Efficient Virtual Memory for Big Memory Servers

Arkaprava Basu[1] Jayneel Gandhi[1] Jichuan Chang[2] Mark D. Hill[1] Michael M. Swift[1]

[1]University of Wisconsin-Madison
Madison, WI, USA
{basu, jayneel, markhill, swift}@cs.wisc.edu

[2] Hewlett-Packard Laboratories
Palo Alto, CA, USA
jichuan.chang@hp.com

ABSTRACT

Our analysis shows that many "big-memory" server workloads, such as databases, in-memory caches, and graph analytics, pay a high cost for page-based virtual memory. They consume as much as 10% of execution cycles on TLB misses, even using large pages. On the other hand, we find that these workloads use read-write permission on most pages, are provisioned not to swap, and rarely benefit from the full flexibility of page-based virtual memory.

To remove the TLB miss overhead for big-memory workloads, we propose mapping part of a process's linear virtual address space with a *direct segment*, while page mapping the rest of the virtual address space. Direct segments use minimal hardware—base, limit and offset registers per core—to map contiguous virtual memory regions directly to contiguous physical memory. They eliminate the possibility of TLB misses for key data structures such as database buffer pools and in-memory key-value stores. Memory mapped by a direct segment may be converted back to paging when needed.

We prototype direct-segment software support for x86-64 in Linux and emulate direct-segment hardware. For our workloads, direct segments eliminate almost all TLB misses and reduce the execution time wasted on TLB misses to less than 0.5%.

Categories and Subject Descriptors:

B.3.2 [**Virtual Memory**] OS-Hardware Virtual Memory management.

General Terms:

Design, Performance.

Key Words:

Virtual Memory, Tanslation Lookaside Buffer.

1. INTRODUCTION

"Virtual memory was invented in a 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. Notably, its basic formulation remains largely unchanged since the late 1960s when translation-lookaside buffers (TLBs) were introduced [13]. In contrast, virtual memory usage has changed dramatically in recent years. For example, historically, a key motivation behind page-based virtual memory was to virtualize and overcommit *scarce* physical memory without programmer intervention. Today, the availability of 64-bit addressing and the decline in memory price have led to servers with tens of gigabytes or even terabytes of physical memory: HP's DL980 currently ships with up to 4TB physical memory, and Windows Server 2012 supports 4TB memories, up from 64GB a decade before.

However, page-based virtual memory as of today is *far from a free lunch*. The primary cost comes 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 100s of cycles, and frequent TLB lookups cost non-negligible energy [5, 41]

To reevaluate the cost and benefit of decades-old page-based virtual memory in today's context, we 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.

Our 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* [19] 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 [17, 35]—contributes to high TLB miss rates and consequently we expect even higher address translation overheads in future. Moreover, the trends to larger physical memory sizes and byte-addressable access to storage class memory [37, 47] increase address mapping pressure.

Despite these costs, we 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. We find that only a small fraction of memory uses per-page protection for mapping files, for executable code. Nevertheless, current

ISCA'13 Tel-Aviv, Israel

Table 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)

Table 2. Workload Description

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., 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

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, we 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 same 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 can 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 paper are:

- We 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.
- We propose *direct-segment* hardware for efficient mapping of application-specified large *primary regions*, while retaining full compatibility with standard paging.
- We demonstrate the correctness, ease-of-use and performance/efficiency benefits of our proposal.

2. BIG-MEMORY WORKLOAD ANALYSIS

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

1. *Use of virtual memory*: we study what virtual memory functionalities are used by such big-memory workloads;
2. *Cost of virtual memory*: we measure the overhead of TLB misses with conventional page-based virtual memory;
3. *Execution environment*: we study common characteristics of the execution environment of big-memory workloads.

Table 1 describes the test machine for our experiments.

