a process the OS can reserve a contiguous address partition in the infant process to be used *exclusively* for memory allocations in the primary region. This guarantees contiguous space for the primary region. Conservatively, this partition must be big enough to encompass the largest possible primary region—i.e., the size of the physical memory (e.g., 4TB). Since 64-bit architectures have an abundance of virtual address space—128TB for a process in Linux on x86-64—it is cheap to reserve space for the primary region. Alternatively, the operating system can defer allocating virtual addresses until the application creates a primary region; in this case the request may fail if the virtual address space is heavily fragmented

In Figure 3-3 the top outermost rectangular box shows the virtual address space layout of a process with primary region. The lightly shaded inner box represents the address range provisioned for the primary region. The remainder of the address space can be mapped through conventional pages (narrow rectangles).

Memory allocations in primary region: The OS must decide which memory allocation requests use the primary region. As mentioned earlier, any memory allocation that does not benefit from paging is a candidate.

There are two broad approaches: opt in and opt out. First, a process may *explicitly* request that a memory allocation be put in its primary region via a flag to the OS virtual-memory allocator (e.g., *mmap()* in Linux), and all other requests use conventional paging. Second, a process may *default* to placing dynamically allocated anonymous (not file-backed) with uniform read-write permission in the primary region. Everything else (e.g., file-mapped regions, thread stacks) use paging. Anonymous memory allocations can include a flag to "opt out" of the primary region if paging features are needed, such as sparse mapping of virtual addresses to physical memory.

3.3.2.2 Managing Direct-Segment Hardware

The other major responsibility of the OS is to set up the direct-segment hardware for application use. This involves two tasks. First, the OS must make contiguous physical memory available for mapping primary regions. Second, it must set up and manage the direct-segment registers (BASE, LIMIT, and OFFSET).

Managing physical memory: The primary task for the OS is to make contiguous physical memory available for use by direct segments. As with primary regions, there are two broad approaches. First, the OS can create contiguous physical memory dynamically through periodic memory compaction, similar to Linux's Transparent Huge Pages support [23]. The cost of memory compaction can be amortized over the execution time of long-running processes. I measured that it takes around 3 seconds to create a 10GB range of free, contiguous physical memory. For this memory size, compaction incurs a 1% overhead for processes running 5 minutes or longer (and 0.1% overhead for one hour).

The second, simpler approach is to use *physical memory reservations* and set aside memory immediately after system startup. The challenge is to know how much memory to reserve for direct-segment mappings. Fortunately, big-memory workloads are already cognizant of their memory use. Databases and web caches often pre-configure their primary memory usage (e.g., cache or buffer pool size), and cluster jobs, like those at Google, often include a memory size in the job description [78].

Managing direct-segment registers: The OS is responsible for setting up and managing direct-segment registers to map part or all of the primary region of one or few critical primary processes on to contiguous physical memory. To accomplish this task the OS first needs to decide which processes in a system should use direct segment for address translation. To minimize administration costs, the OS can monitor processes to identify long-running processes with large anonymous memory usage, which are candidates for using direct segments. To provide more predictability, the OS can provide an explicit admission control mechanism where

system administrators identify processes that should map their primary region with a direct segment.

With explicit identification of processes using direct segments, the system administrator can specify the desired amount of memory to be mapped with a direct segment. The OS then finds a contiguous chunk of physical memory of the desired size from the reserved memory. If no such chunk is found then the largest available contiguous physical memory chunk determines the portion of the primary region mapped through the direct segment. However, this region can be dynamically extended later on as more contiguous physical memory becomes available, possibly through compaction or de-allocations.

Figure 3-3 provides a pictorial view of how the values of three direct-segment registers are determined; assuming all of primary region of the process is mapped through direct segment in the example. As shown in the figure, the OS uses the BASE and LIMIT register values to demarcate the part of the primary region of a process to be mapped using direct segment (dark-shaded box in the upper rectangle). OFFSET register is simply the difference between BASE and the start address of the physical memory chunk mapping the direct segment. The OS disables a process's direct segment by setting BASE and LIMIT to the same value, e.g., zero.

The values of the direct-segment registers are part of the context of a process and maintained within the process metadata (i.e., process control block or PCB). When the OS dispatches a thread, it loads the BASE, LIMIT, and OFFSET values from the PCB.

Growing and shrinking direct segment: A primary region can be mapped using a direct segment, conventional pages, or both. Thus, a process can start using primary regions with

paging only (i.e., BASE = LIMIT). Later, the OS can decide to map all or part of the primary region with a direct segment. This may happen when contiguous physical memory becomes available or if the OS identifies the process as "big-memory". The OS then sets up the direct-segment registers and deletes the page table entries (PTEs) for the region. This can also be used to grow the portion of primary region mapped through a direct segment. For example, initially the new space can be mapped with paging, and later converted to use an expanded direct segment if needed.

The OS may also decide to revert to paging when under memory pressure so it can swap out portions. The OS first creates the necessary PTEs for part or all of the memory mapped by the direct segment, and then updates the direct-segment registers. As with other virtual-memory mapping updates, the OS must send shootdown-like inter-process interrupts to other cores running the process to update their direct-segment registers.

3.4 Software Prototype Implementation

The direct-segment prototype is implemented by modifying Linux kernel 2.6.32 (x86-64). The code has two parts: implementation of the primary region abstraction, which is common to all processor architectures, and architecture-specific code for instantiating primary regions and modeling direct segments.

3.4.1 Architecture-Independent Implementation

The common implementation code provisions physical memory and assigns it to primary regions. The prototype implementation is simplified by assuming that only one process uses a

direct segment at any time (called the *primary process)*, but this is *not* a constraint of the design. Further, the prototype uses explicit identification of the primary process and physical memory reservations, although more automatic implementations are possible as detailed in Section 3.3.2. Below, I describe the main aspects of the implementation—identifying the process using a direct segment, managing virtual address and managing physical memory.

Identifying the primary process: A new system call was implemented to identify the executable name of the primary process. The kernel stores this name in a global variable, and checks it when loading the binary executable during process creation. If a new process is identified as primary process then OS sets an "*is_primary*" flag in the Linux task structure (process control block). The OS must be notified of the executable name before the primary process launches.

Managing virtual address space: When creating a primary process, the OS reserves a contiguous virtual address range for a primary region in the process's virtual address space that is of 4TB in size – much larger than the physical memory in the test system. This guarantees the availability of a contiguous virtual address range for memory allocations in the primary region.

The prototype uses an "opt in" policy and places all anonymous memory allocations with read-write permission contiguously in the address range reserved for the primary region. This way, all heap allocations and *mmap()* calls for anonymous memory are allocated on the primary region, unless explicitly requested otherwise by the application with a flag to *mmap()*.

Managing physical memory: The prototype reserves physical memory to back direct segments. Specifically, the new system call described above notifies the OS of the identity of the primary process also specifies the estimated size of the physical memory that is to be mapped through direct segment. Then a contiguous region of physical memory of the given size is reserved using Linux's *memory hotplug* utility [56], which takes a contiguous physical memory region out of the kernel's control (relocating data in the region if needed). The current prototype does not support dynamic resizing primary regions or direct segments.

3.4.2 Architecture-Dependent Implementation

The direct segment design described in Section 3.3.1 requires new hardware support. To evaluate primary region with direct segment capability on real hardware without building new hardware, I emulate its functionality using 4KB pages. Thus, I built an architecture-dependent implementation of direct segments.

The architecture-dependent portion of my implementation provides functions to create and destroy virtual-to-physical mappings of primary regions to direct segments, and functions to context switch between processes.

On a machine with real direct-segment hardware, establishing virtual-to-physical mapping between a primary region and a direct segment would require calculating and setting the direct-segment registers for primary processes as described in Section 3.3.2. The OS creates direct-segment mappings when launching a primary process. It stores the values of the direct-segment registers as part of process's metadata. Deleting a direct segment destroys this

information. On a context switch the OS is responsible for loading the direct-segment registers for the incoming process.

Without real direct-segment hardware, I emulate direct-segment functionalities using 4KB pages. More specifically, I modify Linux's page fault handler so that on a page fault within the primary region it calculates the corresponding physical address from the faulting virtual page number. For example, let us assume that $VA_{start\_primary}$ and $VA_{end\_primary}$ are the start and end virtual addresses of the address range in the primary region mapped through direct segment, respectively. Further, let PA_{start_chunk} be the physical address of the contiguous physical memory chunk for the direct segment mapping. The OS then sets the BASE register value to $VA_{start\_primary}$, LIMIT register value to $VA_{end\_primary} + 1$, and OFFSET register value to (PA_{start_chunk} - $VA_{start\_primary}$). If VA_{fault} is the 4KB page-aligned virtual address of a faulting page, then the modified page-fault handler first checks if BASE $\leq VA_{fault} <$ LIMIT. If so, the handler adds a mapping from VA_{fault} to VA_{fault} + OFFSET to the page table.

This implementation provides a *functionally complete* implementation of primary region and direct segments on real hardware, albeit without its performance. It captures all relevant hardware events for direct segment and enables performance estimation of direct segment for big-memory workloads without waiting for new hardware. Section 3.5.1 describes the details of my performance evaluation methodology.

3.5 Evaluation

In this section, I describe the evaluation methodology for quantifying the potential benefits of direct-segment. To address the challenges of evaluating long-running big-memory workloads, which would have taken months of simulation time, I devise an approach that uses kernel modification and hardware performance counters to estimate the number of TLB misses avoided by direct segments.

3.5.1 Methodology

Evaluating big-memory workloads for architectural studies is itself a challenging task. Full-system simulations would require very high memory capacity and weeks, if not months, of simulation time. Actually, a single simulation point using the gem5 simulator [14] would take several weeks to months and at least twice as much physical memory as the actual workload. It is particularly difficult for TLB studies, where TLB misses occur much less often than other micro-architectural events (e.g., branch mispredictions and cache misses). Downsizing the workloads not only requires intimate knowledge and careful tuning of the application and operating system, but also can change the virtual memory behavior that I want to measure.

I address this challenge using a combination of hardware performance counters and kernel modifications that together enable performance estimation. With this approach, I can run real workloads directly on real hardware. I first use hardware performance counters to measure the performance loss due to hardware page-table walks triggered by TLB misses. I then modify the kernel to capture and report the fraction of these TLB misses that fall in the primary region mapped using direct-segment hardware. Because these TLB misses would not have happened

with direct-segment hardware, this allows us to estimate the reduction in execution cycles spent on servicing TLB misses. I conservatively assume that TLB miss rate reduction directly correlates to the reduction in time spent on TLB misses, although the design can improve the TLB performance for addresses outside direct segments by freeing TLB and page-walk-cache resources. The following equation describes how I estimate the DTLB miss overhead when direct segment is employed (**Y%**).

$$\mathbf{Y\% = X*(1 - F)/(1 - F*X)}$$

In the above equation, **X** represents the DTLB miss cycles expressed as percentage of the execution cycles in the baseline (uses 4KB pages). **F** represents the fraction of DTLB misses that falls in direct-segment memory and thus can be eliminated if direct-segment hardware existed. The numerator of the above equation encapsulates the reduction in cycles spent on DTLB misses in proportion to the reduction in the number of DTLB misses. The denominator captures the estimated reduction in execution cycles as contribution due to DTLB misses reduces proportionally.

1. Baseline: I use hardware performance counters to estimate the fraction of execution cycles spent on TLB misses. I collect data with *oprofile* [84] by running each of the workloads for several minutes on the test platform described in Table 3-1. In my test machine, the hardware performance counter number 0x3c captures the execution cycles, while the counter numbers 0x08 (mask 0x04) and 0x49 (mask 0x04) capture the cycles spent on servicing the DTLB misses for load and store instructions, respectively.

2. Primary Region/Direct Segment: To estimate the efficacy of the proposed scheme, I determine what fraction of the TLB misses would fall in the direct segment. To achieve this, it needs to determine whether the miss address for each TLB miss falls in the direct segment. Direct segments eliminate these misses.

Unfortunately, the x86 architecture uses a hardware page table walker to find the PTEs on a TLB misses so an unmodified system cannot immediately learn the address of a TLB miss. I therefore tweak the Linux kernel to artificially turn each TLB miss into a *fake* page fault by making PTEs invalid after inserting them into the TLB. This mechanism follows from the methodology similar to Rosenblum's context-sensitive page mappings [79]. I modify the page fault handler to record whether the address of each TLB miss comes from the primary or conventional paged memory. See Box 1 for more details on this mechanism.

I then estimate the reduction in TLB-miss-handling time with direct-segment using a linear model. This is similar to a recent work on coalesced TLB by Pham et al. [73]. More specifically, I found the fraction of total TLB misses that fall in the primary region mapped with a direct segment using the methodology described above. I also measure the fraction of execution cycles spent by hardware page walker on TLB misses in the baseline system using performance counters. Finally, I estimate that the fraction of execution cycles spent on TLB misses with direct segment is linearly reduced by the fraction of TLB misses eliminated by direct segment over that of the baseline system.

Box 1. TLB miss tracking method

The prototype tracks TLB misses by making an x86-64 processor act as if it had a software-filled TLB by making TLB misses trap to the OS.

Forcing traps: In x86-64, page table entries (PTEs) have a set of reserved bits (41-51 in the test platform here) that cause a trap when loaded into the TLB. By setting a reserved bit, the prototype implementation ensures that any attempt to load a PTE will cause a trap.

TLB Incoherence: The x86-64 architecture does not *automatically* invalidate or update a TLB entry when the corresponding memory-resident PTE is modified. Thus, a TLB entry can continue to be used even after its corresponding PTE in the memory has been modified to set a reserved bit.

Trapping on TLB misses: The prototype uses these two features to intentionally make PTEs incoherent and generate a *fake* page fault on each TLB miss. All user-level PTEs for a primary process are initially marked *invalid*. The first access to a page triggers a page fault. In this handler, the PTE is made valid and then take two additional actions. First, it forces the TLB to load the correct PTE by touching the page with the faulting address. This puts the correct PTE into the TLB. Second, it *poisons* the PTE by setting a reserved bit. This makes the PTE in memory invalid and inconsistent with the copy in the TLB. When the processor tries to re-fetch the entry on a later TLB miss, it will encounter the poisoned PTE and raises an exception with a unique error code identifying that reserved bit was set.

When a fake page fault occurs (identified by the error code), I record whether the address falls in the primary region mapped using direct segment. The prototype then performs the two actions above to reload the PTE into the TLB and re-poison the PTE in memory.

My colleagues are re-implementing this mechanism as a tool named *BadgerTrap* in a newer version of Linux kernel.

This estimation makes the simplifying assumption that the average TLB miss latency remains same across different number of TLB misses. However, this is likely to *underestimate* the benefit of direct segments, as it does not incorporate the gains from removing L1 TLB misses that hit in L2 TLB. A recent study shows that L2 TLB hits can potentially have non-negligible performance impact [59]. Further, unlike page-based virtual memory the direct-segment region does not access page tables and thus, it does not incur data cache pollution due to them. The direct segment also frees up address translation resources (TLB and page-walk cache) for others to use. However, I omit these potential benefits in the evaluation.

3.5.2 Results

In this section I discuss two aspects of performance:

1. What is the performance gain from primary region/direct segments?
2. How does the primary region/direct-segment approach scale with increasing memory footprint?

Performance gain: Figure 3-4 depicts the percentage of total execution cycles spent by the workloads in servicing D-TLB misses (i.e., the TLB miss overhead). For each workload I evaluate four schemes. The first three bars in each cluster represent conventional page-based virtual memory, using 4KB, 2MB and 1GB pages, and are measured using hardware performance counters. The fourth bar (often invisible) is the estimated TLB overhead for primary region/direct segments. As observed earlier, I find that even with larger page sizes significant execution cycles can be spent on servicing TLB misses.

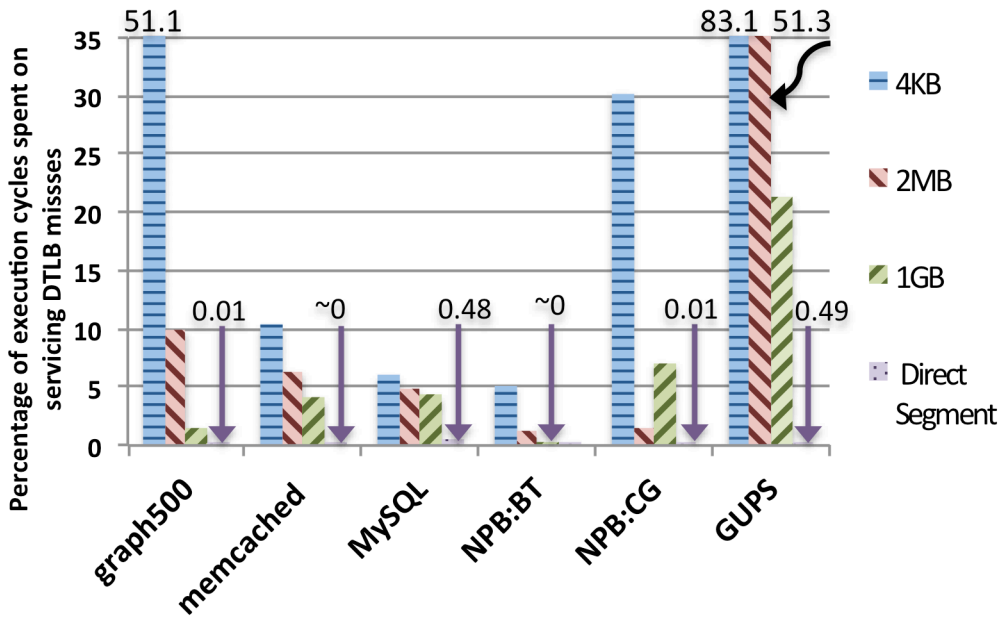


Figure 3-4. Percentage of execution cycles spent on DTLB misses. Values larger than 35% are shown above the bars, while very small values shown by straight arrows.

With primary regions and direct segments, the workloads waste practically *no* time on D-TLB misses. For example, in *graph500* the TLB overhead dropped to 0.01%. Across all workloads, the TLB overhead is below 0.5%. Such results are hardly surprising: from the Table 3-5, as one can observe that most of the TLB misses are captured by the primary region and thus avoided by the direct segment. This correlates well with Table 3, which shows that more than 99% of allocated memory belongs to anonymous regions that can be placed in a primary region and direct segment. The only exception is MySQL, where the direct segment captured only 92% of TLB misses. I found that MySQL creates 100s of threads and many TLB misses occur in the thread stacks and the process's BSS segment memory that holds compile-time constants and global data structures. Many TLB misses also occur in file-mapped regions of memory as well.

Table 3-5. Reduction in D-TLB misses.

	Percent of D-TLB misses in the direct segment
graph500	99.99
memcached	99.99
mySQL	92.40
NBP:BT	99.95
NBP:CG	99.98
GUPS	99.99

.

Scalability: Direct segments provide *scalable* virtual memory, with constant performance as memory footprint grows. To illustrate this benefit, Figure 5 compares the fraction of execution cycles spent on DTLB misses with different memory footprints for three x84-64 page sizes and direct segments. I evaluate GUPS, whose dataset size can be easily configured with a scale parameter. I however note that GUPS represents worst-case scenario of random memory access.

As the workload scales up, the TLB miss overhead grows to an increasing portion of execution time across all page sizes with varying degree (e.g., from 0% to 83% for 4KB pages, and from 0% to 18% for 1GB pages). More importantly, one can notice that there are distinct inflection points for different page sizes, before which TLB overhead is near zero and after which TLB overhead increases rapidly as the workload's working set size exceeds TLB reach. Use of larger pages can only push out, but not eliminate, these inflection points. The existence of

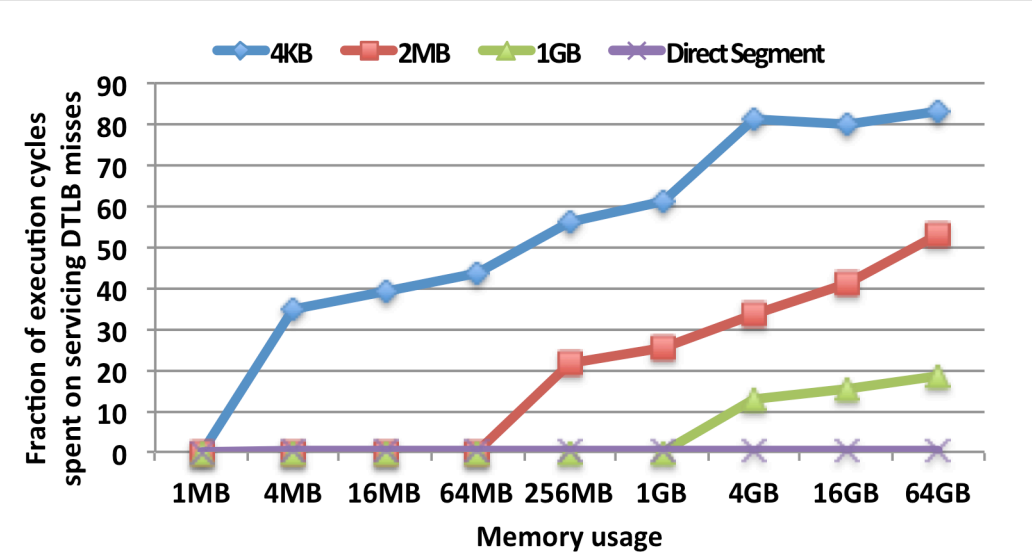


Figure 3-5. DTLB miss overhead when scaling up GUPS.

these inflection points and the upward overhead trends demonstrate the TLB's scalability bottleneck. In contrast, primary regions with direct segments provide a scalable solution where the overhead of address translation remains constant and negligible when memory footprints increase.

