A Study of Virtual Memory Usage and Implications for Large Memory

Peter Hornyack   Luis Ceze   Steve Gribble   Dan Ports   Hank Levy
University of Washington

Abstract

The mechanisms now used to implement virtual memory – pages, page tables, and TLBs – have worked remarkably well for over fifty years. However, they are beginning to show their age due to current trends, such as significant increases in physical memory size, emerging data-intensive applications, and imminent non-volatile main memory. These trends call into question whether page-based address-translation and protection mechanisms remain viable solutions in the future. In this paper, we present a detailed study of how modern applications use virtual memory. Among other topics, our study examines the footprint of mapped regions, the use of memory protection, and the overhead of TLBs. Our results suggest that a segment-based translation mechanism, together with a fine-grained protection mechanism, merit consideration for future systems.

1. Introduction

Virtual memory is one of the great ideas in computer science. The basic mechanisms used to implement VM – pages, page tables, and TLBs – have served remarkably well since their introduction in the 1960s. However, applications and their memory use have changed enormously over time, in large part due to the tremendous growth in physical memory capacities. In the late 1960s, systems had at most a few megabytes of physical memory; servers now offer hundreds of gigabytes or even terabytes. This growth is expected to continue and accelerate: although DRAM scaling is slowing [32], non-volatile memory technologies are rapidly advancing and should provide greater density than DRAM [28], further increasing physical memory capacity.

As a result of physical memory growth, varied applications now make heavy use of in-memory data processing, e.g., memcached is pervasive [18], and in-memory databases are proliferating [31, 25, 29]. An important characteristic of many "big-memory" applications is wide, unpredictable memory reference patterns. For example, graph analysis has irregular access patterns [22], and bioinformatics applications tend to have poor locality [1]. Such applications are problematic for virtual memory because they put more pressure on the TLB.

We believe that these technology changes and trends call for a re-examination of VM use by modern applications. As a basis for this reexamination, this paper presents a detailed measurement study of VM use by today's client and server applications. Our goals are to characterize VM usage in order to: (1) highlight the memory behaviors important for future systems, (2) consider how application behaviors may change as physical memory continues to grow, and (3) point out VM features that may be of decreasing importance over time. The major study findings are:

- Application memory needs are diverse. The total amount of virtual memory allocated by our applications ranges across three orders of magnitude. Many applications access all of the virtual memory that they allocate, while others access only a small fraction.
- The maximum number of virtual memory mappings actually used is in the thousands, a relatively small number given that the maximum supported is effectively unlimited.
- Applications perform most of their virtual memory allocations at startup or at well-defined boundaries between workload phases.
- Linking memory protection to virtual address translation artificially reduces the granularity of address translation, which may increase performance overhead.
- Breaking virtual memory mappings into page-sized chunks means that the hardware and OS must track two orders of magnitude more address translations than there are virtual memory mappings, increasing TLB overhead.

These findings, along with trends in memory technology and OS features, suggest that approaches for address translation other than TLB-accelerated page tables may be more appropriate for future VM systems. Separating memory protection from address translation would also offer benefits, such as supporting guard regions in large mappings. To avoid the cost of handling TLB misses, variable-sized segments may be a better fit for address translation than pages and superpages. Some of these ideas are not new, and there are many tradeoffs in using these designs over current mechanisms. This work aims to analyze a broad range of applications so that the right tradeoffs can be made in the future. The rest of this paper describes the measurements we performed, the results that led to our conclusions, and ideas for future virtual memory designs.

2. Study Methodology

Our measurement study evaluated Linux applications on the x86 platform. We instrumented the Linux kernel to report on applications' virtual and physical memory behavior by emitting a trace of all memory events. Our tracing system used infrastructure already present in the Linux kernel and imposed no discernable overhead on the applications we study. We also gathered data from hardware performance counters using the Linux `perf` tool.

Our analysis includes only those events that occurred after the `exec` system call that starts each application. When multi-process applications `fork` child processes, we tracked the entire hierarchy of processes and included in our measurements the memory behavior of all of them. Each new Linux process uses its own private virtual memory map,[1] introducing its own OS and hardware overhead, so we considered all processes to be part of the application. When processes migrated to different cores or when applications had multiple processes on separate cores, the kernel timestamped our trace events using a logical clock that established a sequential ordering of events.