Table 2 describes the workloads used in our study. These applications represent important classes of emerging workloads, ranging from in-memory key-value stores (*memcached*), web-scale databases (*MySQL*), graph analytics (*graph500*) and supercomputing (*NAS* parallel benchmark suite). Further, we also studied the *GUPS* micro-benchmark designed by HPC community to stress-test random memory access in high-performance computing settings.

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 [15]. 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.

We 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 [28]. This drives large websites such as Facebook, Google, Microsoft Bing, and Twitter to keep their user-facing data (e.g., search indices) in memory [12]. 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. We examine this hypothesis by measuring the amount of swapping in these workloads with the *vmstat* Linux utility. As expected, we observe *no swapping activity*, indicating little value in providing the capability to swap.

Memory allocation and fragmentation. Frequent allocation and de-allocation of different size memory chunks can leave holes in physical memory that prevent subsequent memory allocations,

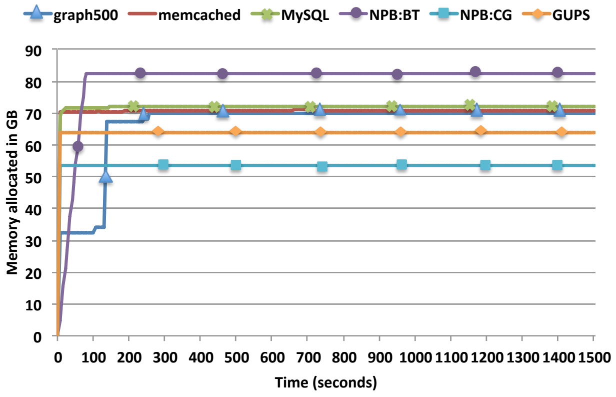

Figure 1. Memory allocated over time by workloads.

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.

We corroborate this behavior by tracking the amounts of memory allocated to a workload over its runtime. We use Linux's *pmap* utility [26] to periodically collect total allocated memory size for a given process. Figure 1 shows the allocated memory sizes in 5-second intervals over 25 minutes of execution for each workload on our test machine described in Table 1.

Across our workloads, we see 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 [38]. Furthermore, because memory allocation stabilizes after startup, these workloads have *predictable* memory usage.

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, we examine the kernel metadata for memory allocated to a given process. Specifically, we focus on one type of commonly used memory regions—*anonymous* regions (not backed by file) (excluding stack regions). Table 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).

We 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, our targeted big-memory workloads do not require these features for *most* of the memory they allocate.

Table 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%

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.

2.2 Cost of Virtual Memory

Here we 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 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, we use hardware performance counters to measure the processor cycles spent by the hardware page-table walker in servicing TLB misses. 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. Our estimate for TLB-miss latency is conservative as we do not account for L1 TLB misses that hit in L2 TLB (for 4KB pages), which can take around 7 cycles [27]. We run the experiments with base (4KB), large (2MB) and huge page (1GB).

We report TLB miss latency as a fraction of total execution cycles to estimate its impact on the execution time. Table 4 lists our findings (the micro-benchmark GUPS is separated at the bottom). First, we observe that TLB misses on data accesses (D-TLB miss-

Table 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

es), 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 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 [25].

In summary, across most workloads (*graph500, memcached, MySQL*) we observe substantial overhead for servicing TLB misses on our 96 GB 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 [41], 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.

2.3 Application Execution Environment

Finally, we 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, database). 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 [29], low latency and performance predictability. A recent study by Reiss et al. [38] 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.