Selective raw numbers for this chapter appears in Appendix in page 163.

3.6 Discussion

In this section, I discuss possible concerns regarding primary region/direct segment.

Why not large pages? Modern hardware supports large pages to reduce TLB misses. Large pages optimize within the framework of page-based virtual memory and are hence constrained by its limitations, such as alignment restrictions. In contrast, my proposal is based on analysis of the memory needs of big-memory workloads, and meets those needs with minimal hardware

independent of the TLB. In the following paragraphs I describe major shortcoming of large pages that direct segments proposal can overcome.

First, large pages and their TLB support do not automatically scale to a much larger memories. To support big-memory workloads, the size of large pages and/or size of TLB hierarchy must continue to scale as memory capacity increases. This requires continual updates to the processor micro-architecture and/or operating system, and application's memory management functionality. Being a cache, TLBs are reliant on memory-access locality to be effective and it can be a mismatch for future big-memory workloads with poor locality (e.g., streaming and random access) [71,77]. In contrast, direct segments only need a one-time, much simpler change in processor, OS, and applications, with full backward compatibility. It can then map arbitrarily large amounts of memory, providing a *scalable* solution for current and future systems.

Second, efficient TLB support for multiple page sizes is difficult. Because the indexing address bits for large pages are unknown until the translation completes, a split-TLB design is typically required where separate sub-TLBs are used for different page sizes [88]. This design, as employed in recent Intel processors such as Westmere, Sandy Bridge, and Ivy Bridge, can suffer from performance unpredictability while using larger page sizes as observed in experiments described in Section 3.2.2. For the application *NPB:CG* the fraction of processor cycles spent on servicing TLB misses rises substantially when 1GB pages are used instead of 2MB pages. This demonstrates that performance with large pages can be micro-architecture dependent. An alternative design could use a unified, fully associative TLB, but this increases TLB power and

access latency while limiting its size. In contrast, direct segment obviate such TLB design complexities and is *micro-architecture agnostic*.

Third, large page sizes are often few and far apart. For example in x86-64, the large page sizes correspond to different levels in the hardware-defined multi-level radix-tree structure of the page table. For example, recent x86-64 processors have only three page sizes (4KB, 2MB, 1GB), each of which is *512 times* larger than the previous. This constant factor arises because 4KB pages that hold page tables contain 512 8-byte-wide PTEs at each node of the page table. Such page-size constraints make it difficult to introduce and flexibly use large pages. For example, mapping a 400GB physical memory using 1GB pages can still incur substantial number of TLB misses, while a 512GB page is too large. A direct segment overcomes this shortcoming, as its size can adapt to application or system needs.

Virtual machines with direct segment: Direct segments can be extended to reduce TLB miss overhead in virtualized environments as well. In a virtualized environment the memory accesses goes through two levels of address translations: (1) guest virtual address (*gVA*) to guest physical address (*gPA*) and (2) guest physical address (*gPA*) to system physical address (*sPA*). In x86-64 with hardware virtualization of the MMU, a TLB miss in a virtual machine is serviced by a 2-D page-table-walker that may incur up to 24 memory references [2]. Direct segments could be extended to substantially reduce this cost in the following ways.

The simplest extension of direct segment to virtualized environment would be to map the entire guest physical memory (*gPA*) to system physical memory (*sPA*) using a direct segment.

This extension can reduce a 2-D page-table walk to 1-D walk where each TLB miss incurs at most 4 memory accesses instead of 24.

Further, a direct segment can be used to translate addresses from gVA to gPA for primary processes inside the guest OS, similar to use in a native OS. This also reduces the 2-D page-table walk to one dimension.

Finally, direct segments can be used for *gVA* to *sPA* translations. This can be accomplished in two ways. First, similar to shadow paging [1], a hypervisor can populate the direct segment OFFSET register with the two-step translation of the direct-segment base from *gVA* to *sPA*. Any update by the guest OS to segment registers must trap into the hypervisor, which validates the base and limit, and calculates and installs the offset. Second, if a trap is deemed costly then nested BASE/LIMIT/OFFSET registers in hardware, similar to hardware support for nested paging, could be added without significant cost. However, evaluation of these techniques is beyond the scope of this dissertation. I also note that, similar to large pages use of direct segment may reduce opportunities for de-duplication[92].

Direct segments in presence of faulty memory: Modern operating systems like Solaris [89] may use fine-grain address remapping capability of page-based virtual memory to mask permanent faults in DRAM pages. More specifically, a given virtual page address that maps to a physical page frame with permanent faults is re-mapped to a new (non-faulty) physical page. The older faulty physical page is never used again. Unfortunately, if the faulty physical page frame is mapped using direct segment such a fine grain remapping is not possible. However, note that

transient faults in the DRAM are generally handled by error-correcting codes (ECC) and are unaffected by direct segment.

On a permanent fault in direct-segment memory it is possible to revert back to paging for part or all of memory mapped using direct segment. Direct segment mechanisms can do so by adjusting the values of BASE, LIMIT, OFFSET registers and creating necessary PTEs for paging. If the faulty page frame maps a virtual address near the start or the end of contiguous virtual address range of the direct segment then slightly shrinking the boundaries of that address range can allow faulty address being mapped using conventional paging, while most of non-faulty memory to continues to be mapped using direct segment. In the worst case, if the faulty page frame maps an address in the middle of direct segment's address range then the size of the memory mapped by direct segment may be halved. A naïve alternative solution could be to revert back to paging for the entire direct segment. Although correct, this solution is likely to be overkill and can lead to severe performance unpredictability.

Instead, another solution would be to relegate the responsibility of remapping faulty page frames in DRAM to the on-chip memory controller. The memory controller could be augmented to hold a list of faulty physical addresses that would be remapped to a non-faulty "real" physical address at the memory controller level instead being remapped by the OS. To enable such a design I propose to add a small hardware lookup table at the memory controller that would contain only the faulty physical address and its corresponding re-mapped (non-faulty) real physical address. I call this table Faulty Physical Address Table (FPAT). A memory request that misses in all on-chip caches and reaches memory controller needs to lookup the FPAT first. On a

match in the FPAT, the memory request would use the remapped real physical address instead of the original address to lookup the data in the DRAM. On a miss however, the memory request proceeds in conventional way.

The number of faulty physical page frames that could be remapped in memory controller is constrained by the number of entries in FPAT. Thus, a small and efficient FPAT could be effective only if a small number of physical page frame suffers faults. Indeed, a recent large-scale study on DRAM errors in IBM's Blue Gene supercomputer and Google datacenter demonstrates that a significant majority of all DRAM errors are accounted by a handful of faulty pages [37]. The study shows that on average around 4-18 pages in a multi-GB DIMM often covers more than 90% of the DRAM errors. This suggests that a FPAT design with 32-64 entries is likely to be sufficient under most scenarios. Further, since FPAT is accessed only on an off-chip memory access any performance impact due to access latency of FPAT is highly unlikely. Finally, if FPAT is full then DIMMs containing faulty physical address needs to be replaced. Replacing faulty DIMMs in servers are not uncommon.

Compared to the OS-mediated remapping of faulty physical pages the proposed handling at the memory controller has its advantages as well. If OS is to remap faulty pages then a feedback path from the DRAM to OS is necessary to identify the faulty page frames. In contrast, if the memory controller handles the remapping functionality locally then the OS can be kept agnostic of the fault behavior of the DRAM. Further, OS is constrained to remap pages at page size granularity only. However, a memory controller can remap faults at finer granularity, potentially avoiding wasting physical memory.

Direct segments for kernel memory: So far, I have focused on TLB misses to user-mode application memory. However, workloads like *memcached* that exercise the kernel network stack can waste up to additional 3.2% of execution cycles servicing TLB misses for kernel memory.

Direct segments may be adapted to kernel memory by exploiting existing regularity in kernel address space. For example, Linux's kernel memory usage is almost entirely *direct-mapped*, wherein the physical address is found by subtracting a static offset from the virtual address. This memory matches direct segments' capabilities, since they enable calculating physical address from a virtual address in similar fashion. If direct segments are not used in user mode, they can be used by the kernel for this memory (using paging for processes that do use a direct segment). Alternatively, additional segment registers can be added to each hardware thread context for a second kernel-mode direct segment.

The Linux kernel maps some memory using variable virtual addresses, which cannot use a direct segment. However, I empirically measured that often nearly 99% of kernel TLB misses reference direct-mapped addresses and thus can be eliminated by a direct segment.

3.7 Limitations

Not general (enough): Direct segments are not a fully general solution to TLB performance. I follow Occam's razor to develop the simplest solution that works for many important big-memory workloads, and thus propose a *single* direct segment. Future workloads may or may not justify more complex support (e.g., for kernels or virtual machines) or support for more segments.

Dynamic execution environment: While direct segments can simultaneously achieve address translation efficiency and compatibility, it should not be misused in environments with mismatching characteristics. In general, the proposed technique is less suitable for dynamic execution environments where many processes with unpredictable memory usage execute for short periods. However, I believe that it is straightforward to identify, often without human intervention, whether a given workload and execution environment is a good fit (or not) for direct segments and avoid misuse.

Sparse address space: In addition, software that depends on sparse virtual memory allocations may waste physical memory if mapped with direct segments. For example, *malloc()* in glibc-2.11 may allocate separate large virtual-memory heap regions for each thread (called an arena), but expects to use a small fraction of this region. If these per-thread heaps are mapped using a direct segment then the allocator could waste physical memory. My experiments use Google's *tcmalloc()* [33], which does not suffer from this idiosyncrasy.

3.8 Related Work

Virtual memory has long been an active research area. Past and recent work has demonstrated the importance of TLBs to the overall system performance [5,9,13,11,19]. I expect big-memory workloads and multi-TB memories to make this problem even more important**.**

Efficient TLB mechanisms: Prior efforts improved TLB performance either by increasing the TLB hit rate or reducing/hiding the miss latency. For example, recent proposals increase the effective TLB size through co-operative caching of TLB entries [86] or a larger second-level TLB shared by multiple cores [11]. Prefetching was also proposed to hide the TLB

miss latency [6,17,22]. SpecTLB [7] speculatively uses large-page translations while checking for overlapping base-page translations. Zhang et al. proposed an intermediate address space between the virtual and physical addresses, under which physical address translation is only required on a cache miss [103]. Recently, Pham et al. [73] proposed hardware support to exploit naturally occurring contiguity in virtual to physical address mapping to coalesce multiple virtual-to-physical page translations into single TLB entries.

Since servicing a TLB miss can incur a high latency cost, several processor designs have incorporated software or hardware PTE caches. For example, UltraSPARC has a software-defined Translation Storage Buffer (TSB) that serves TLB misses faster than walking the page table [66]. Modern x86-64 architectures also use hardware translation caches to reduce memory accesses for page-table walks [6].

There are also proposals that completely eliminate TLBs with a virtual cache hierarchy [45,101], where all cache misses consult a page table. However, these techniques work only for uniprocessors or constrained memory layout (e.g., to avoid address synonyms).

While these techniques make TLBs work better or remove them completely by going straight to the page table, they still suffer when mapping the large capacity of big-memory workloads. In contrast, I propose a small hardware and software change to eliminate most TLB misses for these workloads, independent of memory size and available hardware resources (e.g., TLB entries).

Support for large pages: Almost all processor architectures including MIPS, Alpha, UltraSPARC, PowerPC, and x86 support large page sizes. To support multiple page sizes these

architectures implement either a fully associative TLB (Alpha, Itanium) or a set-associative split-TLB (x86-64). Talluri et al. discusses the tradeoffs and difficulties of supporting multiple page sizes in hardware [88]. However, system software has been slow to support the full range of page sizes: operating system support for multiple pages sizes can be complicated [32,87] and generally follows two patterns. First, applications can explicitly request large pages either through use of libraries like libHugeTLBFS [35] or through special *mmap* calls (in Linux). Second, the OS can automatically use large pages when beneficial [23,70,87].

Although useful, I believe large pages are a non-scalable solution for very large memories as discussed in detail in Section 3.6. Unlike large pages, primary regions and direct segments do not need to scale the hardware resources (e.g., adding more TLB entries or new page size) or change the OS, which are required to support new page sizes as memory capacity scales.

TLB miss reduction in virtualized environment: Under virtual machine operation, TLB misses can be even more costly because addresses must be translated twice [10]. Researchers have proposed solutions specific to the virtualized environment. For example, hardware support for nested page tables avoids the software cost of maintaining shadow page tables [10], and recent work showed that the VMM page table could be flat rather than hierarchical [2]. As mentioned in Section 3.6, I expect direct segments can be made to support virtual machines to further reduce TLB miss costs.

Support for segmentation: Several past and present architectures supported a segmented address space. Generally, segments are either supported *without* paging, as in early

Intel 8086 processors [97], or more commonly on top of paging as in MULTICS [24], PowerPC, and IA-32 [44]. Use of pure segmentation is incompatible with current software, while segmentation on top of paging does not reduce the address translation cost of page-based virtual memory. In contrast, direct segments use both segments and paging, but never for the same addresses, and retains the same abstraction of a linear address space as page-based virtual memory.

4

Reducing Address Translation Energy

4.1 Introduction

A key design constraint of modern era computing is energy dissipation. In this chapter, I present Opportunistic Virtual Caching (OVC), which reduces the energy dissipation due to virtual memory's address translation. In particular OVC saves substantial TLB and L1 cache lookup energy by reducing the frequency of address translation and by enabling lower-associative cache lookup.

Almost all commercial processors today cache data and instructions using physical addresses and consult a TLB on every load, store, and instruction fetch. Thus, a TLB access must be performed for each cache access. However, processor designs are increasingly constrained by energy, and such a physically addressed caches lead to energy dissipation inefficiencies. TLB lookup must be fast and rarely miss—often necessitating an energy-hungry highly associative structure. Industrial sources report that 3-13% of core power (including caches) can be due to the

TLB [83], and an early study finds that TLB power can be as high as 15-17% of chip power [46,48]. My analysis shows that a TLB lookup can consume 20-38% of the energy of an L1 cache lookup.

Energy consumption is further exacerbated by efforts to reduce the critical-path latency of a cache access. Most current processors overlap the TLB lookup with indexing the L1 cache and use the TLB output during tag comparison [64,72]. Such a virtually indexed, physically tagged cache requires that the virtual index bits equal the physical index bits, which is only true if the index comes from the page offset. Thus, the L1 cache size divided by its associativity must be less than or equal to the page size. To satisfy this constraint, some L1 cache designs use a larger associativity than needed for good miss rates (e.g., 32KB L1 ÷ 4KB page size = 8-way), which leads to higher energy consumption compared to lower associative caches (Section 4.2.1).

Now that energy is a key constraint, it is worth revisiting *virtual caching* [15,101]. While most past virtual cache research focused on its latency benefit [15,34,93], OVC focuses on its potential energy benefits. A virtual L1 cache is accessed with virtual addresses and thus requires address translation only on cache misses. This design makes TLB accesses much less frequent and thus could reduce TLB-lookup energy substantially. Further, it can lower the L1 cache lookup energy by removing the associativity constraint on the L1 cache design described above.

However, virtual caches present several challenges that have hindered their adoption. First, a physical address may map to multiple virtual addresses (called synonyms). An update to one synonym must be reflected in all others, which could be cached in different places. Thus it requires additional hardware or software support to guarantee correctness. Second, virtual caches

store page permissions with each cache block, so that these can be checked on cache hits without a TLB access. When page permissions change, associated cache blocks must be updated or invalidated beyond the normal TLB invalidation. Third, virtual caches require extra mechanisms to disambiguate homonyms (a single virtual address mapped to different physical pages). Fourth, they pose challenges in maintaining coherence, as coherence is traditionally enforced using physical addresses. Finally, virtual caches can be incompatible with commercially important architectures. For example, the x86 page-table-walker uses physical addresses to find page-table entries [38], which creates problem for caching entries by virtual address.

I find, though, that many of these problems occur rarely in practice. I analyze the behavior of a set of applications running on real hardware with the Linux operating system to understand how synonyms are actually used and to measure the frequency and characteristics of page permission changes. As detailed in Section 4.3, I find that synonyms *are* present in most processes, but account for only up to 9% of static pages and up to 13% of dynamic references. Furthermore, 95-100% of synonym pages are read-only, for which update inconsistencies are not possible. I also find that page permission changes are relatively rare and most often involve all pages of a process, which allows permission coherence to be maintained through cache flushes at low overhead. Thus, a virtual cache, even without synonym support, could perform well, save energy, and almost always work correctly.

Since correctness must be absolute, I instead propose a best-of-both-worlds approach with opportunistic *virtual caching* (OVC) that exposes virtual caching as a *dynamic optimization* rather than a hardware design point. OVC hardware can cache a block with either a virtual or

physical address. Rather than provide complex support for synonyms in hardware [34,93] or enforce limits on which virtual addresses can have synonyms [60], OVC requires that the OS (with optional hints from applications) declare which addresses are not subject to read-write synonyms and can use virtual caching; all others use physical addresses and a normal TLB. This flexibility provides 100% compatibility with existing software by defaulting to physical caching. The OS can then save energy by enabling virtual caching when it is safe (*e.g.*, no read-write synonyms) and efficient (*e.g.*, few permission changes). OVC also provides a graceful software adoption strategy, where OVC can initially be disabled, then used only in simple cases (*e.g.*, read-only and private pages) and later extended to more complex uses (*e.g.*, OS page caches).

With simple modifications to Linux (roughly 240 lines of code), my evaluation shows that OVC can eliminate 94-99% of TLB lookup energy and saves more than 23% of L1 cache dynamic energy compared to a virtually indexed, physically tagged cache.

This chapter of the dissertation makes three contributions. First, I analyze modern workloads on real hardware to understand virtual memory behavior. Second, based on this analysis, I develop policies and mechanisms that use physical caching for backward compatibility, but virtual caching to save energy by avoiding many address translations. Third, I develop necessary low-level mechanisms for realizing OVC.

4.2 Motivation: Physical Caching Vs. Virtual Caching

In this section I first compare and contrast the benefits and limitations of existing physically and virtually addressed L1 caches to understand where virtual caching may be desirable and where physical caching may be needed.

4.2.1 Physically Addressed Caches

A physical L1 cache requires the address translation to finish before a cache lookup can be completed. In one possible design, the address translation completes before L1 cache lookup starts, which places the entire TLB lookup latency in the critical path. However, a more common design is to overlap the TLB lookup with the cache access [64,72]. The processor sends the virtual page number to the TLB for translation while sending the page offset to the cache for indexing into the correct set. Then the output of the TLB is used to find a matching way in the set. Such a design is termed a virtually-indexed/physically-tagged cache. In both designs, all cache accesses, both instruction and data, require a TLB lookup. When latency (and thus performance) is the single most important design objective, a virtually indexed/physically tagged design is attractive as it hides the TLB lookup latency from the critical path of cache lookups while avoiding the complexities of implementing a virtual cache.

In the remainder of this chapter, I focus on the second and more commonly used design. Henceforth the term *physical cache* is used to refer to a virtually indexed and physically tagged cache.

Table 4-1. L1 cache (32KB, 64B block) miss ratios with varying associativity.

L1 Data Misses per 1K Cache references			
	4-way	**8-way**	**16-way**
Parsec	34.400	33.885	33.894
Commercial	41.634	40.721	39.636
L1 Instr. Misses per 1K Cache references			
	4-way	**8-way**	**16-way**
Parsec	1.053	0.991	0.934
Commercial	12.202	12.011	11.938

When energy-dissipation is a first-class design constraint, two aspects of this physical cache design lead to higher energy consumption. First, TLB lookups are energy-hungry: they occur frequently – on every memory reference. TLBs can also cause thermal hotspots due to high power density [75]. As increasing working sets put further pressure on TLB reach [7], processors may require yet larger TLBs, thus making TLB energy consumption worse. Second, to allow indexing with virtual addresses, the address bits used for cache indexing must be part of the page offset. This requires that cache size ÷ associativity ≤ page size. For example, a 32KB L1 cache requires at least an 8-way set-associative design for 4KB pages. Accesses to larger-associative structures dissipate more energy.