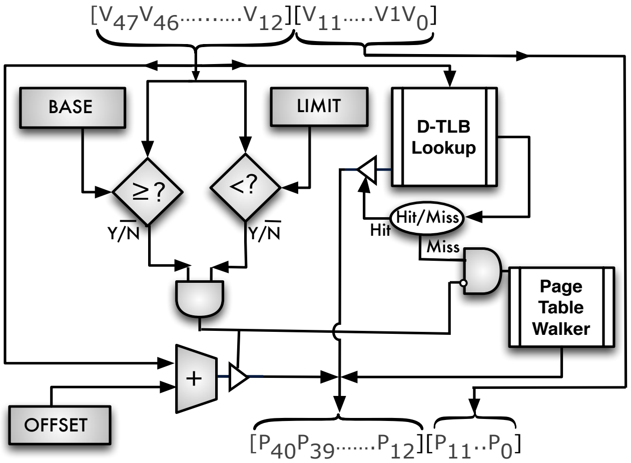
$[V_{47}V_{46}\ldots\ldots V_{12}][V_{11}\ldots V1V_0]$
BASE
LIMIT
D-TLB Lookup
≥?
<?
Y/$\overline{N}$
Y/$\overline{N}$
Hit/Miss
Hit
Miss
Page Table Walker
+
OFFSET
$[P_{40}P_{39}\ldots\ldots P_{12}][P_{11}..P_0]$

Figure 2. Logical view of address translation with direct segment. Added hardware is shaded.

> *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. MORE EFFICIENT VIRTUAL MEMORY

Inspired by the observations in Section 2, we 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, we 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, we propose *direct-segment* hardware (Section 3.1) that is used via a software *primary region* (Section 3.2).

3.1 Hardware Support: Direct Segment

Our 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, we translate 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 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 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,
- 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 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. We describe how the OS calculates and handles the value of these registers in the next section.

Unlike some prior segment-based translation mechanisms [14, 21] 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.

3.2 Software Support: Primary Region

System software has two basic responsibilities in our 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.

3.2.1 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 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.

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, 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.

3.2.2 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

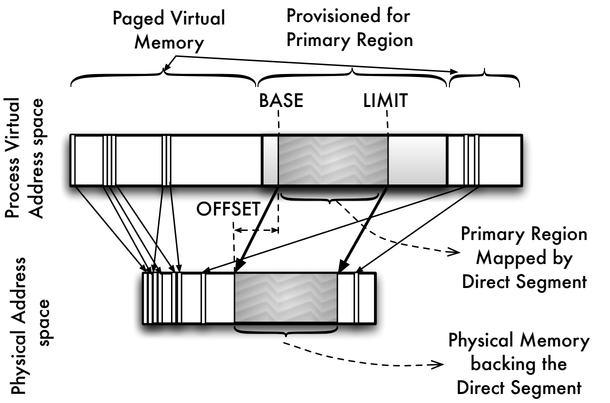
Paged Virtual Memory
Provisioned for Primary Region
BASE
LIMIT
Process Virtual Address space
OFFSET
Primary Region Mapped by Direct Segment
Physical Address space
Physical Memory backing the Direct Segment

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

through periodic memory compaction, similar to Linux's Transparent Huge Pages support [46]. The cost of memory compaction can be amortized over the execution time of long-running processes. We measured that it takes around 3 seconds to crate 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 [38].

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 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.

3.3 Discussion

In this section, we 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, our 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 we describe major shortcoming of large pages that we overcome in our proposal.

First, large pages and their TLB support do not automatically scale to 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) [35, 37]. 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 [43]. 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 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 paper. We also note that, similar to large pages use of direct segment may reduce opportunities for deduplication[48].

Direct segments for kernel memory: So far, we 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, we empirically measured that often nearly 99% of kernel TLB misses reference direct-mapped addresses and thus can be eliminated by a direct segment.

Not general (enough): Direct segments are not a fully general solution to TLB performance. We 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.

Limitations: While our approach simultaneously achieves address translation efficiency and compatibility, it should not be misused in environments with mismatching characteristics. In general, our proposed technique is less suitable for dynamic execution environments where many processes with unpredictable memory usage execute for short periods. However, we 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.

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. Our experiments use Google's *tcmalloc()* [45], which does not suffer from this idiosyncrasy.

4. SOFTWARE PROTOTYPE

We implement our prototype by modifying Linux kernel 2.6.32 (x86-64). Our 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.