With the method explained in Section 4.5.2, I empirically find that such a highly associative L1 cache can lead to energy-inefficient cache designs that provide little hit-rate improvement benefit from their increased associativity. Table 4-1 shows the number of misses per 1K cache references (MPKR) for Parsec and commercial workloads for a 32 KB L1 cache with varying associativity. For example, when associativity increases from 4 to 8 the MPKR

changes very little (e.g., < 1 MPKR). Even ignoring extra latency of higher-associativity lookups, this can at best lead to a 0.12-0.14% speedup when associativity is increased from 4 to 8 and 16 respectively (Table 4-3). However, Table 4-2 shows that a cache lookup consumes 30% more energy when the associativity is increased from 4 to 8, and 86% more energy when increased from 4 to 16. While extra misses can burn more energy due to access to lower-level caches, the high L1 hit rates makes this a non-issue. This data shows that designing higher-associativity caches to overlap TLB latency can lead to energy-inefficient L1 caches.

Table 4-2. Normalized energy per access to L1 cache (32KB) with varying associativity (w.r.t. 4-way).

	4-way	8-way	16-way
Read Dynamic Energy	1	1.309	1.858
Write Dynamic Energy	1	1.111	1.296

Table 4-3. Normalized runtime with varying L1 cache associativity (w.r.t. 4-way).

	4-way	8-way	16-way
Parsec	1	0.998	0.998
Commercial	1	0.998	0.996

4.2.2 Virtually Addressed Caches

A virtual L1 cache is both indexed and tagged by virtual address, and consequently does not require address translation to complete a cache hit. Instead, virtual caches usually consult a TLB on a miss to pass the physical addresses to the next level in the cache hierarchy. The primary advantage of this design is that TLB lookups are required only on misses. L1 cache hit rates are generally high and thus a virtual L1 cache acts as an effective energy filter on the TLB. Moreover, a virtual L1 cache removes the size and associativity constraints on the L1, which enables more energy-efficient L1 designs.

However, several decades of research on virtual caches have showed that they are hard:

Synonyms: Virtual-memory synonyms arise when multiple, different virtual addresses map to the same physical address. These synonyms can reside in multiple places (sets) in the cache under different virtual addresses. If one synonym of a block is modified, access to other synonyms with different virtual addresses may return stale data.

Homonyms: Homonyms occur when a virtual address refers to multiple physical locations in different address spaces. If not disambiguated, incorrect data may be returned.

Page mapping and protection changes: The page permissions must be stored with each cache block to check permissions on cache hits. However, when permissions change, these bits must be updated. This is harder than with a TLB because many blocks may be cached from a single page, each of which must be updated. In addition, when the OS removes or changes a page mapping, the virtual address for a cache block must change.

Cache block eviction: Evicting a block cached with a virtual address requires translating the address to a physical address to perform write-back to physical caches further down the hierarchy.

Maintaining cache coherence: Cache coherence is generally performed with physical addresses. With a virtual cache, the address carried by the coherence messages cannot be directly used to access the cache. Thus a reverse translation (physical-to-virtual) is logically required to find the desired cache block.

Backward compatibility: Virtual caches can break compatibility with existing processor architectures and operating systems. For example, the x86's hardware page-table-walker uses only physical addresses to find PTEs in caches or memory [38]. With a virtual cache, it is unclear how to make x86's page table walker work both correctly (a virtual L1 cache entry is like a synonym) and efficiently (if caching of PTEs is disabled). Moreover, virtual caches often break compatibility by requiring explicit OS actions (*e.g.*, cache flushes on permission changes) to maintain correctness.

These challenges hinder the adoption of virtual L1 caches despite their potential energy savings. OVC seeks an ideal situation that provides most of the benefits of virtual caches by using it as *a dynamic optimization* while avoiding their complexities to an extent possible and maintaining compatibility.

4.3 Analysis: Opportunity for Virtual Caching

I set out to determine how often the expensive or complex virtual cache events actually happen in the real world by studying several modern workloads running on real x86 hardware under Linux. First, I measure the occurrences of virtual-memory synonyms to determine how often and where they occur in practice. As noted in Section 2.2, synonyms pose a correctness problem for virtual caches. Second, I measure the frequency of page protection/mapping changes, as these events may be more expensive with virtual caches.

I measured applications drawn from Parsec benchmark suite [74], as well as some important commercial applications (workloads explained in Section 4.5.2) listed in Table 4-4 running on Linux. The synonym pages are identified by analyzing the kernel's physical page frame descriptors, and dynamic references to synonyms are measured using PIN [58].

4.3.1 Synonym Usage

Table 4-4 presents a characterization of synonyms for the workloads. A page with a synonym is a virtual page whose corresponding physical page is mapped by at least one other user-space virtual address. I make three observations from this data. First, all but one application had synonym pages, but very few pages (0.06-9%) had synonyms. Second, the dynamic access rate of synonym pages was low (0-26%), indicating that virtual caching could be effective for most references. Finally, most synonym pages are mapped read-only, and therefore cannot introduce inconsistencies. This occurs because these pages were often from immutable shared library code (95-100% of the synonym pages).

Table 4-4. Virtual memory synonym analysis.

Applications	Percentage of application - allocated pages that contains synonyms	Percentage of synonym containing pages that are *read-only*	Percentage of all dynamic user memory accesses to pages with synonyms
canneal	0.06%	100%	0%
fluidanimate	0.28%	100%	0%
facesim	0.00%	100%	0%
streamcluster	0.23%	100%	0.01%
swaptions	5.90%	100%	26%
x264	1.40%	100%	1%
bind	0.01%	100%	0.16%
firefox	9%	95%	13%
memcached	0.01%	100%	0%
specjbb	1%	98%	2%

I also found that the OS kernel could sometimes use synonyms in the kernel virtual address space to access user memory. For example, kernel-space synonyms are used during a copy-on-write page fault to copy content of the old page to the newly allocated page. Similarly kernel space synonyms are used to zero-fill in a newly allocated page. The kernel-space synonyms are temporary but can introduce inconsistency through read-write synonyms. Further to process a direct I/O request that bypasses the operating system's page cache (used by databases), kernel-space aliases may be used to zero-fill the pages belonging to the user-space buffer.

Observation 1: While synonyms are present in most applications, conflicting use of them is rare. This suggests that virtual caches can be used safely for most, but not all, memory references.

4.3.2 Page Mapping and Protection Changes

The operating system maintains coherence between the page-table permissions and the TLB by invalidating entries on mapping changes or protection downgrades, or by flushing the entire TLB. Table 4-5 presents the average inter-arrival time of TLB invalidations for the workloads (and its reciprocal – the TLB invalidation request per sec). The inter-arrival time of TLB invalidations varies widely across the workloads, but I make two broad observations. First, even the smallest inter-arrival time between invalidations (2.325ms for memcached) is orders of magnitude longer than a typical time to flush and refill a L1 cache (~ 5µs). Hence, flushing the cache is unlikely to have much performance impact. Second, I observe that almost all TLB invalidations (97.5-100%) flush the entire TLB rather than a single entry. Most TLB invalidations occur on context switches that invalidate an entire address space, and only a few are for page protection/permission changes. Consequently, complex support to invalidate cache entries from a single page may not be needed.

Observation 2: TLB invalidations that occur due to page mapping or protection changes are infrequent and are thus unlikely to create much overhead for virtual caching.

Table 4-5. Frequency of TLB invalidations.

	Mean time between TLB invalidation in ms (avg. TLB invalidations per sec per core)	Fraction of TLB invalidations to a single page
canneal	132.62 (7.5)	0%
facesim	75.64 (13.2)	0%
fluidanimate	52.63 (19.2)	0%
streamcluster	55.53 (18)	0%
swaptions	51.81 (19.3)	0%
x264	111.11 (9.4)	0%
bind	7.571 (132.1)	0.00%
firefox	4.761 (210.3)	0.10%
memcached	2.325 (430.1)	0%
specjbb	39.011 (25.6)	2.50%

4.4 Opportunistic Virtual Caching: Design and Implementation

The empirical analyses in the previous section suggest that while virtual cache challenges are real, they occur rarely in practice. All the applications studied provide *ample dynamic opportunities* for safe (*i.e.*, no read-write synonyms) and efficient (*i.e.*, no page permission/protection changes) use of virtual caches. Unfortunately, correctness and backward compatibility must be absolute and *not* "almost always".

To benefit from virtual caches and while sidestepping their dynamically-rare issues, I propose *opportunistic virtual caching* (OVC). OVC hardware can cache a block with *either* virtual or physical address (Section 4.4.1). Virtual caching saves energy (no TLB lookup on L1 hits and reduced L1 associativity). Physical caching provides compatibility for read-write

synonyms and caching page-table entries (and other structures) accessed by the processor with physical addresses.

To reap the benefits of OVC, the operating system must enable virtual caching for memory regions that are amenable to the use of virtual caching (Section 4.4.2). Importantly, I find that the OS kernel (Linux in this study) already possesses most of the information needed to determine which memory regions are suitable for virtual caching and which are not. While OVC defaults to physical-only caching to enable deployment of unmodified OSes and applications, changes to support virtual caching affected only around 240 lines of code in the Linux kernel (version 2.6.28-4).

4.4.1 OVC Hardware

OVC requires that hardware provide the following services to realize the benefits of caching with virtual addresses – (1) determining when to use virtual caching and when physical caching, (2) reducing power when possible by bypassing the TLB and reducing cache associativity (3) handling-virtual memory homonyms and page permission/protection changes, and (4) handling coherence requests for virtually cached blocks.

Determining when to use virtual caching: The hardware defines a one-bit register named ***ovc_enable*** that an operating system can set to enable OVC (default is unset). When OVC is enabled, the hardware needs to determine which memory accesses should use virtual caching and which should use physical caching. While there could be many possible mechanisms to make this differentiation, in the proposed solution the large virtual address space of modern 64-bit OSes is logically partitioned into two non-overlapping address ranges (partition $\boldsymbol{P_{physical}}$

and $\boldsymbol{P_{virtual}}$). The highest order bit of the user-space virtual address range (e.g., VA_{47}, the 48th bit in Linux for x86-64) determines the partition in which a virtual address of a cache lookup belongs to (in $\boldsymbol{P_{physical}}$ if VA_{47} is unset and $\boldsymbol{P_{virtual}}$ otherwise). Only cache lookups with virtual address in the partition $\boldsymbol{P_{virtual}}$ can use the virtual address to cache data. Thus, there is no added lookup cost to determine how an address is cached. A L1 miss for an access with address in $\boldsymbol{P_{virtual}}$ does address translation through a conventional TLB mechanism.

Opportunistically reducing lookup energy: When data can be cached using virtual address OVC takes advantage of it in two ways. First, OVC avoids TLB lookups on L1 cache hits. Second, OVC allows lower associativity L1 cache lookups. As shown in Figure 4-1, when cache lookup address falls in partition $\boldsymbol{P_{virtual}}$ (i.e., ***ovc_enable*** and VA_{47} are set), the TLB lookup is disabled and part of the virtual address is used for cache tag match. Otherwise, conventional physical cache lookup is performed where the TLB is performed in parallel with indexing into the L1 cache. On virtual L1 cache miss a TLB lookup is required before sending the request to the next cache.

Second, OVC dynamically lowers the associativity of L1 cache lookups. Note that the cache-associativity constraint of a physical cache, described in Section 4.2.1, need not hold true for virtually cached blocks. Figure 4-2 shows an example of how a banked L1 cache organization can be leveraged to allow lower-associativity cache lookup for a 32KB, 8-way set associative cache. The 8-way set-associative cache is organized in two banks each holding 4-ways of each set. For virtual addresses (*i.e.*, ***ovc_enable*** and VA_{47} are set), the processor only accesses one of the two banks (*i.e.*, 4 ways) based on the value of a single virtual-address bit from the tag (VA_{12}

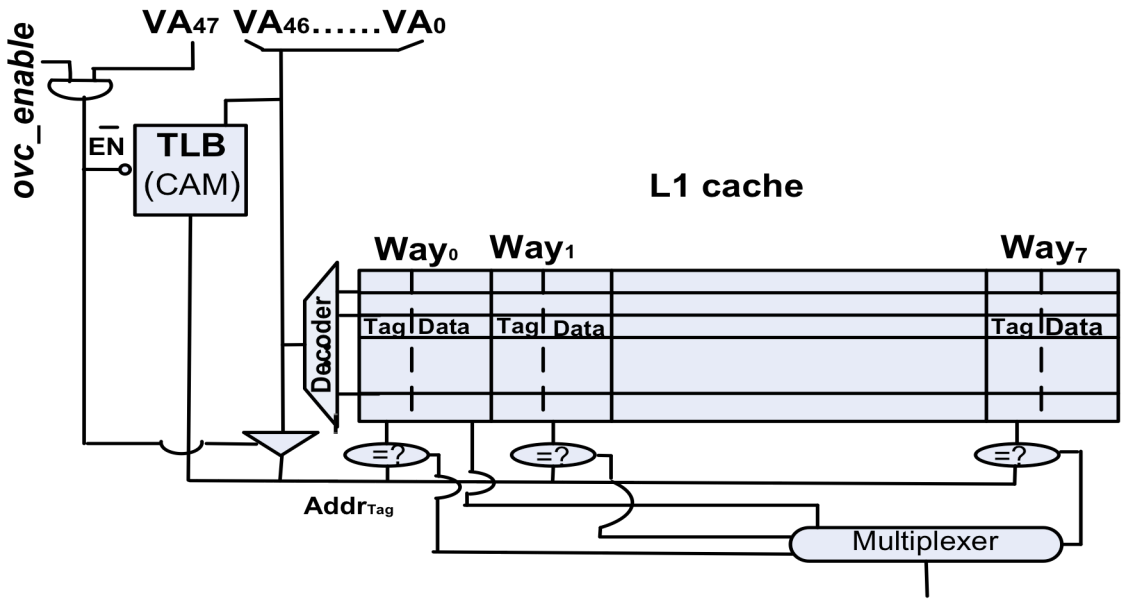


Figure 4-1. OVC L1 cache and TLB organization allow opportunistically bypassing TLB lookups.

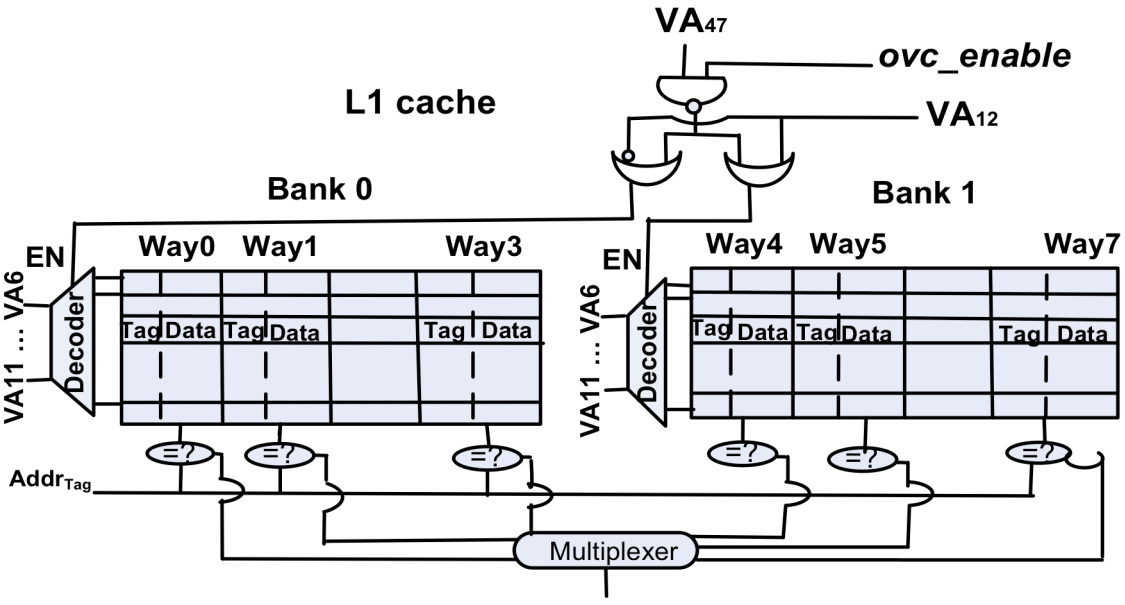


Figure 4-2. Opportunistic smaller associative L1 cache lookup using banked organization.

in the example). For other accesses using physical addressing, the processor performs a full 8-way lookup as in a conventional cache.

Handling homonyms and page permission changes: OVC implementation uses conventional address-space identifiers (ASIDs) to distinguish between different mappings of the same virtual address and avoids cache flushes on context switches. Both the ASID and the tag need to match for a cache hit to occur. OVC uses an all-zero ASID for blocks cached under the physical address (which results in an ASID match for any physical cache access). To handle the kernel address space, which is shared by all processes, OVC copies the *global* bit of the x86 PTE (which is set for globally shared kernel memory) to each cache block. Privileged mode access for blocks with this bit set do not need an ASID match. ASID overflow can be handled by modifying Linux's existing ASID management code to trigger a cache flush before reusing an ASID.

Page permissions (*e.g.*, read, write, execute, privileged) augment the coherence state permissions for each cache block and are checked along with coherence permissions. A page permission miss-match (*e.g.*, write request for a block with read permission) triggers a cache miss, which results in access to the TLB. It is then handled appropriately as in conventional physical cache for page permission miss-matches. Page mapping or permission downgrades trigger a cache flush.

Cache block eviction: Eviction of a dirty L1 block invokes a write-back to a physical L2 cache. OVC—like most virtual caches—logically augments each virtually-tagged block with a physical tag to avoid deadlock issues with doing an address translation at eviction. This physical tag adds a small state (e.g., 28 bits on 544 bits state, tag, and data) and can either be

stored (a) in the L1 cache or (b) an auxiliary structure (not shown) that mirrors L1 dimensions, but is accessed only on less-frequent dirty evictions.

Coherence: L2 caches and beyond typically process coherence with physical addresses. To access virtually tagged L1 blocks, incoming (initiated by other cache controllers) back-invalidations and forwarded requests may require reverse address translation (physical to virtual). Reverse translation can be avoided by serially searching physical tags (added for cache block eviction) for all sets that might hold a block. Since OVC already provides the processor with an associative lookup on physical addresses, it associatively handles incoming coherence lookups with the same mechanism. For example, an incoming coherence request to the cache depicted in Figure 4-2, would simply access the physical tags in both banks (8-way total). Further, this action may be handled with an auxiliary structure (option (b) for handling eviction) and my empirical results find this occurs less than once per 1K L1 cache accesses due to L1 high hit rates and low read-write sharing. Note that, coherence messages received due to local cache misses (*e.g.*, data reply, acks) use miss-status handling register entries to find the corresponding location in the cache and hence do not require reverse translation lookup.

Space and Power Cost: As depicted in Figure 4-3, OVC's space overhead in the L1 cache stem primarily from the addition of an ASID (16 bits) and physical tag (28 bits) per cache block. The primary tag must be extended (8 bits) to accommodate larger virtual address tag. OVC also adds page permission/privileged bits (3 bits) and a global bit. This totals approximately 10% space overhead for the L1 assuming 64-byte cache blocks. Given that L1 caches comprise a small fraction of the total space (and thus transistor count) for the cache

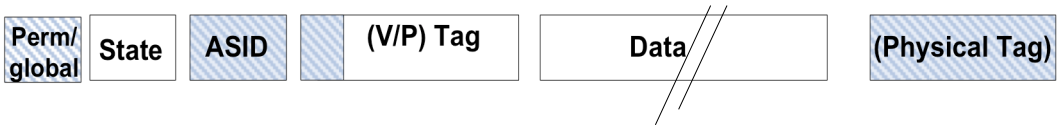


Figure 4-3. OVC overheads per L1 cache block. Additions are shaded.

hierarchy, which is dominated by larger L2 and L3 caches, the overall static power budget (which is grows roughly in proportion to transistor count) of the on-chip caches barely changes: ~ 1% overhead for the cache hierarchy in Table 4-6. Furthermore, the extra physical tag is accessed only for uncommon events: back invalidations, forwarded coherence messages and dirty evictions. L1 cache lookups and L1 cache hits do not accesses this physical tag. As a result, it causes ~ 1% energy overhead on L1 cache lookups, because most the energy is spent on data access, which has not changed. I will later show that the benefits of OVC out-weigh this overhead. I also note that cycle time is not affected as data lookup latency overshadows the tag lookup latency.

4.4.2 OVC Software

The operating system for OVC hardware has three additional responsibilities: (1) predicting when virtual caching of an address is desirable (safe and efficient); (2) informing the hardware of which memory can use virtual caching; and (3) ensuring continued safety as memory usage changes. I extend the Linux virtual-address allocator to address the first two and make minimal changes to the page-fault handler and scheduler for the third.

Deciding when to use virtual caches: The OS decides whether virtual caching may be used at the granularity of memory regions. These are an internal OS abstraction for contiguous virtual-address ranges with shared properties, such as for program code, the stack, the heap, or a

memory-mapped file. When allocating virtual addresses for a memory region the OS virtual-address range allocator predicts whether the region could have read-write synonyms (unsafe) or frequent permission/mapping changes (inefficient), and if so, uses addresses that allows physical caching and otherwise uses virtual caching.

While predicting future memory usage may seem difficult, I observe that the OS *already possesses* much of the information needed. The kernel virtual-address allocator defines flags specifying how the memory region will be used, which guides its assignment of page permissions for the region. For example, in Linux, the VM_PRIVATE flag indicates pages private to a single process, VM_SHARED indicates a region may be shared with other processes, and VM_WRITE/VM_MAYWRITE indicates that a region is writable. From these flags, the kernel can easily determine that read-write synonyms occur only if the VM_SHARED and VM_WRITE/ VM_MAYWRITE flags are set, which causes the kernel to use physical caching. For all other memory regions kernel predicts that use of virtual caching would not cause read-write synonyms. This enables a straightforward identification of which memory regions can use virtual caching.

Unfortunately, these flags do not provide hints about efficiency: some regions, such as transiently mapped files, may observe frequent page-mapping or protection changes (*e.g.*, through *mprotect()* and *mremap()*) that can be expensive with virtual caches. I thus add an additional flag, MAP_DYNAMIC, to the kernel allocator to indicate that the mapping or page permissions are likely to change. Applications can use this flag while allocating memory to indicate frequent protection/mapping changes or the need for physical caching for other semantic or performance reasons.

Communicating access type to hardware: The kernel uses the prediction techniques described above to select either virtually or physically cached addresses for a region. If MAP_DYNAMIC is specified, physical caching is used irrespective of the prediction. OVC minimally extends the OS virtual-address range allocator to allocate addresses from two non-overlapping address pools, partitions $\boldsymbol{P_{physical}}$ and $\boldsymbol{P_{virtual}}$ (described in Section 4.4.1), depending on whether physical or virtual caching is to be used.

Ensuring correctness: While the kernel only uses virtual caching when it predicts that conflicting synonyms will not arise, they may still be possible in some rare cases. First, the kernel itself may use temporary kernel address space synonyms to access some user memory (Section 4.3.1). Second, the kernel allows a program to later change how a memory region can be used (e.g., through Linux's *mprotect()* system call). OVC provides a fallback mechanism to ensure correctness in these cases by detecting when the change occurs, and then flushing the cache between conflicting uses of memory.

OVC inserts two checks into the Linux kernel for conflicting synonyms. Within the page fault handler, I added code to check whether a virtually cached page is being mapped with write permissions at another address in another process. If the above checks detect possibility of a conflicting synonym in the page-fault handler, the OS marks the process with write access to a synonym as *tainted*, meaning that when it runs, it may modify synonym pages. I modified the OS scheduler to flush the L1 cache before and after the tainted process runs. If hyper-threading is enabled, scheduler needs to prohibit tainted process from sharing the same core (and thus L1 cache) with another process. The current implementation however does not enforce this check

since simulated system does not employ hyper-threading. Further, I put a check in the Linux's kernel routine that creates temporary kernel mappings to user memory (e.g., *kmap(), kmap_atomic()* in Linux) and the kernel routine that grabs user-space page frames (e.g., *get_user_pages()* in Linux) to detect conflicting use of synonyms. The L1 cache is flushed before and after such conflicting uses.

For frequent and performance-sensitive synonym uses, such as direct I/O, a program can prevent these flushes by mapping I/O buffers using the MAP_DYNAMIC flag, which will use physical caching. However, even if a user fails to do so, the above mechanism ensures correctness. Further, note that it is possible to have read-write synonyms within single process's address space (*e.g.*, if same file is simultaneously memory mapped by a single process at different places in writable mode). If such cases ever occur (I have encountered none), I propose to turn off OVC capability (unset ***ovc_enable***) for the offending process.

4.5 Evaluation

In this section, I evaluate the OVC design. First, I will describe the baseline architecture for the experiments and the methodology. I will then provide quantitative results of the evaluation.

4.5.1 Baseline Architecture

I modeled a 4-core system with an in-order x86-64 CPU detailed in Table 4-6. The simulated system has two levels of TLB and three levels of caches. Each core has a separate L1 data and instruction TLB and a unified L2 TLB. The cache hierarchy has a split L1 instruction

Table 4-6. Baseline system configuration.

CPU	4-core, in-order, x86-64
L1 TLB	Private, Split Data and Instruction L1 TLB, 64 entries, Fully associative
L2 TLB	Private, 512 entries, 4-way set associative
L1 cache	Private, Data and Instruction L1 Cache, 32 KB, 8-way set associative
L2 cache	Private, 256KB, 8-way set associative
L3 cache	Shared, 8MB, 16-way set-associative, MESI Directory cache coherence

and data cache private to each core. Each core also has a private L2 cache that is kept exclusive to the L1 cache. The L3 cache is logically shared among all the cores, while physically distributed in multiple banks across the die.

4.5.2 Methodology and Workloads

I used x86 full system simulation with gem5 [14] to simulate a 4-core CMP with the configuration listed in Table 4-6. I modified the Linux 2.6.28-4 kernel to implement the operating system changes required for leveraging OVC. I used CACTI 6.5 [68] with the 32nm process for computing energy numbers. For TLBs, L1 caches, and L2 caches, I used high performance transistors ("itrs-hp"), while low static power transistors ("itrs-lstp") were used for L3. L1 and L2 caches lookup both tag and data array in parallel for providing faster accesses. However, L3 caches lookup the tag array and data array in sequence.

I used several of RMS workloads (*canneal, facesim, fluidanimate, streamcluster, swaptions, x264*) from Parsec [74]. I also used a set of commercial workloads: *SpecJBB* 2005 [85], a server benchmark that models Java middle-tier business-logic processing; *memcached* [65], an open source in-memory object store used by many popular web services including Facebook and Wikipedia; and *bind*, the BIND9 Domain Name Service (DNS) lookup service [22]. I also analyzed the open-source web browser Firefox [67] for synonym usages and TLB invalidation characterization. However, as an interactive workload, it does not run on the simulator.

4.5.3 Results

To evaluate OVC, I seek to answer three questions: (1) How much TLB lookup energy is saved? (2) How much of L1 cache lookup energy is saved? (3) What is the performance impact of the OVC?

The evaluation focuses on dynamic (lookup) energy as TLBs and L1 caches are frequently accessed, but relatively small, making OVC's static-energy impact insignificant.

Table 4-7. Percentage of TLB lookup energy saved by OVC.

	L1 Data TLB	L1 Instr. TLB
canneal	72.253	99.986
facesim	96.787	99.999
fluidanimate	99.363	99.999
streamcluster	95.083	99.994
swaptions	99.028	99.989
x264	95.287	99.304
specjbb	91.887	99.192
memcached	94.580	98.605
bind	97.090	98.310
Mean	93.484	99.486

Table 4-8. Percentage of accesses that use virtual caching.

	Data V-address access percentage	Instruction V-address access percentage
canneal	80.791	100
facesim	99.843	100
fluidanimate	99.925	100
streamcluster	98.575	100
swaptions	99.990	100
x264	99.933	100
specjbb	96.650	100
memcached	99.291	100
bind	98.97	100
Mean	97.116	100

TLB Energy savings: Table 4-7 shows the percentage of L1 data and instruction TLB dynamic energy saved by the OVC. I observe that more than 94% of the L1 data TLB energy and more than 99% of L1 Instruction TLB lookup energy is saved by OVC. To analyze this result, I first note that the cache accesses that use virtual addresses and hit in the L1 cache avoid using energy for TLB lookups. Table 4-8 shows the percentage of data and instruction accesses that can complete without needing address translation, while the L1 cache hit rates for accesses using virtual addresses are listed in Table 4-9. I observe that on average 97% of data accesses and almost 100% of instruction accesses complete without needing address translation, while a very

Table 4-9. L1 cache hit rates for virtual caching.

	L1 Data Cache	L1 Instruction Cache
canneal	0.894	0.999
facesim	0.969	0.999
fluidanimate	0.994	0.999
streamcluster	0.964	0.999
swaptions	0.990	0.999
x264	0.953	0.993
specjbb	0.950	0.991
memcached	0.952	0.986
bind	0.980	0.983
Mean	0.961	0.994

Table 4-10. Dynamic energy savings in caches and TLBs (percentages).

	L1 Data $ dynamic energy savings	L1 Instr. $ dynamic energy savings	TLBs + $ hierarchy dynamic energy savings
canneal	17.381	22.800	9.989
facesim	22.252	22.800	18.575
fluidanimate	22.727	22.801	30.672
streamcluster	21.805	22.802	16.709
swaptions	22.797	22.807	32.542
x264	27.737	23.230	25.446
specjbb	23.229	22.771	17.547
memcached	23.352	23.155	16.765
bind	22.812	22.784	28.283
Mean	22.624	22.883	19.546

high fraction these accesses (0.96 and 0.99 respectively) hit in the cache, saving TLB lookup energy.

L1 cache energy savings: OVC saves L1 cache lookup energy by accessing only a subset of the ways in a set when using virtual addresses (Section 4.4.1). Table 4-10 presents percentage savings in dynamic energy by OVC from opportunistic use of partial lookups (4-ways out of 8-ways) in the L1 cache. The second column shows that on average more than 22% of the dynamic energy spent on L1 data cache lookups is saved, while the third column shows similar savings for an instruction cache. The rightmost column provides a more holistic view of the

energy savings in the chip by showing how much of dynamic energy of TLBs and all the three levels of on-chip caches taken together is saved. On average, more than 19% of the dynamic energy spent on the on-chip cache hierarchy and the TLBs is eliminated by the OVC. The savings can be as high as 32% (*swaptions*) for applications with small working sets that rarely access L2 or L3 caches. In total, OVC saves a considerable portion of on-chip memory subsystem dynamic energy through lower associative L1 cache lookups and TLB lookup savings as these two frequent lookups account for most of the dynamic energy in the on-chip memory.

Performance impact: I quantify the performance implications of OVC in Table 4-11 that shows the number of misses per 1K cache reference (MPKR) for the baseline and the OVC L1 data and instruction caches. For the L1 data cache, the change in the number of misses is within a negligible 0.7 misses per 1K cache reference, while changes for instruction caches are even smaller. A couple of the workloads (*specjbb, memcached*) experience larger L1-D cache miss rate decrease with OVC (~2 misses per 1K reference, which translates to a minuscule hit-rate difference), while the L1 I-cache miss rate increases for one workload (*bind*). I note that cache hit/miss patterns are slightly perturbed due to use of a single bit from the virtual page number in selection of the bank where an access should go when virtual address is used under OVC. More importantly from Table 4-11 (right-most column), I observe that OVC hardly changes runtime compared to the baseline system (within 0.017%). The unchanged runtime, coupled with OVC's small static power overhead to the whole on-chip cache hierarchy (Section 4.4.1) indicates that OVC leaves the static power consumption of the on- chip memory subsystem largely unchanged while saving substantial dynamic energy. Furthermore, for these workloads, the operating system never needed to use the taint bit (Section 4.4.2) as they do not

Table 4-11. Miss ratios and runtime comparison between the baseline and the OVC.

	Baseline L1D MPKR	OVC L1D MPKR	Baseline L1I MPKR	OVC L1I MPKR	Normalized runtime
canneal	105.62	105.68	0.120	0.133	0.9994
facesim	30.476	30.613	0.084	0.093	0.9999
fluidanimate	5.735	5.622	0.003	0.006	0.9999
streamcluster	35.436	35.421	0.037	0.037	1.00001
swaptions	9.668	9.716	0.106	0.106	1.0004
x264	47.329	46.492	6.53	6.977	1.00099
specjbb	51.704	49.289	7.683	8.008	0.99330
memcached	49.699	47.349	14.235	13.947	0.99632
bind	20.527	19.630	13.981	16.893	1.00701
Mean	39.576	38.879	4.745	5.133	1.00017

use direct I/O or make system calls to conflicting change of page protection. Moreover, there were no cache flushes due to memory-mapping changes.

Selective raw numbers for this chapter appears in Appendix in page 163.

4.6 OVC and Direct Segments: Putting it Together

Direct segments reduce TLB misses while OVC reduces TLB and cache lookup energy. Thus an obvious question is, “Can direct segments and OVC work together?” In this subsection, I sketch one possible answer. However, the proposed technique is not evaluated.

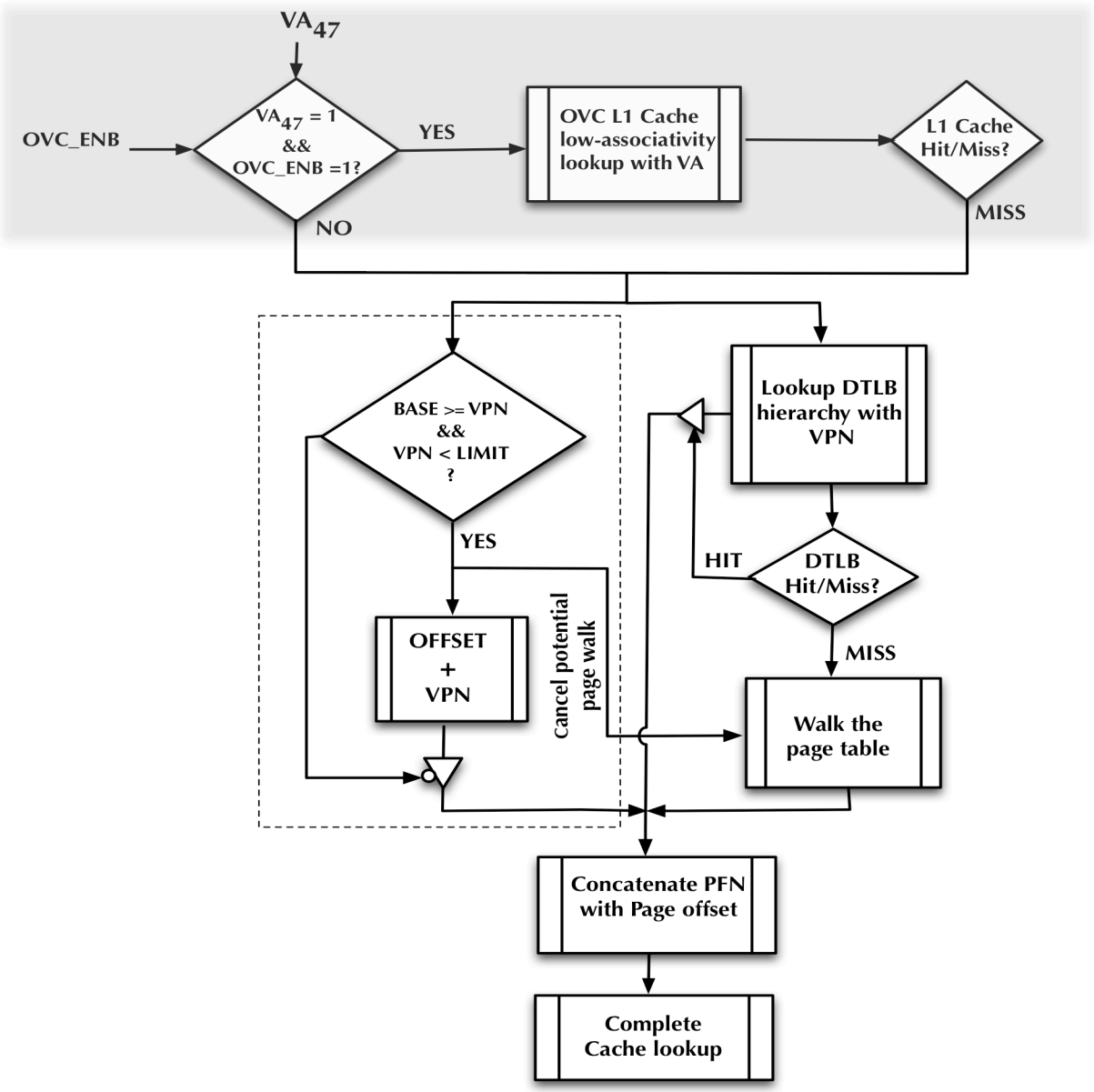


Figure 4-4. OVC with Direct Segments. OVC in shades. Direct segment additions within dotted box. VPN = Virtual Page Number, PFN = Physical Frame Number.

To save both energy and latency due to address translation, I propose combine the OVC and the direct segments in the way shown in the flowchart of Figure 4-4. Specifically, as with original OVC proposal, if the OVC is enabled (*OVC_ENB=1*) then a high-order virtual-address

bit (VA_{47}) decides whether to use virtual caching or physical caching. If virtual caching is used for a memory access and a L1-cache hit occurs, then the memory reference completes without requiring an address translation, as is the case with original OVC proposal. In case of an L1-cache miss or if the memory access is to the physically addressed region then an address-translation is needed. On an address translation the proposed technique employs direct segment's way of doing address translation – lookup direct-segment registers (BASE, LIMIT) in parallel to conventional TLB lookup. If the address falls within BASE and LIMIT register values then

One potential limitation however is that it may not be possible to use a direct segment to map all of the primary region (memory not needing paging features) if one part of the region is accessed using virtual caching and the other using physical caching in OVC. This limitation stems from the fact that OVC partitions the virtual address space between virtual caching and physical caching, while the direct segment needs a single contiguous virtual address range. However, if this is an important use-case then two sets of direct-segment registers – one set for virtually cached region and another for physically cached region -- can be introduced. Depending upon whether the address translation is needed for the virtually cached or physically cached region the corresponding set of direct segment registers can be accessed during the translation.

In summary, OVC can be combined with direct segments to provide reduce both address translation energy and latency.

4.7 Related Work

There has been decades of research on implementing virtual caches, which are summarized by Cekleov and Dubois for both uniprocessor [15] and multiprocessor systems [16].

Here I discuss a few of the most related work on virtual caches. I also discuss relevant work on reducing TLB power.

Goodman proposed an all-hardware solution for handling synonyms by introducing dual-tag store for finding reverse translations on possible synonyms [34], while a similar technique uses back-pointers in L2 physical caches for finding synonyms in L1 virtual caches [93]. Kim et al. proposed the U-cache in which a small physically indexed cache was added to hold reverse translations for pages with possible synonyms [51]. A few other works advocate for side-stepping the problem of synonyms by constraining sharing (and thus synonyms) through shared segments only [27,101] or through constrained virtual-to-physical memory mapping (page coloring) to ensure synonyms always fall in the same cache set [60]. Qiu et al. [76] proposed a small synonym lookaside buffer in place of a TLB to handle synonyms in a virtual cache hierarchy. On the other side of the spectrum, single address space operating systems like Opal [18,30] and Singularity [52] propose a new OS design philosophy that does away with private per-process address spaces altogether (and thus no possibility of synonyms). Although many of the above techniques for virtual caching are used in OVC; OVC exposes virtual caching as an optimization rather than a design point to leverage benefits of virtual caching when suitable and defaulting to physical cache when needed for correctness, performance or compatibility.

Several past hardware proposals addressed the problem of TLB power consumption through TLB CAM reorganization [46], by adding hardware filter or buffering for TLB access [17,43] or by using banked TLB organization [17,61]. Kadayif et al. [47] proposed adding hardware translation registers to hold frequently accessed address translations under compiler

directions. Ekman et al. [28] evaluated possible TLB energy savings by using virtual L1 caches as a pure hardware technique while also proposing a page grain structure to reduce the coherence snoop energy in the cache. Wood et al. [101] advocated doing away with TLBs by using virtual caches and using in-cache address translation. Jacob et al. [45] proposed handling address translation with software exceptions on cache miss to also get rid of the TLB. OVC, on the other hand, is a software-hardware co-design technique that aims to maintain full backward compatibility with existing software while opportunistically allow both TLB and L1 cache lookup energy reduction.

Woo et al. [100] proposed using bloom filter to hold synonym addresses to save L1 cache lookup energy by allowing lower associativity. Ashok et al. [5] proposed compiler directed static *speculative* address translation and cache access support to save energy. Different from their work, OVC does not burden the hardware with the onus of ensuring correctness for static miss-speculation; neither does OVC require recompilation of application to take advantage of OVC. Zhou et al. [104] proposed heterogeneously tagged (both virtual and physical tag) to allow cache access without TLB access for memory regions explicitly annotated by the application. Unlike their work, application modification is not necessary for OVC. Furthermore, OVC saves L1 cache lookup energy through reduced associativity. Lee et al. [53] proposed to exploit the distinct characteristics of accesses to different memory regions of a process (e.g., stack, heap etc.) to statically partition the TLB and cache resources to save energy. OVC does not require static partitioning of hardware resources and instead opportunistically use virtual caching to allow substantial energy benefits.

Some of older embedded processors from ARM also allowed virtual caches and flushed caches on context switch/page permission changes [94].

5

TLB Resource Aggregation

5.1 Introduction

Direct segments (Chapter 3) can eliminate most of the DTLB misses for big memory workloads that often have fairly predictable memory usage and allocate most memory early in execution. However, direct segments are less suitable under a more dynamic execution environment [102] where application characteristics are unknown a priori and applications can allocate/de-allocate memory frequently. In such a scenario, large-page support may lend more flexibility by avoiding memory fragmentation and by allowing memory overcommit through swapping pages to the disk. In this section, I proposed a *merged-associative TLB* that tries to improve large page support in commercially prevalent processors.