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, our prototype uses explicit identification of the primary process and physical memory reservations, although more automatic implementations are possible as detailed in Section 3.2. Below we describe the main aspects of our implementation—identifying the process using a direct segment, managing virtual address and managing physical memory.

Identifying the primary process: We implemented a new system call 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 address range for a primary region in the process's virtual address space that is the size of the physical memory in the system. This guarantees the availability of a contiguous virtual address range for memory allocations in the primary region.

Our 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: Our 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 its primary region. We then reserve a contiguous region of physical memory of the given size using Linux's *memory hotplug* utility [31], which takes a contiguous physical memory region out of the kernel's control (relocating data in the region if needed). During startup of a primary process, the kernel maps this physical memory region into the reserved virtual memory region described above. Our current prototype does not support dynamic resizing primary regions or direct segments.

4.2 Architecture-Dependent Implementation

The direct segment design described in Section 3.1 requires new hardware support. To evaluate primary region with direct segment capability on real hardware without building new hardware, we emulate its functionality using 4KB pages. Thus, we built an architecture-dependent implementation of direct segments.

The architecture-dependent portion of our 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 3.2. 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, we emulate direct-segment functionalities using 4KB pages. More specifically, we 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 our 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 5.1 describes the details of our performance evaluation methodology.

5. EVALUATION

In this section we describe our 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, we devise an approach that uses kernel modification and hardware performance counters to estimate the number of TLB misses avoided by direct segments.

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 [10] 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 we want to measure.

We address this challenge using a combination of hardware performance counters and kernel modifications that together enable performance estimation. With this approach, we can run real workloads directly on real hardware. We first use hardware performance counters to measure the performance loss due to hardware page-table walks triggered by TLB misses. We 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. We conservatively assume that TLB miss rate reduction directly correlates to the reduction in time spent on TLB misses, although our design can improve the TLB performance for addresses outside direct segments by freeing TLB and page-walk-cache resources.

1. Baseline: We use hardware performance counters to estimate the fraction of execution cycles spent on TLB misses. We collect data with *oprofile* [34] by running each of the workloads for several minutes on the test platform described in Table 1.

2. Primary Region/Direct Segment: To estimate the efficacy of our proposed scheme, we determine what fraction of the TLB misses would fall in the direct segment. To achieve this, we need 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. We therefore tweaked 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 [39]. We 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.

Box 1. TLB tracking method

We track 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 our test platform) that cause a trap when loaded into the TLB. By setting a reserved bit, we can ensure 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: We use 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, we make the PTE valid and then take two additional actions. First, we force the TLB to load the correct PTE by touching the page with the faulting address. This puts the correct PTE into the TLB. Second, we *poison* 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), we record whether the address falls in the primary region mapped using direct segment. We then perform the two actions above to reload the PTE into the TLB and re-poison the PTE in memory.

We 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. [36]. More specifically, we find the fraction of total TLB misses that fall in the primary region mapped with a direct segment using the methodology described above. We also measure the fraction of execution cycles spent by hardware page walker on TLB misses in the baseline system using performance counters. Finally, we 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.

This estimation makes the simplifying assumption that 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 [27]. 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, we omit these potential benefits in our evaluation.

5.2 Results

In this section we discuss two aspects of performance:

1. What is the performance gain from primary region/direct segments?

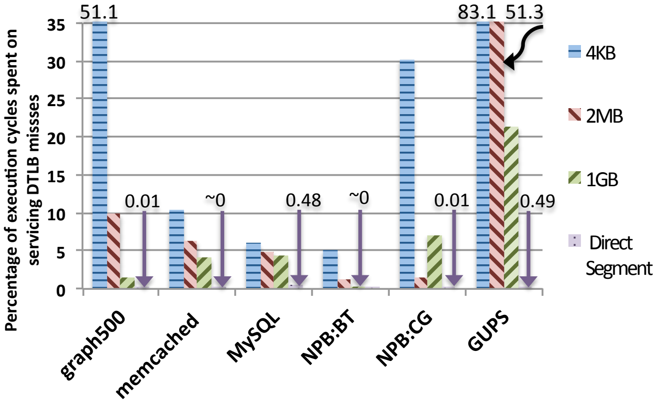

Figure 4. Percentage of cycles spent on DTLB misses. Values larger than 35% are shown above the bars, while very small values shown by straight arrows.

2. How does the primary region/direct-segment approach scale with increasing memory footprint?

Performance gain: Figure 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 we 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, we find that even with larger page sizes significant execution cycles can be spent on servicing TLB misses.

With primary regions and direct segments, our 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 5, as we observe 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. We 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.

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

Table 5. Reduction in TLB misses

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

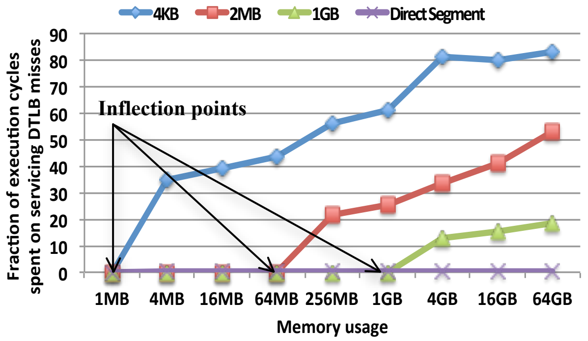

Figure 5. DTLB miss overheads when scaling up GUPS.

three x84-64 page sizes and direct segments. We evaluate GUPS, whose dataset size can be easily configured with a scale parameter. We 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, we 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 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.

6. 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]. We 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 [42] or a larger second-level TLB shared by multiple cores [7]. Prefetching was also proposed to hide the TLB miss latency [6,17,22]. SpecTLB [3] 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 [50]. Recently, Pham et al. [36] 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 [32]. Modern x86-64 architectures also use hardware translation caches to reduce memory accesses for page-table walks [4].

There are also proposals that completely eliminate TLBs with a virtual cache hierarchy [22, 49], 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, we 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 [43]. However, system software has been slow to support the full range of page sizes: operating system support for multiple pages sizes can be complicated [18, 44] and generally follows two patterns. First, applications can explicitly request large pages either through use of libraries like libHugeTLBFS [20] or through special *mmap* calls (in Linux). Second, the OS can automatically use large pages when beneficial [33, 44, 46].

Although useful, we believe large pages are a non-scalable solution for very large memories as discussed in detail in Section 3.3. 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 [6]. 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 [6], and recent work showed that the VMM page table could be flat rather than hierarchical [2]. As mentioned in Section 3.3, we expect our proposed design 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 [21], or more commonly on top of paging as in MULTICS [14], PowerPC, and IA-32 [23]. 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, we 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.

7. CONCLUSION

We find that many big-memory server workloads suffer from high TLB misses (consuming up to 51% of execution cycles) but rarely use swapping or fine-grained page protection, paying the cost of page-based virtual memory without exploiting its full benefits. We also find that most memory accesses in each of these workloads is to a large anonymous region allocated considering available physical memory. We eliminate almost all TLB misses to this region with a primary region software abstraction supported in hardware with a direct segment while other virtual addresses use conventional page-based virtual memory for compatibility. With a Linux 2.6.32 prototype and hardware approximation, we show that our proposal eliminates almost all TLB performance loss in these big-memory workloads.

8. ACKNOWLEDGEMENT

We thank Sankaralingam Panneerselvam and Haris Volos for their help during the project. We thank Dan Gibson, Benjamin Serebin and David Wood for their thoughtful comments. We thank Wisconsin Computer Architecture Affiliates for their feedback on an early version of the work. We thank Richardson Addai-Mununkum for proof-reading our drafts.