Large pages are one of the most widely deployed mechanisms to reduce TLB misses in current processors. A large page size maps a larger amount of contiguous and aligned virtual addresses into contiguous and aligned physical addresses. Thus, a given number of TLB entries

can potentially map a much larger amount of memory compared to the base page size (e.g., 512 times larger for 2MB page size vs. 4KB). This can potentially help reduce the number of TLB misses. Almost all processors today support multiple large page sizes. For example, x86-64 processors support 2MB and 1GB large page size beyond the 4KB base page size. ARM supports 4KB and 64KB page sizes. PowerPC supports 4KB, 64KB, 16MB, 16GB page sizes. UltraSPARC supports 8KB, 64KB, 4MB and 256MB pages.

However, use of large pages introduces its own set of difficulties − both software and hardware. In this chapter, I focus only on the hardware challenges. The use of large pages affects TLB designs. Larger page size introduces an *additional unknown* in the address translation process – the page size of the translation. This lack of knowledge about the page size makes it hard to determine the virtual page number for searching the TLB for a given virtual address. This in turn introduces TLB design tradeoffs among design complexity, performance [88] and power.

Commercial processors use two approaches to support multiple page sizes in the TLB. One approach to allow multiple page sizes in the TLB is to use a fully associative TLB, as often implemented in many AMD and IBM processors. The tag in each entry of the TLB contains page size information in addition to the virtual page number. The hit/miss logic uses page size information to select the actual page number for the virtual address tag comparison. A fully associative design looks up all its entries and thus a match will be triggered if desired address translation entry is present in the TLB, irrespective of the page size. In short, a fully associative TLB makes it easy to support multiple page sizes because it requires only minimal change in

TLB entries to include page size information and requires the corresponding logic to match page size information.

However, a fully associative design is often slower, more power hungry, and needs more chip area compared to a set-associative design with equal number of entries [98]. In addition, a fully associative design needs as many comparators as the number of entries to find a tag match. It also needs a wide (logical) multiplexer (input equal to number of entries) to select the matching entry. The high latency and power overheads make it hard to scale a fully associative TLB to more entries. To the best of my knowledge there has been no commercial processor with more than 64 entries in a fully associative TLB.

The second approach for supporting multiple page sizes uses set-associative TLBs. Set-associative TLBs can address scalability challenge of a fully associative design. Intel's prevalent commercial designs like Sandy Bridge and Ivy Bridge employ set-associative TLBs. Set-associative designs are more scalable compared to fully associative designs as the number of comparators does not grow with the number of entries but remains equal to the associativity.

However, a set-associative design for TLBs comes with its own drawbacks. In a conventional virtual memory system the *page size for mapping a given address is unknown until the address translation is itself complete*. This makes it difficult to select the address bits to index into a set-associative TLB. A part of the virtual page number needs to be used to index into the set-associative TLB. However, depending upon the page size the location of virtual page number in the address bits changes. For example, if a virtual address $VA_{n-1}VA_{n-2}....VA_0$ is mapped

using a 4KB page size then the virtual page number is $VA_{n-1}VA_{n-2}...VA_{12}$, while if the same address is mapped using a 2MB page then virtual page number is comprised of $VA_{n-1}VA_{n-2}...V_{21}$.

Intel's designs address this challenge through a *split-TLB design* where each different page size has its *own sub-TLB* and all sub-TLBs are accessed in parallel on each memory access [88]. At most one of the sub-TLBs can produce a hit since OS maps a virtual address using only one of the page sizes. Since each sub-TLB can only hold translations of a single page size, the indexing bits for a sub-TLB is fixed. The TLB resources are statically partitioned at the design time among the sub-TLBs for different page sizes.

However, such a split-TLB design has several potential disadvantages. First, static partitioning of TLB resources among sub-TLBs for different page sizes can lead to a performance anomaly where use of larger pages can degrade performance. The static partitioning of resources among the sub-TLBs reflects the chip-designer's expectation about the mix of page sizes used by the applications. Often fewer entries are provisioned for large page sizes compared to base (smaller) page sizes as each large page has greater reach and also since the use of large pages is less prevalent today compared to base pages. For example, Intel's recent Ivy Bridge processors have for 4 entries for 1GB pages while there are 64 entries for 4KB pages in the L1 data-TLB. Unfortunately, a diverse set of applications with wide-ranging behaviors is unlikely to match the chip-designer's intuition most of the time. Thus, an access pattern with low access locality may lead to more TLB misses with large pages than smaller pages due to the lower number of TLB entries available to larger pages – leading to a performance anomaly where use of larger pages could actually hurt performance. Indeed, Linux kernel developers are aware of

this performance unpredictability as evident in the following quote published in Linux Weekly News -- "differences in TLB structure make predicting how many huge pages can be used and still be of benefit problematic" [35]. VMWare also warns its users of possible performance degradation with use of large pages due to same reasons [90], while my own experiments on Intel's Sandy Bridge shows TLB performance anomaly where the application *CG* from NAS parallel benchmark suite incurs more TLB misses when using 1GB pages than with 2MB pages.

Second, a split-TLB design leads to *underutilization* of critical TLB resources. For example, an application that makes no use of large pages wastes TLB entries provisioned for large pages. In general, underutilization of TLB resources is possible whenever chip designer's expected usage mix of page size in applications mismatches with the actual page size mix.

Thus, ideally, one would wish to have a *single set-associative TLB* that avoids scalability limits of a fully associative TLB while accommodating PTEs for any page size *without statically partitioning the TLB resources among sub-TLBs for different page sizes*. Such a design can scale better than a fully associative TLB while allowing resource aggregation to provide a larger effective TLB size than a split TLB design, avoid performance unpredictability with large pages and reduce power waste due to accessing multiple TLBs.

In this work, I propose a hardware-software co-design that logically provides a single set-associative TLB design that can hold a flexible mix of PTEs for any page size (e.g., 0, some, or all entries can map 2MB pages). Another important goal of my proposed design is to maintain backward compatibility with unmodified OSes and applications. In particular, I start from a commercially prevalent split-TLB design but overcome its shortcomings like static partitioning

of resources wherever possible, while falling back to the baseline split-TLB design when necessary for compatibility. They key idea is to split the *abundant virtual address space of 64-bit address space among the page sizes instead of splitting more constrained hardware TLB resources*. I call this a *merged-associative TLB* since it can merge or aggregate hardware TLB resources of provisioned for different page sizes.

At the high level, the merged-associative TLB proposal put the onus on the operating system to keep memory mapped using distinct page sizes in distinct regions of a process's virtual address space. The hardware TLB then interprets the page size used to map a given virtual address by inferring the address region it is coming from – allowing it to correctly calculate the virtual page number even before the address translation process is initiated. This, in turn, enables the hardware TLB to *logically* aggregate the entries of sub-TLBs in a split-TLB design in to a single logical set-associative TLB that can hold PTEs for any page size.

More specifically, the operating system splits virtual address space of a process in $\boldsymbol{k}$ equal-sized regions. Each of these regions contains memory mapped with a single page size. In effect, this establishes a correspondence between virtual addresses and the page size used to map these addresses. A typical value of $\boldsymbol{k}$ could be 4, which my current prototype implements. The hardware exposes a k-entry table for each core (or each hardware context if multi-threaded) to the OS, called the *region table* or RT. The $\boldsymbol{n}^{\text{th}}$ ($\boldsymbol{0} \leq \boldsymbol{n} < \boldsymbol{k}$) entry of the region table should contain the encoding for the page size used for mapping $\boldsymbol{n}^{\text{th}}$ virtual address region in the address space of the currently running process in the given core. This encoding acts as hint to the hardware about

the page size used for mapping a virtual address and this information is available before the address translation is started.

The hardware of a merged-associative TLB *logically* merges the separate sub-TLBs in a split-TLB design to provide an illusion of a larger combined TLB shared by all page sizes. By logically combining sub-TLBs, merged-associative TLB (*mTLB*) aggregates TLB resources when possible and falls back on the conventional split-TLB when needed for compatibility. The mTLB logically appends sets (rows) of one sub-TLB after another sub-TLB. For example, in a split-TLB design as in Intel's Sandy Bridge, with 64-entry, 4-way set-associative TLB for 4KB pages and 32-entry, 4-way set-associative TLB for 2MB pages are logically coalesced to form a 96-entry, 4-way set associative merged-associative TLB. To index into this logical merged-associative TLB structure two-step process is followed. First, on each memory access the region table is looked up using top $\boldsymbol{log_2 k}$ bits of the virtual address to find out the page size used to map the given address. It then uses this page size hint to calculate the correct virtual page number (VPN) and finally indexing into the correct set using part of the VPN.

To allow backward compatibility with unmodified OSes, the default for the encoding for the page size in the region table is set to a special value that indicates "*unknown_pagesize*". If for a given virtual address the corresponding region table entry contains "*unknown_pagesize*" then a merged-associative TLB reverts to a split-TLB design where all possible sub-TLBs are searched for a potential match. Further note that, although not fundamental to the design of merged-associative TLB, logically aggregated TLB can have non-power-of-2 number of sets due to an

uneven number of sets in sub-TLBs of typical split-TLB design. The indexing scheme of a merged-associative TLB may thus need to handle this situation.

A merged-associative TLB alleviates several shortcomings of a split-TLB and a fully associative TLB designs. First, unlike a split-TLB design unpredictable performance loss with use of large pages is not possible as the number of entries for large pages is never less than that of base pages. Second, by enabling resource aggregation across multiple TLBs, a merged-associative TLB addresses the potential underutilization of TLB resources. Finally, since merged-associative TLB is built essentially out of set-associative TLBs, it provides better scalability in terms of both latency and power consumption compared to a fully associative design. Merged-associative TLB can enable provide energy benefits over a split-TLB design since only one instead of multiple TLBs are accessed.

A merged-associative TLB has its own share of shortcomings, though. First, a merged-associative TLB requires modifications to the OS to partition a process's virtual address space based on page sizes. It reverts to split-TLB with an unmodified OS where all the sub-TLBs are accessed for finding a potential match. Further, a merged-associative TLB requires that the page size used to map a given virtual address should be fixed at the time of the memory allocation request. Thus a merged-associative TLB reverts to a split-TLB design with online page size promotion or demotion where a given virtual address can be mapped using different page sizes during its lifetime.

In summary, following are the contributions of this work.

- I demonstrate that a merged-associative TLB design could alleviates performance anomalies possible with use of large pages in a split-TLB design.
- I demonstrate that a few applications like *graph500* [36] -- a graph-analytics application -- can observe substantial reduction in number of TLB misses with merged-associative TLB design through resource aggregation.

5.2 Problem Description and Analysis

In this chapter, I will first recap large-page support in contemporary x86-64 processors and then describe and analyze tradeoffs of currently available TLB designs that support multiple page sizes. I will then state the problem statement addressed in the work described in this chapter.

5.2.1 Recap: Large pages in x86-64

TLB reach is the total memory size mapped by a TLB (number of entries times their page sizes). Large TLB reach tends to reduce the likelihood of misses. Larger page sizes – that map larger amount of contiguous virtual address to larger amount of physical address space -- were introduced to reduce number of TLB misses without needing to scale TLB entries.

The page sizes are defined in the Instruction Set Architecture (ISA) of a processor. For example, x86-64 supports 4KB (base page), 2MB and 1GB page sizes while ARM-64 would support 4KB (base page) and 64KB. The OS also needs to extend support for large pages to enable application use larger page sizes. Further, to make effective use of multiple page sizes it

Table 5-1. Comparative Analysis of set-associative and fully associative TLBs. 4-SA = 4-way set-associative, FA= Fully associative.

	64 Entries		128 Entries		256 Entries	
	4-SA	**FA**	**4-SA**	**FA**	**4-SA**	**FA**
Access Latency	0.14 ns	0.39 ns	0.15 ns	0.47 ns	0.17 ns	0.67 ns
Cycle time	0.16 ns	0.49 ns	0.16 ns	0.58 ns	0.18 ns	0.77 ns
Dyn. Access Energy(nJ)	0.003 nJ	0.008 nJ	0.004 nJ	0.016 nJ	0.006 nJ	0.031 nJ
Static Power (mW)	1.72 mW	3.87 mW	2.69 mW	7.57 mW	4.81 mW	14.37 mW

needs to be decided which page size to use for mapping which part of memory usage of an application.

5.2.2 TLB designs for multiple page sizes

Supporting multiple page sizes leads to important TLB design tradeoffs [88]. In this section I discuss different TLB designs and their tradeoffs.

Fully associative TLB design: As described in Section 5.1, a fully associative TLB design offer a straightforward way to allow multiple page sizes by adding page size information at each entry. The number of virtual address bits in the tag is determined by the smallest page size. For each TLB entry the hit/miss logic then selects the actual virtual page number to be compared based on the page size information of the TLB entry. For example, let us assume there is a two-entry fully associative TLB. Further let us assume that the 0^{th} TLB entry (T0) contains a mapping for 4KB page size and 1^{st} TLB entry (T1) contains a mapping for a 2MB page size. If a

48 bit address 0x7FFFFFAB0128 is to be looked up in the TLB, then the hit/miss logic will mask last 12 bits of the lookup address (page offset for 4KB page) and use 0x7FFFFFAB0000 for TLB entry T0, while for entry T1 it will mask last 21 bits of the lookup address (page offset for 2MB page) to use 0x7FFFFFA00000 for a possible match in entry T1. Since every TLB entry is searched in a fully associative design for possible match, the desired entry, if present, would certainly produce a hit, irrespective of page size of the mapping. Thus it's straightforward to extend a fully associative TLB to support multiple page sizes.

Unfortunately, fully associative structures are slower and require more area and power than a set-associative structure. Since a fully associative TLBs are usually implemented as content-addressable-memory or CAM, it needs as many comparators as number of entries and wide multiplexer. This makes a fully associative design often hard to scale. To quantify the tradeoffs between fully associative and set-associative TLB design, I used CACTI 6.5 [68] to estimate latency, area and power of various TLB designs. Table 5-1 lists this comparison of a fully associative and a set-associative TLB designs for various sizes. For a typical 64-entry TLB, the access time for a fully associative design is almost 2.8-times that of a 4-way set-associative design. When the TLB size is scaled to 256 entries the access latency of a fully associative design is even higher at 4-times of that of a set-associative design. This demonstrates the poor scalability of a fully associative design. Further, as discussed in the Chapter 4 (OVC) that the TLB lookup energy can be non-negligible fraction of core energy dissipation [9,83]. Table 5-1 shows that a 64-entry fully associative TLB spends 2.67X more dynamic energy on each access compared to a 4-way set-associative design. Static power and cycles time also scale poorly in a

Table 5-2. Number of TLB misses for 1K memory accesses on Intel Sandy Bridge machine.

Page / Size	4KB	2MB	1GB
NPB:CG	279.5	42.1	130.7
Nano-benchmark (Fig. 1)	0	487.8	490.7

fully associative design. In summary, a fully associative TLB design simplifies supporting multiple page sizes but can incur more latency and access energy.

Split TLB design: Intel's recent designs (e.g., Nehalem, SandyBridge, IvyBridge) instead adopt set-associative TLBs. However, to overcome the challenge posed by the lack of knowledge about the page size during the translation such designs employ separate sub-TLBs for each different page sizes. Each of these sub-TLBs is searched in parallel (and thus called split-TLB design) to find a hit/miss. Note that since a virtual address can be mapped using only one page size at most one of the TLBs can produce a hit.

Unfortunately, a split-TLB has several drawbacks. First, a split-TLB design can lead to performance unpredictability where use of larger page size can lead to more TLB misses than smaller page sizes. The potential increase in the number of TLB misses is due to asymmetric number of TLB entries available for different page sizes. For example, each core of Intel's recent Ivy Bridge architecture implements 64-entry L1-TLB for 4KB pages while there are only 32 entries for 2MB pages and 4 entries for 1GB pages. While TLBs of larger pages have larger

```
array[] = malloc(64×sizeof(long pointer)); //allocate array of 64 pointers

//allocate memory that the pointers would point to
for (int i=0; i < 64; i++) {
if (use_1GB_page)
      array[i] = mmap_with_1GB_pages(1GB); // mapped with 1GB pages
else if (use_2MB_pages)
      array[i] = mmap_with_2MB_pages(1GB); // mapped with 2MB pages
else
        array[i] = allocate_using_4KB_pages(1GB); // mapped with 4KB pages
}
//Loop num_iterations times and access the array entries
for(int i=0; i < num_iterations; i++) {
   //Inner loop to stream through the array
    for (int j=0; j < 64; j++) {
     //Offset makes sure that for each page size accesses do not conflict on
     // same set of a TLB
      if (use_1GB_page) {
            offset = j*64; //64 bytes offset
        }else if(use_2MB_page) {
                  offset = (j*2M+ j*64); //2MB + 64 bytes offset
            }else if(use_4KB_page) {
                  offset = (j*4KB + j*64) //4KB + 64 bytes offset
        }
       //Touch one cache line of a single page within each of 64 elements
       result = (array[j] + offset); //Access the array
   }
}
```

Figure 5-1. Pseudo-code for nano-benchmark with performance pathologies with split-TLB design.

TLB reach (number of entries × page size), the smaller number of entries can increase misses due to certain sparse access patterns. Although uncommon, such cases indeed exist. I found that the CG (Conjugate Gradient) workload from NAS parallel benchmark suite incurs far a higher number of TLB misses if 1GB pages are used compared to 2MB pages while running on Intel's Sandy Bridge processor. Table 5-2 (first row) lists TLB misses per 1K memory accesses for the workload CG for different page sizes – 4KB, 2MB and 1GB. The number of TLB misses jumps nearly 3 times when 1GB pages are used instead of 2MB pages.

To better understand this apparent anomaly, I designed a nano-benchmark that streams through an array of elements and incurs different number of TLB misses for different page sizes. The pseudo-code of the nano-benchmark is shown in Figure 5-1. The nano-benchmark allocates an array with 64 pointers each of which points to 1GB of memory that is allocated using 4KB, 2MB or 1GB page sizes. The number of entries in the array is fixed at 64 since the largest number of entries in L1-TLB in the experimental machine is 64 (for 4KB pages). This ensures that the allocated array would have enough number of pages to fully populate the TLBs. The program then loops through (inner loop) all 64 elements of the array while loading an address from each unique element. The program calculates different offsets for each page size, which are then added to the base address of each element in the array. The offset has two purposes. First, it makes sure that each access goes to consecutive set (row) in the TLB instead of conflicting within the same set. For example, the virtual address of the base of each element in the array is 1GB apart and 4KB PTE entries mapping these base virtual address of each element would map to the same 0^{th} set (row) of a 4KB TLB. To avoid this and make use of the full capacity of the TLB, the loop instead access pages in each element that maps on the consecutive sets of the TLB. This is accomplished by adding an offset equal to page size for 4KB and 2MB page sizes. For, when 1GB pages are used no offset needs to be added since the base of each element itself is 1GB apart. Second, a 64-byte offset is added to make sure the loop accesses consecutive cache blocks within a page to avoid similar set-conflicts in the L1 cache. The loop iterates for ten million times in the experiments.

The Table 5-2 (second row) lists the number of TLB misses per 1K memory accesses for the nano-benchmark using different page sizes. While with 4KB pages there are no measurable TLB misses, with 2MB and 1GB pages nearly every other memory access misses in the TLB. In the case of 4KB pages all 64 pages of the workload fit into the TLB. However, since the TLBs for 2MB and 1GB pages can accommodate at most 32 and 4 entries, respectively, it incurs a large number of TLB misses. This nano-benchmark shows that even with simple access pattern, use of large pages can incur many more of TLB misses than with the base page size. This unpredictability in number of TLB misses with large pages demonstrates one of the key drawbacks of the split-TLB design.

Further, static partitioning of critical TLB resource among page sizes disallows resource aggregation. For example, if an application uses only 4KB pages it wastes hardware dedicated to hold 2MB page translation. Finally, the split TLB design wastes power. On every memory access TLBs for every page size is looked up in parallel, while at most only one of them can produce a hit. As discussed in Chapter 4 TLB energy dissipation is non-negligible [9] and TLB can become hot spot in a chip design due to high power density [75], eliminating unnecessary TLB lookups could be important.

5.2.3 Problem Statement

In view of above discussion it is obvious that ideally one would like to have *a single set-associative* TLB that can efficiently hold PTEs for *any page size*. Compared to a split set-associative TLB design it could better aggregate critical TLB resources and avoid performance unpredictability due to large pages. Compared to a fully associative design such a design should

incur much lower latency per access and scale better while helping to avoid design constraint of implementing a CAM on the critical path of every memory access.

5.3 Design and Implementation

The goal of our proposed merged-associative TLB (mTLB) design is three fold.