This work is supported in part by the National Science Foundation (CNS-0720565, CNS-0916725, and CNS-1117280, CNS-0834473), Google, and the University of Wisconsin (Kellett award and Named professorship to Hill). The views expressed herein are not necessarily those of any sponsor. Hill has a significant financial interest in AMD, and Swift has a significant financial interest in Microsoft.

9. REFERENCES

[1] Adams, K. and Agesen, O. 2006. 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* (Oct. 2006), 2–13.

[2] Ahn, J. et al. 2012. Revisiting Hardware-Assisted Page Walks for Virtualized Systems. *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Jun. 2012).

[3] Barr, T.W. et al. 2011. SpecTLB: a mechanism for speculative address translation. *Proceedings of the 38th Annual International Symposium on Computer Architecture* (Jun. 2011).

[4] Barr, T.W. et al. 2010. Translation caching: skip, don't walk (the page table). *Proceedings of the 37th Annual International Symposium on Computer Architecture* (Jun. 2010).

[5] Basu, A. et al. 2012. Reducing Memory Reference Energy With Opportunistic Virtual Caching. *Proceedings of the 39th annual international symposium on Computer architecture* (Jun. 2012), 297–308.

[6] Bhargava, R. et al. 2008. Accelerating two-dimensional page walks for virtualized systems. *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Mar. 2008).

[7] Bhattacharjee, A. et al. 2011. Shared last-level TLBs for chip multiprocessors. *Proc. of the 17th IEEE Symp. on High-Performance Computer Architecture* (Feb. 2011).

[8] Bhattacharjee, A. and Martonosi, M. 2009. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques* (Sep. 2009).

[9] Bhattacharjee, A. and Martonosi, M. 2010. Inter-core cooperative TLB for chip multiprocessors. *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (Mar. 2010).

[10] Binkert, N. et al. 2011. The gem5 simulator. *Computer Architecture News (CAN)*. (2011).

[11] Chen, J.B. et al. 1992. A Simulation Based Study of TLB Performance. *Proceedings of the 19th Annual International Symposium on Computer Architecture* (May. 1992).

[12] Christos Kozyrakis, A.K. and Vaid, K. 2010. Server Engineering Insights for Large-Scale Online Services. *IEEE Micro* (Jul. 2010).

[13] Couleur, J.F. and Glaser, E.L. 1968. Shared-access Data Processing System. Nov. 1968.

[14] Daley, R.C. and Dennis, J.B. 1968. Virtual memory, processes, and sharing in MULTICS. *Communications of the ACM*. 11, 5 (May. 1968), 306–312.

[15] Denning, P.J. 1970. Virtual Memory. *ACM Computing Surveys*. 2, 3 (Sep. 1970), 153–189.

[16] Emer, J.S. and Clark, D.W. 1984. A Characterization of Processor Performance in the vax-11/780. *Proceedings of the 11th Annual International Symposium on Computer Architecture* (Jun. 1984), 301–310.

[17] Ferdman, M. et al. 2012. 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* (Mar. 2012).

[18] Ganapathy, N. and Schimmel, C. 1998. General purpose operating system support for multiple page sizes. *Proceedings of the annual conference on USENIX Annual Technical Conference* (1998).

[19] graph500 --The Graph500 List: *http://www.graph500.org/*.

[20] Huge Pages/libhugetlbfs: 2010. *http://lwn.net/Articles/374424/*.

[21] Intel 8086: *http://en.wikipedia.org/wiki/Intel_8086*.

[22] Jacob, B. and Mudge, T. 2001. Uniprocessor Virtual Memory without TLBs. *IEEE Transaction on Computer*. 50, 5 (May. 2001).

[23] Jacob, B. and Mudge, T. 1998. Virtual Memory in Contemporary Microprocessors. *IEEE Micro*. 18, 4 (1998).