- mTLB should prevent performance unpredictability possible with use or large page sizes in a split-TLB design while avoiding hard-to-scale fully associative designs.
- mTLB should allow resource aggregation across separate TLBs for each page size to increase TLB reach.
- Finally, mTLB should preserve backward compatibility to allow unmodified OSes and applications to run as before and could fallback to conventional split-TLB design when necessary.

To achieve the above-mentioned goals, I propose a hardware-software co-design that extends split-TLB design. At the high level, the hardware enables logical aggregation of separate sub-TLBs in split-TLB design allowing a TLB entry to hold translation for any page size. The OS segregates address mappings for a given page size in non-overlapping. This enables virtual address ranges to act as a hint for the page size used to map a given address.

5.3.1 Hardware: merged-associative TLB

The goal of the merged-associative TLB is to enable *logical aggregation* of separate set-associative L1 sub-TLBs of a split-TLB design. Figure 5-2 depicts the high level view of how a merged-associative TLB aggregates the sub-TLBs of a split-TLB design.

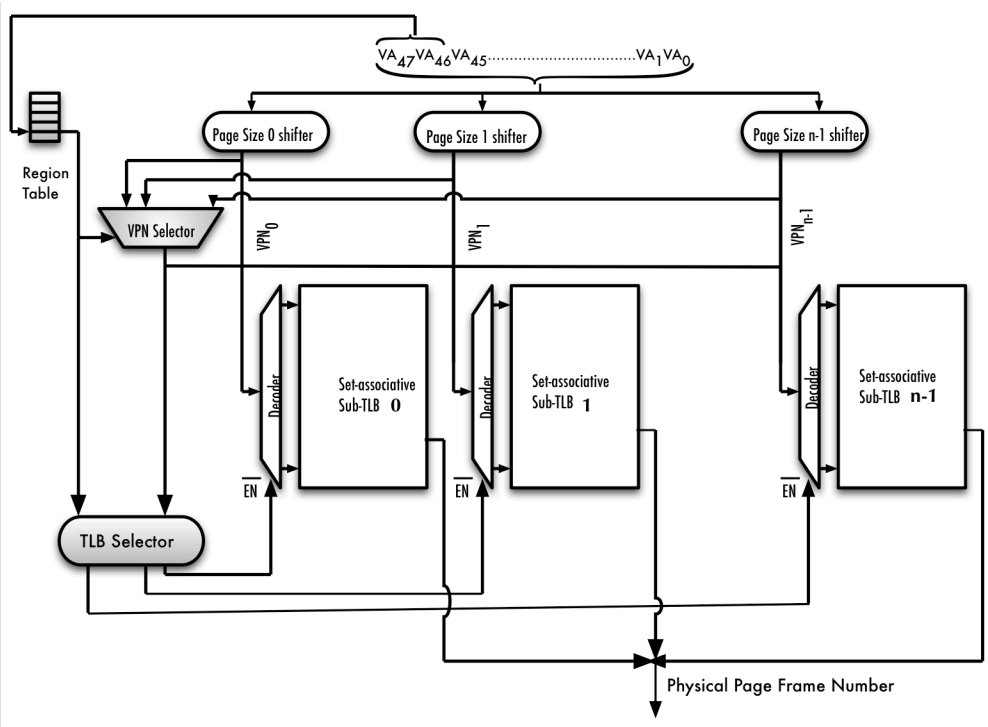


Figure 5-2. Logical view of Merged-associative TLB to aggregate the sub-TLBS of a split-TLB design. Added elements for merged-associative TLB is in shades.

As shown in the figure, let us assume there are '*n'* distinct page sizes with '*n'* sub-TLBs in a split-TLB design. In a conventional split-TLB design, all 'n' possible virtual page numbers (VPN_0, VPN_1,...,VPN_{n-1}), are formed by shifting the bits of the virtual address according to the page sizes. The VPNs thus formed, are then used to look up their corresponding sub-TLBs, in parallel.

In a merged-associative TLB, the key challenge in aggregating sub-TLBs of split-TLB design is to correctly determine the virtual page number in the absence of page size information. To overcome this challenge the mTLB requires that the OS partition a process's virtual address space and make sure that each address partition contains memory mappings from only one page

size. To achieve this, the virtual address space of a process is divided into *k* ($k=2^m$, a typical value of k is 4) equal sized partitions and a tiny hardware lookup table with m entries, called the region table or RT, is added to each core as depicted in Figure 5-2. Each entry in this table provides the page size used to map a virtual addresses in a given address region. For example, if there are three possible page sizes supported in the architecture then a 2-bit entry can be used to encode one of the three possible page sizes and a default value when page size is *unknown*. This table is called region table or RT. The region table is indexed by high order *m* bits of the (implemented) virtual address (e.g., bit $48\text{-}47^{th}$ of the virtual address when m=2). Thus a typical region table would be only need 8-bits and thus incur negligible access latency or power overhead. Larger number of partitions could allow the OS finer grain of control on addressing mapping, however at the cost of larger lookup-table size.

As depicted in the Figure 5-2, the VPN selector – which is essentially a multiplexer – uses the page size information available in the region table entry indexed by the higher order bits of the virtual address to select the correct virtual page number. This virtual page number is broadcasted to all sub-TLBs. However, unlike the split-TLB design only one of the sub-TLBs will be activated as decided by the TLB selector (Figure 5-2), which logically aggregates the sub-TLBs. The virtual page number is fed to the TLB selector and the separate sub-TLBs of split-TLB design works as banks of the logically aggregated merged-associative TLB. The TLB selector logic uses a few bits from the virtual page number to decide which bank in the logically aggregated TLB to index into. For example, let's assume n=2 and each of the two sub TLB has 64 entries, and is 4-way set-associative. Thus, each sub-TLB has 16 sets (row) and requires 4 bits

of the virtual page number for indexing. The TLB selector then uses 5th least significant bit of the VPN to choose one of the two sub-TLBs to enable and within each sub-TLB four least significant bits are used to index into a set (row) of the TLB. Thus, in this way logically a 128-entry (64 + 64) merged-associative TLB is formed.

As mentioned in Section 4.1 a merged-TLB maintains backward compatibility with unmodified OS. It does so by allowing a region table (RT) entry to specify "*unknown_pagesize*". For example, in x86-64 a 2-bit RT entry could the page size information. Three of the four possible bit combinations (00, 01, 10) can encode three possible page sizes in x86-64 system while the code "11" indicates unknown page size. If the RT entry corresponding to virtual address range is "11" then the TLB lookup proceeds as if split-TLB design is used. All the sub-TLBs are searched in parallel for the desired address translation. Thus, even with an unmodified OS the merged-associative TLB operates correctly; albeit without any benefit over a split-TLB design.

Non-power-of-2 entries: Often the split-TLB designs have sub-TLBs with unequal number of entries. For example, Intel's Ivy Bridge's split-TLB design has 64 entries for 4KB pages, 32 entries for 2MB pages and 4 entries for 1GB pages. In such scenario, the merged-associative TLB (mTLB) could logically aggregate 96 entries (64 +32), while excluding 4 entries for ease of implementation of TLB selector logic. For sake of simplicity of implementation, the TLB selector could divide the address space equally between the sub-TLBs of unequal size using higher order bits of the virtual address. This could lead to loss of opportunity to save more TLB misses in a merged-associative TLB due to extra indexing-conflicts but still is strictly better than

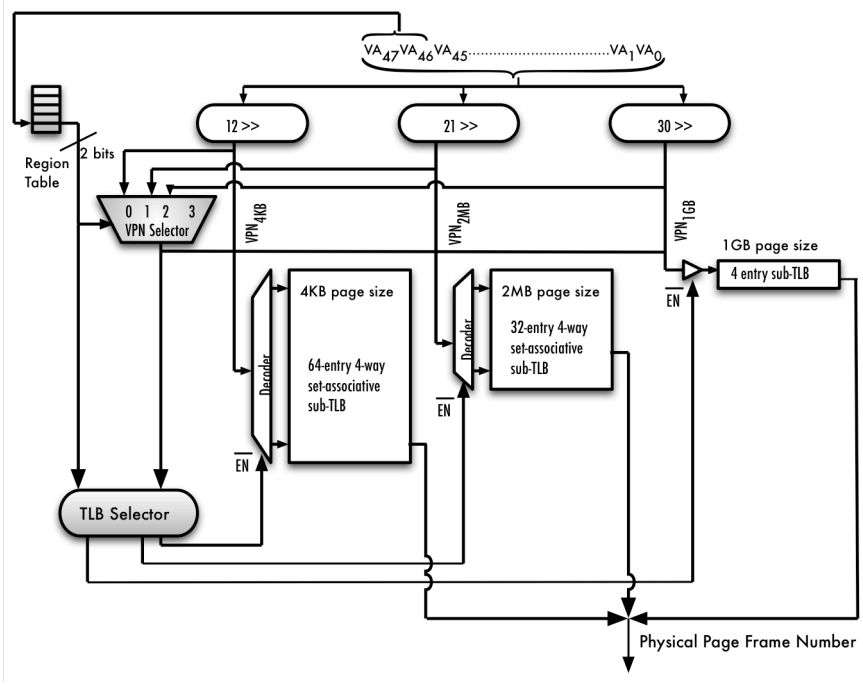


Figure 5-3. Logical view of merged-associative TLB for Intel's Sandy Bridge L1-TLB hierarchy.

split-TLB designs due to larger aggregated number of entries. My current implementation (evaluated in Section 5.5) logically merges Intel's Sandy Bridge design into a 96 entry merged-associative TLB in this fashion. Figure 5-3 depicts the logical view of such merged-associative TLB hierarchy built out of Intel's Sandy Bridge L1-TLBs.

In summary, a merged-associative TLB enables the logical aggregation of separate TLBs under split-TLB design when page size information is made available through virtual address space partitioning. By default, the contents of the region table are initialized with each entry indicating page size is unknown. Thus, a merged-associative TLB design defaults to a conventional split-TLB design.

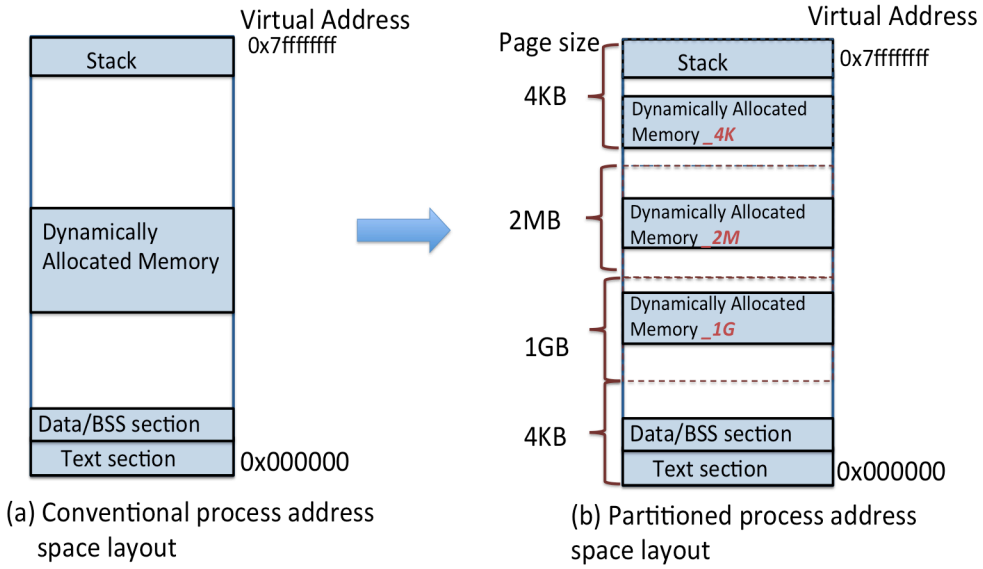


Figure 5-4.Virtual address layout of a process.

5.3.2 Software

The operating system (OS) needs modifications to its virtual-memory management to make use of a merged-associative TLB. Applications remain unchanged since the application-binary-interface (ABI) is unaltered.

The OS has two primary responsibilities. First, the OS makes sure that memory allocation requests for different page sizes go to different partitions of the virtual address space. Second, the OS is responsible for setting up the region table used by mTLB to set the e page size used to map the addresses in each range.

A typical virtual address space of processes running under Linux (x86-64) is 128TB in size and the conventional use of address space is depicted in Figure 5-4(a). As depicted, the lower portion of the address space is used for code, data and BSS segment (used for compile

time allocations and constants), while the top part is used for keeping the process stacks. These memory regions are usually always mapped using the base page size. The dynamically allocated memory is put in the middle part of the virtual address space. Dynamically allocated memory include heap allocation and memory allocated using *mmap(),* and *shmget()* system calls in Linux. Dynamically allocated memory can be mapped using base or larger page sizes. The goal of the software prototype is to enable this common use of large pages while partitioning the virtual address space according to page sizes.

I modified Linux 3.5.3 to partition the virtual address space to establish correspondence between an address partition and page size used to map addresses in the partition. At the boot time, the OS reads the number of partitions (‘*k’*) enabled by the mTLB (i.e., number of entries in the region table) by reading a specific model-specific-register (MSR). The OS then logically partitions the virtual address space of a process in up to ‘k’ equal-sized partitions and designates the page size for each partition. The current prototype logically partitions the virtual address space of each process in four equal-sized partitions (i.e., ‘k’=4) as depicted in Figure 5-4(b). In the current implementation each of these partitions are 32TB in size (128TB/4). The top and bottom most partitions are mapped with base page size (4KB). This enables ease of implementation as these partitions contain process stack, code, bss and data segment, which are mapped using base page size. As shown in the Figure 5-4(b) the dynamically allocated memory is put in to different partitions of the virtual address space depending upon the page sized used for mapping the allocations.

More specifically, I modified Linux system calls *mmap*() and *shmget*() such that depending upon requested page size virtual addresses from different address partition are used. Note that memory allocation requests for large pages have to go through either mmap() or shmget() system calls in Linux. Further, I modified *brk()* system call in Linux, which is used to allocate memory on the heap, to ensure that process heap is assigned virtual addresses from the bottom most address partition that is set aside for base page size. These simple modifications ensured that addresses mapped with different page sizes belong to non-overlapping designated virtual address partitions. Linux kernel also makes use of large pages. For example, in x86-64, Linux uses large pages to map entire physical memory into its "direct-mapped" virtual address space. In this work however, I only consider user-space memory usage.

The second responsibility of the OS is to populate the region table (Section 5.3.1) that contains the correspondence between the virtual address partition and the page size. The content of the RT is part of process context in the process control block or PCB. When a process is first created, the kernel initializes a new field in the PCB corresponding to store the virtual address partitioning as depicted in Figure 5-4(b). The initial content of this field reflects the OS's default correspondence between the virtual address partitions and the page sizes used to map them. The OS memory allocator also consults this field on an application memory request to return virtual addresses in accordance to the intended correspondence between the virtual address ranges and the page size. On a process context switch the contents of the hardware RT is switched from the old process to that of the new process. I also note that unmodified OS can run correctly with the

merged-associative TLB. However, since all page size information in the RT will be “unknown”, it degenerates to the split-TLB design.

5.4 Dynamic page size promotion and demotion

The merged-associative TLB proposal relies on page size information being available at the time of memory allocation so as to assign virtual addresses from correct partition. Unfortunately, dynamic page size promotion or demotion changes the page size for a virtual address during its lifetime, and is thus incompatible with the merged-associative TLB proposal. In this section, I describe how a merged-associative TLB could co-exist with dynamic page promotion or demotion by reverting back to split-TLB design.

Recently, Linux enabled Transparent Huge Page or THP [23], which enables opportunistic allocation of large pages and dynamic page size promotion or demotion without application knowledge. THP achieves this in two ways. First, on a memory allocation request for larger amount of memory (e.g., > 2 MB), the THP tries to use large pages to map these large allocations. Second, THP periodically scans process’s address space to find contiguous allocations in virtual address space mapped with small pages that could be mapped with large page and if large physical page(s) are available then it does the remapping with large pages. This process is called page promotion. Finally, THP can also demote the page size whereby it remaps a memory region with smaller page size that was earlier mapped with large pages. Page size demotion happens if a memory region needs to be swapped out or page protection is changed for portion of the memory region under consideration.

The goal of this proposal is to ensure that mTLB does not perform worse than the baseline split-TLB design under online promotion or demotion of page size. This is achieved through two mechanisms. The hardware first detects when the expected correspondence between the virtual address and page size is violated and then take corrective action to ensure that mTLB defaults to the split-TLB design for the offending virtual address region.

If the actual page size of a mapping of a given address differs from the expected page size, the mTLB will index into wrong place in the set-associative TLB. Thus, even if the PTE for the requested address is in the TLB, a miss will be triggered. In x86-64, on a TLB miss a hardware page table walk is initiated which locates the desired entry in the kernel memory and loads it into the TLB. In my proposed design, the page walker observes that the predicted page size is different from the page size of the correct PTE that it located in the memory. This indicates that correspondence between the virtual address region where the given address falls and the page size as noted in the lookup table is stale. At this point the hardware alters the region table contents to change the entry corresponding to the offending address region to "*unknown_page_size*". As described in Section 5.3.1, all the sub-TLBs are searched in parallel

Table 5-3. Baseline TLB hierarchy.

Core	6 core, x86-64
L1-TLB	64 entry, 4-way set-associative for 4KB pages 32 entry, 4-way set-associative for 2MB pages 4-entry, fully-associative for 1GB pages
L2-TLB	512-entry, 4-way set-associative for 4KB pages

from now onwards for any address is the offending address region. Thus, the scheme works correctly as in a split-TLB design, albeit without the benefit of the mTLB.

5.5 Evaluation

In this section I evaluate merged-associative TLB design against baseline split-TLB design and fully associative TLB design.

5.5.1 Baseline

I model the baseline TLB hierarchy after that of Intel's Sandy Bridge processor. Table 5-3 details the TLB hierarchy. Intel's Sandy Bridge processor employs a split-TLB design. Each core has three separate L1 Data sub-TLBs for each of the two different page sizes – 4KB, 2MB and 1GB. Larger number of TLB entries is statically allocated to smaller page sizes while large pages get smaller number of TLB entries. There is also a L2-TLB only for 4KB pages, which is accessed on a miss in L1-TLB. The L2-TLB behavior remains unchanged with mTLB.

5.5.2 Workloads

The workloads used in the evaluation includes commercial workloads like MySQL running TPC-C and memcached, as well as scientific workloads like BT and CG from NAS benchmark suite. The graph500 workload was run with graph of size approximately 32GB. The MySQL workload used 1000 warehouses (100GB). The memecached server used 32GB of in-memory object cache. The NAS workloads (BT and CG) used class "D" input set. I also evaluated the nano-benchmark presented in the Section 5.2.2.

5.5.3 Methodology

Analysis of TLB behavior needs to run large memory workloads for sufficient amount time to characterize enough number of TLB misses. Unfortunately cycle-accurate full-system simulators are too slow for simulating large memory workloads for reasonable time. More importantly, full system simulators add 2-3X memory overhead compared to the memory size of the simulated system. This makes it hard to simulate large memory workloads. To avoid the above issues I wrote a fast TLB simulator based on PIN dynamic binary instrumentation [57] to yield TLB miss rates. More specifically, the TLB simulator enables us to measure TLB miss rates for different TLB designs. This simulator observes only user-level memory access only and thus the effect of system code is not captured.

5.6 Results

In this section I present results of the evaluation. More specifically I try to answer following questions.

1. How much the TLB reach is increased by mTLB?
2. Can mTLB avoid TLB performance unpredictability when using larger page sizes?
3. Can mTLB enable additional performance benefit through resource aggregation?

To evaluate the efficacy of the different TLB organization I used the *number of TLB misses per 1K memory accesses* or MPKA as the metric. I measured the TLB misses that miss in both the levels of TLB hierarchy.

I evaluated three L1-TLB designs as follows.

Table 5-4. Ideal L1-TLB reach for different TLB configurations and page sizes. FA-TLB stands for fully associative TLB.

	Split-TLB	FA-TLB (64 entry)	FA-TLB (96 entry)	Merged-associative TLB
All 4KB	256KB	256KB	384KB	384KB
All 2MB	64MB	128MB	192MB	192MB
All 1GB	4GB	64GB	96GB	96GB

1. **Split-TLB:** This baseline design emulates split-TLB design as found in Intel's Sandy Bridge and as described in Table 5-3.
2. **Fully associative (FA):** I simulated fully associative TLB design with 64 and 96 entries.
3. **Merged-associative TLB:** This simulates the proposed merged-TLB that logically aggregates the resources of the split-TLB design to enable 96-entry L1-TLB for any page sizes. It logically aggregates 64 entries of the sub-TLB for 4KB pages with 32 entries of the sub-TLB for 2MB pages. Note that this adds no extra TLB entries on top of Intel's Sandy Bridge's split-TLB design.

All configurations used Least-Recently-Used (perfect LRU) replacement policy to select which TLB entry to victimize to make space for a new entry.

5.6.1 Enhancing TLB Reach

The merged-associative TLB increases the TLB reach (number of TLB entries × page size) of a split-TLB design by aggregating TLB resources of its sub-TLBs. Table 5-4 shows the L1-DTLB reach for four TLB configurations and different page sizes.

Table 5-4 lists L1-DTLB reach for different TLB configurations and different page sizes. The TLB reach is calculated by assuming page sizes are not mixed, i.e., if 4KB pages are used then all memory is mapped using 4KB pages and similarly for 2MB and 1GB pages. The split-TLB design has 64 entries for 4KB pages, 32 entries for 2MB pages and 4 entries for 1GB pages. The fully associative design has 64 or 96 entry while the merged-associative TLB has 96 entries. Table 5-4 shows that merged-associative TLB could increase the TLB reach over a split-TLB reach substantially (e.g., 96GB compared to 4GB with 1GB pages). Note that, the TLB reach enabled by the merged-associative TLB is equal to that of a 96-entry fully associative TLB and yet does not make use of fully associative structures. While larger TLB reach can help reduce possibility of TLB misses, the actual reduction in TLB miss rate depends upon workload characteristics.

5.6.2 TLB Performance Unpredictability with Large Pages

As explained in Section 5.2.2, a split-TLB design can incur more TLB misses when using large pages. This could happen due to fewer TLB entries that are generally statically allocated for larger page sizes in a split-TLB design. One of the key motivations behind merged-TLB design is to avoid this unpredictability of TLB performance by aggregating the resources of sub-TLBs of a split-TLB design. Table 5-5 shows the TLB miss rates for Intel's Sandy Bridge's split-TLB design and for the merged-associative TLB (shaded column). As shown in Table 5-5, the workload CG from NAS parallel benchmark suite and the nano-benchmark depicted in Figure 5-1 incur more number of TLB miss with larger page size on Intel's Sandy Bridge machine. For example, when 1GB pages are used instead of 2MB pages the number of TLB

Table 5-5. TLB miss rates per 1K memory accesses for Intel's Split Design and for merged-associative TLB.

	4KB		2MB		1GB	
	Split-TLB	**Merged-associative TLB**	**Split-TLB**	**Merged-associative TLB**	**Split-TLB**	**Merged-associative TLB**
NPB:CG	279.5	282.6	42.1	0	130.7	0
Nano-benchmark (Figure 1)	0	0	487.8	0	490.7	0

misses per 1K memory accesses goes up from 42 to 130 for the workload CG. Similarly, the nano-benchmark incurs nearly 500 TLB misses per 1K memory access with 2MB and 1GB page sizes while incurring no misses when 4KB base page size is used. However, with the simulated merged-TLB architecture these workloads do not show any TLB miss unpredictability. For example, workload CG incurs almost no TLB misses with 2MB and 1GB pages and the same is true across all page sizes for the nano-benchmark. Thus, merged-associative TLB is able to mitigate TLB miss rate unpredictability of a conventional split-TLB design.

5.6.3 Performance benefits of merged TLB

The merged-associative TLB design aggregates TLB resources of sub-TLBs for various page sizes. This helps avoid underutilization of hardware TLB resources that is possible with split-TLB design. For example, 96 PTEs (64 + 32) for 2MB pages could be cached in the merged-TLB configuration, while in a split-TLB design could hold address translation for only up to 32 pages. Table 5-6 lists the MKPA for the D-TLB hierarchy when using only 4KB pages.

Table 5-6. TLB misses per kilo memory access with use of 4KB pages.

	Split-TLB	FA-TLB (64 entry)	FA-TLB (96 entry)	Merged-TLB
graph500	207.9	207.9	207.7	207.9
memcached	4.4	4.36	4.42	4.38
MySQL	4.31	3.88	4.05	3.63
NPB:CG	282.1	281.37	284.29	282.71
NPB:BT	5.77	6.63	5.67	5.50

Table 5-7. TLB misses per kilo memory access with 2MB pages.

	Split-TLB	FA-TLB (64 entry)	FA-TLB (96 entry)	Merged-TLB
graph500	60.13	51.94	39.92	33.52
memcached	3.97	4.05	4.04	4.08
MySQL	2.89	3.73	3.48	3.95
NPB:CG	0.68	0.0017	0.0018	0.035
NPB:BT	2.94	3.13	3.12	3.19

Table 5-8. TLB misses per kilo memory accesses using 1GB pages.

	Split-TLB	FA-TLB (64 entry)	FA-TLB (96 entry)	Merged-TLB
graph500	6.04	0	0	0
memcached	3.57	3.02	2.72	2.76
MySQL	4.21	2.79	2.92	3.23
NPB:CG	109.791	0	0	0
NPB:BT	1.28991	0	0	0

For each workload I run four TLB configurations – split TLB, fully associative TLB with 64 and 96 entries and the proposed merged-associative TLB. Table 5-7 and 5-8 lists the same when 2MB and 1GB pages are used, respectively.

I observe that the merged-associative TLB does not enable any reduction in number of TLB misses over split-TLB design when only 4KB pages are used (Table 5-6). The split-TLB design has 64 entries for 4KB pages in L1-TLB, backed by a larger 512 entry L2-TLB. While the merged-associative TLB design provides up to 96 entries in L1-TLB it does not increase the TLB reach in any substantial way and thus demonstrates no benefit.

Table 5-7 shows the TLB miss rates when 2MB pages are used. The split-TLB design provides 32 L1-TLB entries for 2MB pages (no L2 TLB), while the merged-associative TLB provides up to 96 L1-TLB entries. If a workload's working set for 2MB pages is captured by number of TLB entries between 32 and 96 then the merged-associative TLB provides significant reduction in TLB misses. For example, in case of graph500 the merged-associative TLB incurs only 33 TLB misses per 1K memory access while a split-TLB design would have incurred more than 60 misses. The number of TLB misses is slightly lower than even the fully associative configurations as well. The slight improvement shown over fully associative TLB with 96 entries is most likely due to the LRU replacement policy in the TLB. However, I do not notice any significant benefit from the merged-associative TLB for other workloads. I also note that for 2MB pages NPB:CG has very different TLB miss rates in Table 5-5 and Table 5-7 for the split TLB design. The data in Table 5-5 is collected using hardware performance counter while the

data in Table 5-7 is from the simulator. I was unable to explain this discrepancy, however the simulator provided good estimates of TLB misses for almost all other cases.

When 1GB pages are used the merged-associative TLB significantly reduces TLB miss rates compared to split-TLB design for graph500 and CG (Table 5-8). In a split-TLB design there could be only four entries for 1GB pages in the TLB while a merged-associative TLB can hold up to 96 entries. In general, for 1GB pages merged-TLB is able to nearly eliminate all TLB misses for several workloads.

In summary, I find that the merged-associative TLB fails to provide significant reductions in TLB misses over the conventional split-TLB design. Any significant reduction in TLB miss rates occur only when the number of pages that effectively covers the working set of a program falls within the number of DTLB entries allowed by split-TLB design and that enabled by the merged-TLB design.

5.7 Related Work

Virtual memory has long been an active research area. Past and recent work has demonstrated the importance of TLBs to the overall system performance [5,9,13,11,19].

Support for large pages: Almost all processor architectures including MIPS, Alpha, UltraSPARC, PowerPC, and x86 support large page sizes. To support multiple page sizes these architectures implement either a fully associative TLB (Alpha, Itanium) or a set-associative split-

TLB (x86-64). Talluri et al. was first to discuss the tradeoffs and difficulties of supporting multiple page sizes in hardware [88]. More specifically, they pointed out difficulties in designing set-associative TLBs when multiple page sizes are to be supported.

The work closest to the merged-associative TLB proposal is a skewed-set associative TLB [81] proposed (but not evaluated) by Andre Seznec that extends the idea of skewed-associative caches to TLBs to support multiple page sizes in a single set-associative structure. A skewed-set associative TLB extends a set-associative design where each way (column) is indexed by a different hash function. The hash functions are chosen such that a given virtual address has a unique location in each way depending upon the page size. A skewed-associative TLB enables any location in a set-associative TLB structure to map pages with any possible page size. However, the effective associativity of the skewed-associative TLB, i.e., the number of potential locations for a given virtual address mapped with a given page size is much smaller than the actual number of ways in the TLB structure. In the example presented in the paper, the effective associativity is only 2 against the actual 8-way associative TLB structure. Thus, such a design is likely increase TLB misses due to conflicts. Furthermore, to allow address translations for large contiguous virtual memory regions to be cached in a skewed-associative TLB, the author recommends number of entries in the skewed-associative TLB should be twice of the ratio of maximum to minimum page size. Following this recommendation would need 512K (512*512*2) entries in the TLB for the x86-64's page sizes. Thus, skewed-associative TLB is, at best, more suitable for larger L2 TLBs but not for L1-TLB. In the contrary, merged-associative TLB does not have such constraints and can work well for L1 TLBs.

Efficient TLB mechanisms: Prior efforts improved TLB performance either by increasing the TLB hit rate or reducing/hiding the miss latency. For example, recent proposals increase the effective TLB size through co-operative caching of TLB entries [86] or a larger second-level TLB shared by multiple cores [11]. Prefetching was also proposed to hide the TLB miss latency [6,17,22]. SpecTLB [7] speculatively uses large-page translations while checking for overlapping base-page translations. Zhang et al. proposed an intermediate address space between the virtual and physical addresses, under which physical address translation is only required on a cache miss [103]. Recently, Pham et al. [73] proposed hardware support to exploit naturally occurring contiguity in virtual to physical address mapping to coalesce multiple virtual-to-physical page translations into single TLB entries. While these works focus on reducing TLB misses in general, the proposed merged-associative TLB focuses particularly on the performance unpredictability of a split-TLB design.

Since servicing a TLB miss can incur a high latency cost, several processor designs have incorporated software or hardware PTE caches. For example, UltraSPARC has a software-defined Translation Storage Buffer (TSB) that serves TLB misses faster than walking the page table [66]. Modern x86-64 architectures also use hardware translation caches to reduce memory accesses for page-table walks [6].

There are also proposals that completely eliminate TLBs with a virtual cache hierarchy [45,101], where all cache misses consult a page table. However, these techniques work only for uniprocessors or constrained memory layout (e.g., to avoid address synonyms). Opportunistic

Virtual Caching discussed in Chapter 4, reduces address translation energy, but does not reduce TLB miss rates.

While these techniques make TLBs work better or remove them completely, merged-associative TLB proposes better TLB design in presence of multiple page sizes. Thus, merged-associative TLB is orthogonal to many of the above proposals.

5.8 Conclusion

Use of large pages is most common method to reduce overhead of TLB misses. However, supporting large page sizes complicates hardware TLB design. Commercial processors support multiple page sizes either through fully associative TLB or by having separate set-associative TLBs for each supported page sizes (called split-TLB design). While fully associative TLBs are hard to scale and are often slower than set-associative TLBs, having separate set-associative TLBs for different page sizes leads to potential TLB performance unpredictability and resource underutilization.

A merged-associative TLB proposal overcomes the shortcomings of the split-TLB design while avoiding fully associative TLB. In particular, split-TLB removes possibility of performance degradation with use of large page sizes and in some cases significantly reduces the TLB miss rates through resource aggregation.

6

Summary, Future Work, and Lessons Learned

This chapter summarizes the thesis, mentions a few future research directions, and discusses a few lessons that I learned during the thesis work.

6.1 Summary

Page-based virtual memory (paging) is a crucial piece of memory management in today's computing systems. Notably though, virtual address translation mechanisms' basic formulation remains largely unchanged since the late 1960s when translation lookaside buffers (TLB) were introduced to efficiently cache recently used address translations. However, the purpose, usage and the design constraints of virtual memory have witnessed a sea change in the last decade.

In this thesis, I reevaluated virtual memory management in today's context**.** In particular, I focused on two aspects of address translation mechanism of the virtual memory subsystem -- latency overhead from TLB misses, particularly for big memory workloads and the energy

dissipation due to address translation. I made three high level observations. First, big memory workloads' demand to efficiently address-translate large amounts of data stretches the current TLB mechanism for address translation to new limits. Second, many key features of page-based virtual memory like swapping, fine-grain page protection etc. are less important for many emerging applications that access large amount of memory. Third, the design of the address translation mechanism needs reevaluation considering energy-dissipation as a first class constraint. Based on these observations I proposed three pieces of work. The first piece, called direct segments, aims to reduce latency overhead of address translation. The second piece, called opportunistic virtual caching, reduces energy overheads of address translation. In the third piece, I proposed a merged-associative TLB that improves large page support in commercially prevalent designs by aggregating TLB resources across different page sizes.

I proposed direct segments to reduce the ever-increasing latency overhead of virtual memory's address translation for emerging big memory workloads. Many big memory workloads allocate most of their memory early in execution in large chunks and do not benefit from paging. Direct segments enable hardware-OS mechanisms to dynamically bypass paging for a large part of a process's virtual address space, eliminating nearly 99% of TLB misses for many of these workloads.

I proposed opportunistic virtual caching (OVC) to reduce the energy spent on translating addresses. Accessing TLBs on each memory references uses significant energy, and virtual memory's page size constrains conventional L1 cache designs to be highly associative -- burning yet more energy. OVC makes hardware-OS modifications to expose energy-efficient virtual

caching as a dynamic optimization. This saves 94-99% of TLB lookup energy and 23% of L1 cache lookup energy across several workloads.

While direct segments could eliminate most TLB misses for big memory workloads that often have fairly predictable memory usage pattern, they are less suitable when there are frequent memory allocations/deallocations. Large pages are likely to be more suitable under such dynamic memory management scenarios. Unfortunately, prevalent chip designs like Intel's Ivy Bridge statically partition TLB resources among multiple page sizes (default base page sizes and larger page sizes), which could lead to performance pathologies for using large pages. To this end, I proposed merged-TLB to avoid these performance pathologies and allow dynamic aggregation of TLB resources to reduce TLB miss rates.

A common theme across the techniques proposed in this thesis is to attach semantic information with the virtual addresses that is used by the hardware to enable efficient address translation. For example, in direct segments, a range of virtual address designates memory region that may not benefit from paging. In OVC, the hardware uses virtual caching or physical caching depending upon a higher order bit of the virtual address. In a merged-associative TLB, a hint on the page size for translation is embedded in the virtual address range.

6.2 Future Research Directions

While my thesis research reevaluated virtual memory's address translation mechanisms for latency and energy dissipation, I believe there are at least three additional emerging scenarios in computing that encourages rethinking of virtual memory management. These three computing

trends are: wide use of virtual machines, emergence of non-volatile memory and emergence of single-chip heterogeneous computing.

6.2.1 Virtual Machines and IOMMU

Virtual machines are gaining importance in cloud-era computing as they enable resource consolidation, security, performance isolation. However, under virtualized environments the cost of address translation can be multiple times of that in a native system as the hardware may traverse two levels of address translation [10:-].

This may make future big memory workloads less suitable for running with virtual machines. Further, many emerging workloads are I/O intensive and the IOMMU hardware in modern processors are often used to provide protection against buggy devices and to provide guest operating systems with direct access to devices under virtualization. However, enforcing strict protection through IOMMU often incurs significant overhead to I/O intensive workloads.

In the near term, the direct segment design can be extended to reduce the TLB miss cost in virtual machines by eliminating one or both levels of page-walk. In longer term, I think virtual memory management could be specialized to cater to the needs of operation under virtualized environment. For example, today in x86-64 architecture the same hierarchical page table structures and similar address translation hardware are used for translating addresses for both translation layers under virtualized environment. However, key features like sparsity of address mapping, size of memory mapping can be different among the two address translation layers.

Further, utilizing IOMMU to provide strict protection against buggy devices can incur significant performance cost due to the need of frequent creation/destruction of memory mappings [99]. It may be possible to enable the IOMMU hardware to provide the OS with ephemeral, self-destructing mappings that automatically expires after an OS-specified condition (e.g., access counts, elapsed time). This could avoid costly OS interventions on unmapping operations. Moreover, as use of the IOMMU becomes popular, challenges in efficient virtualization of IOMMU hardware itself -- possibly through two-dimensional IOMMU -- could be interesting. Such a two-dimensional IOMMU could provide protection against device driver bugs in the guest OS, while allowing guest OS to access devices without VMM intervention.

6.2.2 Non-Volatile Memory

As the DRAM technology faces scaling challenges in sub-40 nm process emerging non-volatile memories (NVM) like phase are being touted as potential DRAM replacement. However, unique features of most NVM technologies like non-volatility, read-write asymmetry, and limited write-endurance make a compelling case of revisiting DRAM-era virtual memory design.

For example, while redundant writes (e.g., due to zeroing of zeroed pages) by virtual memory is hardly an issue with DRAM, finding and eliminating these writes may be important due to limited write endurance and high cost of write operation in NVMs. Further, stray writes by buggy applications on the persistent memory can leave inconsistencies that survive restart. One potential solution to contain stray writes may be to treat all persistent user memory as read-only and can be written only after application explicitly requests OS to make a range of addresses writable. However, such a system could require low-overhead page-permission

modifications. Hardware-enforced TLB coherence instead of today's long-latency software enforced one (a.k.a. TLB shootdown) could be helpful in such scenario. NVM's ability to allow both fast, byte-addressable access and non-volatility allows it to be treated as physical memory or storage media. It could be interesting to explore ways to dynamically decide what portion of installed NVM is treated as memory and what portion for persistence. Furthermore, segregation of read-mostly and write-mostly memory regions rather than only identifying read-only, read-execute or read-write regions can be beneficial for future virtual memory that needs to deal with read-write asymmetry and write-endurance of NVMs.

6.2.3 Heterogeneous Computing

There is a growing trend of heterogeneous computing elements (e.g., central processing unit, graphic processing unit, cryptographic unit) being tightly integrated together on a system-on-chip. Extending virtual memory seamlessly across varied computing elements for ease of programming and for better management of heterogeneity is likely to be key for such tight integration. However, different computing elements have very different memory usage needs and simply extending conventional virtual-memory hardware and OS techniques to non-CPU computing units may not be optimal. For example, GPUs tend to demand much more memory bandwidth than CPUs. Further, GPU's lock-step execution model makes its memory-access patterns bursty. Sustaining address translation needs of bursts of concurrent memory accesses may be a real challenge for the hardware virtual memory apparatus. Further, GPU workloads often demonstrate streaming access patterns that may make TLBs less effective due to lack of temporal locality. Moreover, GPU memory architecture is different from that of the conventional

CPU. Unlike CPUs, GPUs enable different types of memory like scratchpad (shared) memory, global memory and includes memory-access coalescer before cache access. Efficiently handling page-faults and enabling demand paging on GPUs may also need further exploration. Exploring the feasibility of virtual caches in GPUs can be another interesting research question as it might lower the latency and energy overhead of address translation in GPUs. In summary, I believe that accommodating memory usage needs of very different computing units while presenting a homogenous virtual address space to programmer is challenging and needs further exploration.

6.3 Lessons Learned

There are few important lessons I learned during my thesis work.

First, the merged-associative TLB work made me realize that I should have done more back-of-the-envelope calculations on potential benefits before delving into implementations. In the hindsight, I observe that merged-associative TLBs can substantially reduce the TLB miss rates over a split-TLB design for a narrow range of applications whose working set fits within the entries enabled by a merged-associative TLB but not by a split-TLB design. Unfortunately, I could have made this observation even before any implementation effort.

Second, during my thesis work, I realized that several inefficiencies in performance and energy dissipation could be eliminated through cross-layer optimizations, as also observed in a recent community whitepaper [21]. In the later part of 20th century and in the early 21st century, the technology scaling provided tremendous impetus to computing capability. Modern computing systems harnessed this capability by evolving into having many layers (e.g., OS,

compiler, architecture), each with often disjoint and well-defined set of responsibilities. This helped divide-and-conquer the design complexity. However, this layering hides much of the semantic information between the layers of computing and results in inefficiencies. These inefficiencies arise from the lost opportunity to optimize an operation by needing to ensure that the desired operations work under *all* possible scenarios. For example, in direct segments work, I found that big memory applications do not benefit from page-based virtual memory for most of its memory allocations and yet systems enforces page-based virtual memory for all memory allocation.

Thus cross layer optimizations can help reduce the inefficiencies and thus enable better computing capability both in terms of performance and energy-efficiency even without same level of technology scaling that were available until early part of 21st century. Consequently, I encourage researchers and engineers to seek out and exploit additional cross-layer optimizations.

.