[24] Kandiraju, G.B. and Sivasubramaniam, A. 2002. Going the distance for TLB prefetching: an application-driven study. *Proceedings of the 29th Annual International Symposium on Computer Architecture* (May. 2002).

[25] Large Page Performance: ESX Server 3.5 and ESX Server 3i v3.5: *http://www.vmware.com/files/pdf/large_pg_performance.pdf*.

[26] Linux pmap utility: *http://linux.die.net/man/1/pmap*.

[27] Lustig, D. et al. 2013. TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs. *ACM Transactions on Architecture and Code Optimization*. (Jan. 2013).

[28] Marissa Mayer at Web 2.0: *http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html*.

[29] Mars, J. et al. 2011. Bubble-Up: increasing utilization in modern warehouse scale computers via sensible co-locations. *Proceedings of the 44th Annual IEEE/ACM International Symp. on Microarchitecture* (Dec. 2011).

[30] McCurdy, C. et al. 2008. Investigating the TLB Behavior of High-end Scientific Applications on Commodity Microprocessors. *Proceedings of IEEE International Symposium on Performance Analysis of Systems and software* (2008).

[31] Memory Hotplug: *http://www.kernel.org/doc/Documentation/memory-hotplug.txt*.

[32] Microsystems, S. 2007. UltraSPARC T2^{TM} Supplement to the UltraSPARC Architecture 2007. (Sep. 2007).

[33] Navarro, J. et al. 2002. Practical Transparent Operating System Support for Superpages. *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Dec. 2002).

[34] Oprofile: *http://oprofile.sourceforge.net/*.

[35] Ousterhout, J. and al, et 2011. The case for RAMCloud. *Communications of the ACM*. 54, 7 (Jul. 2011), 121–130.

[36] Pham, B. et al. 2012. CoLT: Coalesced Large Reach TLBs. *Proceedings of 45th Annual IEEE/ACM International Symposium on Microarchitecture* (Dec. 2012).

[37] Ranganathan, P. 2011. From Microprocessors to Nanostores: Rethinking Data-Centric Systems. *Computer*. 44, 1 (2011).

[38] Reiss, C. et al. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. *Proceedings of the 3rd ACM Symposium on Cloud Computing* (Oct. 2012).

[39] Rosenblum, N.E. et al. 2008. Virtual machine-provided context sensitive page mappings. *Proceedings of the 4th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (Mar. 2008).

[40] Saulsbury, A. et al. 2000. Recency-based TLB preloading. *Proceedings of the 27th Annual International Symposium on Computer Architecture* (Jun. 2000).

[41] Sodani, A. 2011. *Race to Exascale: Opportunities and Challenges*. *MICRO 2011* Keynote address.

[42] Srikantaiah, S. and Kandemir, M. 2010. Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors. *Proceedings of 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (Dec. 2010).

[43] Talluri, M. et al. 1992. Tradeoffs in Supporting Two Page Sizes. *Proceedings of the 19th Annual International Symposium on Computer Architecture* (May. 1992).

[44] Talluri, M. and Hill, M.D. 1994. 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* (Oct. 1994).

[45] TCMalloc : Thread-Caching Malloc: *http://goog-perftools.sourceforge.net/doc/tcmalloc.html*.

[46] Transparent huge pages: 2011. *www.lwn.net/Articles/423584/*.

[47] Volos, H. et al. 2011. Mnemosyne: Lightweight Persistent Memory. *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Mar. 2011).

[48] Waldspurger, C.A. 2002. Memory Resource Management in VMware ESX Server. *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation* (Dec. 2002).

[49] Wood, D.A. et al. 1986. An in-cache address translation mechanism. *Proceedings of 13th annual international symposium on Computer architecture* (Jun. 1986).

[50] Zhang, L. et al. 2010. Enigma: architectural and operating system support for reducing the impact of address translation. *Proceedings of the 24th ACM International Conference on Supercomputing* (Jun. 2010).