Bibliography

1. Adams, K. and Agesen, O. A comparison of software and hardware techniques for x86 virtualization. *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, (2006), 2–13.
2. Ahn, J., Jin, S., and Huh, J. Revisiting Hardware-Assisted Page Walks for Virtualized Systems. *Proceedings of the 39th Annual International Symposium on Computer Architecture*, (2012).
3. AMD, *AMD64 Architecture Programmer's Manual Vol. 2*. http://support.amd.com/us/Processor_TechDocs/24593_APM_v2.pdf.
4. ARM, *Technology Preview: The ARMv8 Architecture*. http://www.arm.com/files/downloads/ARMv8_white_paper_v5.pdf.
5. Ashok, R., Chheda, S., and Moritz, C.A. Cool-Mem: combining statically speculative memory accessing with selective address translation for energy efficiency. *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, (2002).
6. Barr, T.W., Cox, A.L., and Rixner, S. Translation caching: skip, don't walk (the page table). *Proceedings of the 37th Annual International Symposium on Computer Architecture*, (2010).
7. Barr, T.W., Cox, A.L., and Rixner, S. SpecTLB: a mechanism for speculative address translation. *Proceedings of the 38th Annual International Symposium on Computer Architecture*, (2011).
8. Basu, A., Gandhi, J., Chang, J., Hill, M.D., and Swift, M.M. Efficient Virtual Memory for Big Memory Servers. *Proc. of the 40th Annual Intnl. Symp. on Computer Architecture*, (2013).
9. Basu, A., Hill, M.D., and Swift, M.M. Reducing Memory Reference Energy With Opportunistic Virtual Caching. *Proceedings of the 39th annual international symposium on Computer architecture*, (2012), 297–308.
10. Bhargava, R., Serebrin, B., Spadini, F., and Manne, S. Accelerating two-dimensional page walks for virtualized systems. *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, (2008).
11. Bhattacharjee, A., Lustig, D., and Martonosi, M. Shared last-level TLBs for chip multiprocessors. *Proc. of the 17th IEEE Symp. on High-Performance Computer Architecture*, (2011).
12. Bhattacharjee, A. and Martonosi, M. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, (2009).
13. Bhattacharjee, A. and Martonosi, M. Inter-core cooperative TLB for chip multiprocessors. *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, (2010).
14. Binkert, N., Beckmann, B., Black, G., et al. The gem5 simulator. *Computer Architecture News (CAN)*, (2011).
15. Cekleov, M. and Dubois, M. Virtual-Address Caches Part 1: Problems and Solutions in Uniprocessors. *IEEE Micro 17*, 5 (1997).

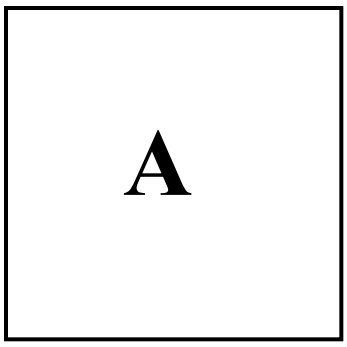
16. Cekleov, M. and Dubois, M. Virtual-Address Caches, Part 2: Multiprocessor Issues. *IEEE Micro 17*, 6 (1997).
17. Chang, Y.-J. and Lan, M.-F. Two new techniques integrated for energy-efficient TLB design. *IEEE Trans. Very Large Scale Integr. System 15*, 1 (2007).
18. Chase, J.S., Levy, H.M., Lazowska, E.D., and Baker-Harvey, M. Lightweight shared objects in a 64-bit operating system. *OOPSLA '92: Object-oriented programming systems, languages, and applications*, (1992).
19. Chen, J.B., Borg, A., and Jouppi, N.P. A Simulation Based Study of TLB Performance. *Proceedings of the 19th Annual International Symposium on Computer Architecture*, (1992).
20. Christos Kozyrakis, A.K. and Vaid, K. Server Engineering Insights for Large-Scale Online Services. *IEEE Micro*, (2010).
21. Computer Architecture Community, *21st Century Computer Architecture*. 2012. http://cra.org/ccc/docs/init/21stcenturyarchitecturewhitepaper.pdf
22. Consortium, I.S. *Berkeley Internet Name Domain (BIND)*. http://www.isc.org .
23. Corbet, J. Transparent huge pages. 2011. www.lwn.net/Articles/423584/.
24. Daley, R.C. and Dennis, J.B. Virtual memory, processes, and sharing in MULTICS. *Communications of the ACM 11*, 5 (1968), 306–312.
25. Denning, P.J. The working set model of program behavior. *Communications of ACM 11*, 5 (1968), 323–333.
26. Denning, P.J. Virtual Memory. *ACM Computing Surveys 2*, 3 (1970), 153–189.
27. Diefendorff, K., Oehler, R., and Hochsprung, and R. Evolution of the PowerPC Architecture. *IEEE Micro 14*, 2 (1994).
28. Ekman, M., Dahlgren, F., and Stenstrom, P. TLB and Snoop Energy-Reduction using Virtual Caches in Low-Power Chip-Multiprocessors. *In Proceedings of International Symposium on Low Power Electronics and Design*, (2002), 243–246.
29. Emer, J.S. and Clark, D.W. A Characterization of Processor Performance in the vax-11/780. *Proceedings of the 11th Annual International Symposium on Computer Architecture*, (1984), 301–310.
30. Eric J. Koldinger, J.S.C. and Eggers, S.J. Architecture support for single address space operating systems. *ASPLOS '92: 5th international conference on Architectural support for programming languages and operating systems*, (1992).
31. Ferdman, M., Adileh, A., Kocberber, O., et al. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. *Proceedings of the 17th Conference on Architectural Support for Programming Languages and Operating Systems*, ACM (2012).
32. Ganapathy, N. and Schimmel, C. General purpose operating system support for multiple page sizes. *Proceedings of the annual conference on USENIX Annual Technical Conference*, (1998).
33. Ghemawat, S. and Menage, P. TCMalloc : Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.
34. Goodman, J.R. Coherency for multiprocessor virtual address caches. *ASPLOS '87: Proceedings of the 2nd international conference on Architectual support for programming languages and operating systems*, ACM (1987), 264–268.
35. Gorman, M. Huge Pages/libhugetlbfs. 2010. http://lwn.net/Articles/374424/.

36.graph500 --The Graph500 List. http://www.graph500.org/.
37.Hwang, A.A., Ioan A. Stefanovici, and Schroeder, B. Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design. *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, (2012), 111–122.
38.*Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, Part1, Chapter 2*. 2009.
39.Intel, *Intel 64 and IA-32 Architectures Optimization Reference Manual, Chapter 2*. 2012.
40.Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation.
41.Intel, *TLBs, Paging-Structure Caches, and Their Invalidation*. .
42. Itjungle. *Database Revenues on Rise*. http://www.itjungle.com/tfh/tfh072511-story09.html.
43.J. H. Lee, C.W. and Kim, S.D. Selective block buffering TLB system for embedded processors. *IEE Proc. Comput. Dig. Techniques 152*, 4 (2002).
44.Jacob, B. and Mudge, T. Virtual Memory in Contemporary Microprocessors. *IEEE Micro 18*, 4 (1998).
45.Jacob, B. and Mudge, T. Uniprocessor Virtual Memory without TLBs. *IEEE Transaction on Computer 50*, 5 (2001).
46.Juan, T., Lang, T., and Navarro, J.J. Reducing TLB power requirements. *ISLPED '97: Proceedings of the international symposium on Low power electronics and design*, ACM (1997).
47.Kadayif, I., Nath, P., Kandemir, M., and Sivasubramaniam, A. Reducing Data TLB Power via Compiler-Directed Address Generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 26*, 2 (2007).
48.Kadayif, I., Sivasubramaniam, A., Kandemir, M., Kandiraju, G., and Chen, G. Generating physical addresses directly for saving instruction TLB energy. *MICRO '02: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, ACM (2002).
49.Kandiraju, G.B. and Sivasubramaniam, A. Going the distance for TLB prefetching: an application-driven study. *Proceedings of the 29th Annual International Symposium on Computer Architecture*, (2002).
50.Killburn, T., Edwards, D.B.., Lanigan, M.J., and Sumner, F.H. One-Level Storage System. *IRE Transaction, EC-11 2*, 11 (1962).
51.Kim, J., Min, S.L., Jeon, S., Ahn, B., Jeong, D.-K., and Kim, C.S. U-cache: a cost-effective solution to synonym problem. *HPCA '95: Fisrt IEEE symposyum on High-Performance Computer Architecture*, (1995).
52.Larus, G.H.J., Abadi, M., Aiken, M., et al. *An Overview of the Singularity Project*. Microsoft Research, 2005.
53.Lee, H.-H.S. and Ballapuram, C.S. Energy efficient D-TLB and data cache using semantic-aware multilateral partitioning. *Proceedings of the international symposium on Low power electronics and design*, (2003).
54.Linden, G. Marissa Mayer at Web 2.0. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html.
55.Linux pmap utility. http://linux.die.net/man/1/pmap.
56.Linux. Memory Hotplug. http://www.kernel.org/doc/Documentation/memory-hotplug.txt.

57. Luk, C.-K., Cohn, R., Muth, R., et al. Pin: building customized program analysis tools with dynamic instrumentation. *PLDI'05: ACM SIGPLAN conference on Programming language design and implementation*, ACM (2005).
58. Luk, C.-K., Cohn, R., Muth, R., et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *Proceedings of the SIGPLAN 2005 Conference on Programming Language Design and Implementation*, (2005), 190–200.
59. Lustig, D., Bhattacharje, A., and Martonosi, M. TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs. *ACM Transactions on Architecture and Code Optimization*, (2013).
60. Lynch, W.L. *The Interaction of Virtual Memory and Cache Memory*. Stanford University, 1993.
61. Manne, S., Klauser, A., Grunwald, D., and Somenzi, F. *"Low power TLB design for high performance microprocessors*. University of Colorado, Boulder, 1997.
62. Mars, J., Tang, L., Hundt, R., Skadron, K., and Soffa, M.L. Bubble-Up: increasing utilization in modern warehouse scale computers via sensible co-locations. *Proceedings of the 44th Annual IEEE/ACM International Symp. on Microarchitecture*, (2011).
63. McCurdy, C., Cox, A.L., and Vetter, J. Investigating the TLB Behavior of High-end Scientific Applications on Commodity Microprocessors. *Proceedings of IEEE International Symposium on Performance Analysis of Systems and software*, IEEE (2008).
64. McNairy, C. and Soltis, D. Itanium 2 Processor Microarchitecture. *IEEE Micro 23*, 2 (2003), 44–55.
65. memcached - a distributed memory object caching system. www.memcached.org.
66. Microsystems, S. UltraSPARC T2^{TM} Supplement to the UltraSPARC Architecture 2007. (2007).
67. Mozilla, M. *Firefox, web browser*. http://www.mozilla.org/en-US/firefox/new/.
68. Muralimanohar, N., Balasubramonian, R., and Jouppi, N.P. *CACTI 6.0*. Hewlett Packard Labs, 2009.
69. Navarro, J., Iyer, S., Druschel, P., and Cox, A. Practical, transparent operating system support for superpages. *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, ACM (2002).
70. Navarro, J., Iyer, S., Druschel, P., and Cox, A. Practical Transparent Operating System Support for Superpages. *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, (2002).
71. Ousterhout, J. and al, et. The case for RAMCloud. *Communications of the ACM 54*, 7 (2011), 121–130.
72. Patterson, D.A. and Hennessy, J.L. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2005.
73. Pham, B., Vaidyanathan, V., Jaleel, A., and Bhattacharjee, A. CoLT: Coalesced Large Reach TLBs. *Proceedings of 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM (2012).
74. Princeton, P. *Princeton Application Repository for Shared-Memory Computers (PARSEC)*. http://parsec.cs.princeton.edu/.

75. Puttaswamy, K. and Loh, G.H. Thermal analysis of a 3D die-stacked high-performance microprocessor. *GLSVLSI '06: 16th ACM Great Lakes symposium on VLSI*, ACM (2006).
76. Qiu, X. and Dubois, M. The Synonym Lookaside Buffer: A Solution to the Synonym Problem in Virtual Caches. *IEEE Trans. on Computers 57*, 12 (2008).
77. Ranganathan, P. From Microprocessors to Nanostores: Rethinking Data-Centric Systems. *Computer 44*, 1 (2011).
78. Reiss, C., Tumanov, A., Ganger, G.R., Katz, R.H., and Kozuch, M.A. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. *Proceedings of the 3rd ACM Symposium on Cloud Computing*, ACM (2012).
79. Rosenblum, N.E., Cooksey, G., and Miller, B.P. Virtual machine-provided context sensitive page mappings. *Proceedings of the 4th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, (2008).
80. Saulsbury, A., Dahlgren, F., and Stenstrom, P. Recency-based TLB preloading. *Proceedings of the 27th Annual International Symposium on Computer Architecture*, (2000).
81. Seznec, A. Concurrent Support fo Multiple Page Sizes on a Skewed Associative TLB. *IEEE Transactions on Computers 53(7)*, (2004), 924–927.
82. Sinharoy, B. IBM POWER7 multicore server processor. *IBM Journal for Research and Development 55*, 3 (2011).
83. Sodani, A. *Race to Exascale: Opportunities and Challenges*. MICRO 2011 Keynote.
84. Sourceforge.net, S. ne. Oprofile. http://oprofile.sourceforge.net/.
85. *SpecJBB 2005*. http://www.spec.org/jbb2005/.
86. Srikantaiah, S. and Kandemir, M. Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors. *Proceedings of 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, (2010).
87. Talluri, M. and Hill, M.D. Surpassing the TLB performance of superpages with less operating system support. *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, (1994).
88. Talluri, M., Kong, S., Hill, M.D., and Patterson, D.A. Tradeoffs in Supporting Two Page Sizes. *Proceedings of the 19th Annual International Symposium on Computer Architecture*, (1992).
89. Tang, D., Carruthers, P., Totari, Z., and Shapiro, M.W. Assessment of the Effect of Memory Page Retirement on System RAS Against Hardware Faults. *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, (2006), 365–370.
90. Vmware Inc. Large Page Performance: ESX Server 3.5 and ESX Server 3i v3.5. http://www.vmware.com/files/pdf/large_pg_performance.pdf.
91. Volos, H., Tack, A.J., and Swift, M.M. Mnemosyne: Lightweight Persistent Memory. *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, (2011).
92. Waldspurger, C.A. Memory Resource Management in VMware ESX Server. *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation*, (2002).
93. Wang, W.H., Baer, J.-L., and Levy, and H.M. Organization and performance of a two-level virtual-real cache hierarchy. *ISCA '89: Proceedings of the 16th annual international symposium on Computer architecture*, ACM (1989).

94. Wiggins, A. and Heiser, G. Fast Address-Space Switching on the StrongARM SA-1100 Processor. *Proc. of the 5th Australasian Computer Architecture Conference*, (1999).
95. Wikipedia, W. *Virtual Memory*. http://en.wikipedia.org/wiki/Virtual_memory .
96. Wikipedia, W. *Burrough Large Systems*. http://en.wikipedia.org/wiki/Burroughs_large_systems.
97. Wikipedia, W. Intel 8086. http://en.wikipedia.org/wiki/Intel_8086.
98. Wikipedia, W. *CPU caches*. http://en.wikipedia.org/wiki/CPU_cache.
99. Willmann, P., Rixner, S., and Cox, A.L. Protection strategies for direct access to virtualized I/O devices. *USENIX '08: Proceedings of the USENIX Annual Technical Conference*, (2008).
100. Woo, D.H., Ghosh, M., Özer, E., Biles, S., and Lee, H.-H.S. Reducing energy of virtual cache synonym lookup using bloom filters. *In Proceedings of the international conference on Compilers, architecture and synthesis for embedded systems (CASES)*, (2006), 179–189.
101. Wood, D.A., Eggers, S.J., Gibson, G., Hill, M.D., and Pendleton, J.M. An in-cache address translation mechanism. *Proceedings of 13th annual international symposium on Computer architecture*, (1986).
102. Yang, H., Breslow, A., Mars, J., and Tang, L. Bubble-flux: precise online QoS management for increased utilization in warehouse scale computers. *Proc. of the 40th Annual Intnl. Symp. on Computer Architecture*, (2013).
103. Zhang, L., Speight, E., Rajamony, R., and Lin, J. Enigma: architectural and operating system support for reducing the impact of address translation. *Proceedings of the 24th ACM International Conference on Supercomputing*, ACM (2010).
104. Zhou, X. and Petrov, P. Heterogeneously tagged caches for low-power embedded systems with virtual memory support. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 13*, 2 (2008).



Appendix . Raw Data Numbers

In the appendix, I put some raw numbers for various experiments conducted for Direct Segment (Chapter 3) and Opportunistic Virtual Caching (Chapter 4). The data presented in Chapter 5 are all absolute and so, I add no more data for the Chapter 5.

Raw data for Direct Segment (Chapter 3)

In Chapter 3 there are primarily two sets of relative numbers. The first set provides the percentage of execution cycles attributed to data-TLB misses (Table 3-4). The second set of data captures the fraction of data-TLB misses that falls in direct segment memory (Table 3-5).

I collected the above-mentioned first set of data using performance counters mentioned in Chapter 3. The performance counter numbers are sampled after the initialization phase of each workload and sampling is continued till the TLB-miss cycles as the fraction of total execution cycles shows no further significant change. Thus, multiple runs of each workload did not necessarily run for same amount of logical unit of work. The raw numbers presented here are also has one more caveat. All though the relative numbers calculated from the raw data presented below is close to all relative numbers presented in Chapter 3, they may not be exactly match.

This happened since the archived profiled raw performance counter values did not exactly correspond to the sampling done for the data but correspond to when the execution of workload ended. Table A-1 presents raw numbers related to Table 3-4. It provides the execution cycles and the DTLB miss cycles.

Table A-1. Execution Cycles.

	4KB		2MB		1GB	
	Total cycles (in million)	DTLB miss cycles (in million)	Total cycles (in million)	DTLB miss cycles (in million)	Total cycles (in million)	DTLB miss cycles (in million)
graph500	10031600	5128000	6114000	602400	5086400	74400
memcached	2540400	238400	2136400	129200	4485200	172000
mySQL	1388800	69200	1665200	72000	1654800	62000
NPB:BT	1229600	685600	18807600	223600	13987600	77600
NPB:CG	20365600	6161600	69655000	1176950	13354800	952000
GUPS	143200	118800	143200	76400	143600	26000

In Table A-2, I provide the total number of DTLB misses and the number that falls within the direct segment memory.

Table A-2. DTLB miss counts.

	DTLB miss in DS	Total DTLB miss
graph500	9090742807	9091425377
memcached	1002012524	1002103147
mySQL	453105523	501628801
NPB:BT	1262602735	1263148127
NPB:CG	5079907025	5080166016

Raw data for Opportunistic Virtual Caching (Chapter 4)

Table B-1 provides the raw energy numbers L1 TLBs. It presents data and instruction TLB's dynamic access energy for the baseline and the opportunistic virtual caching. These numbers corresponds to numbers Table 4-7 in Chapter 4.

Table B-1. TLB energy numbers.

	L1 D TLB Dynamic Energy (nJ)		L1 ITLB Dynamic Energy (nJ)	
	Baseline	**OVC**	**Baseline**	**OVC**
canneal	10171063.1412	2804367.79579	32533227.2304	6929.51215291
facesim	15448572.3787	496105.789414	55033317.1814	6218.76220788
fluidanimate	11911983.2395	75812.2024277	55691262.1606	501.209294257
streamcluster	15820949.7295	777460.978147	50381618.6952	3123.37863463
swaptions	12038730.3317	117748.159708	44604476.28	4729.09656129
x264	12304883.1953	541890.521626	42053075.9906	292748.61594
bind	10060588.2406	300806.244694	27134186.3582	459383.718418
specjbb	13034437.0135	1032790.11189	44576334.9065	358372.779693
memcached	14694311.4458	795933.941261	45221072.5231	639722.305095

Table B-2 provides the raw energy numbers L1 caches. It presents data and instruction cache's dynamic access energy for the baseline and the opportunistic virtual caching. These numbers corresponds to numbers Table 4-10 in Chapter 4.

Table B-2. L1 cache energy numbers.

	L1 D-Cache Dynamic Energy (nJ)		L1 I-Cache Dynamic Energy (nJ)	
	Baseline	**OVC**	**Baseline**	**OVC**
canneal	75157238.0322	61352743.0943	213472195.203	162719484.85
facesim	105425385.611	80917926.0771	361040885.469	275195461.912
fluidanimate	78543497.5052	59921791.1285	365328236.349	278452413.445
streamcluster	108847366.129	84011005.4784	330523189.686	251911011.881
swaptions	78993953.6996	60228798.5231	292598096.757	223070905.714
x264	82154652.1382	58894228.6761	275970385.801	210461550.156
bind	66964562.73	51061748.8156	178903587.764	136426060.701
specjbb	89866798.4328	68197386.7696	293745553.024	224107587.044
memcached	101121133.98	77922454.2875	297146477.955	229334179.882

Table B-3 provides the execution cycles spent for the baseline and OVC configurations. These raw numbers corresponds to Table 4-11.

Table B-3. Execution cycles (in millions).

	Baseline	OVC
canneal	19424405157	19412318708
facesim	4037494795.34256	8162729465
fluidanimate	8164085256.37547	4036934313
streamcluster	9498754713.03902	9498999061
swaptions	3060185160.00724	3062848436
x264	4335044991.28174	4339342731
bind	2282773185.46217	2297728023
memcached	14998123261.9223	15014787106
specjbb	6110624382.00161	6084041979