We ran our applications on a server with 24 GB of physical memory, 4 Intel Xeon L5640 cores, and our modified Linux 3.9.4 kernel. We reserved 6 GB of memory to buffer the trace events emitted during application execution. To achieve reproducible results and observe all virtual memory activity, we rebooted our server before executing each application and gathering its trace. Thus, the applications had no pages already present in physical memory when they started running, except any library pages shared with other processes that started when the system booted.

Limitations. Our measurement infrastructure did not capture the virtual and physical memory usage of the kernel itself. Certain applications may cause increased memory use by the kernel; for example, I/O intensive-applications that directly `read` or `write` files (rather than `mmap`-ing them) load file data into the kernel's page cache to improve performance, and specialized applications such as VMware Workstation load their own modules into the kernel. None of the applications we examined in this study rely on these behaviors.

Applications. We analyzed a mix of twelve popular Linux client and server applications, listed in Table 1. Our desktop client applications included two web browsers (Chrome and Firefox) and the LibreOffice office suite. The two web browsers have different architectures: Firefox executes many threads in a single process, whereas Chrome uses one process per browser tab. The experiments ran applications for approximately ten minutes, forming full sessions of executions (e.g., a full web browsing session).

Our server applications include a web server (Apache), a load balancer (HAProxy), and four different data storage applications: Cassandra, Memcached, MySQL, and Redis. Only Cassandra was written in Java; our server used OpenJDK 1.7.0_51. For Apache and MySQL, we run MediaWiki, a typical LAMP stack application, serving a snapshot of the Spanish language Wikipedia site. The MediaWiki data and PHP code resided in a module loaded into Apache's address space. To exercise HAProxy and Redis, we set up external clients to send requests to our server; for the other applications, we generated load using a separate application on the server itself.

Finally, we analyzed three data-processing applications: Dedup, Graph500, and Python. Dedup and Graph500 are benchmarks for compressing data and analyzing large graph data, respectively (Dedup is part of the Parsec benchmark suite [12]). For Python, we executed one of the trace analysis scripts used in this study.

[1] While there are a few exceptions, none applies to our application suite.

App.	Description	Workload
Desktop client applications		
`chrome`	Chrome web browser	Open 30 pages in separate windows, closing every 5th window
`ffox`	Firefox web browser	Open 30 pages in separate tabs, closing every 5th tab
`office`	LibreOffice word processor	Type 50 KB of text then save document
Server applications		
`apache`	Apache web server	Serve five MediaWiki page requests
`proxy`	HAProxy HTTP load balancer	Proxy 300 concurrent connections to three target servers
`cass`	Cassandra column store (Java)	100K inserts followed by 100K gets/updates
`mcache`	Memcached key-value cache	Load 10 GB of data, then serve 80% gets and 20% updates at 25,000 req/s
`mysql`	MySQL relational DB	Serve five MediaWiki page requests
`redis`	Redis key-value store	Continuous puts of 1 KB values from four clients
Data processing applications		
`dedup`	Data compression benchmark	Input size: 672 MB disk image file
`graph`	Graph500 graph benchmark	Scale factor 24 (10 GB graph)
`python`	Python runtime environment	Execute analysis script for this measurement study

Table 1: The suite of twelve applications executed and analyzed in this study.

Terminology. Throughout this paper, we describe the memory allocations of each application in terms of *virtual memory areas*, or *VMAs*. Each VMA represents an item of code or data stored in a contiguous region of virtual memory and spans one or more contiguous virtual pages; the minimum size of a VMA is one page (4 KB on x86 and ARM architectures). The entire VMA must have the same set of memory access permission bits (read, write, or execute). VMAs may be private to a particular process or shared across processes; private VMAs may still be read-shared across processes in physical memory (e.g., after a `fork`), but are marked as copy-on-write. A VMA is equivalent to a `struct vma` in the Linux kernel code. The full set of VMAs for a process is referred to as its *memory map*.

This paper uses the verbs "allocate" and "deallocate" to describe the creation or removal of VMAs in a process' virtual address space. The term "mapped" describes the VMAs present in a process's virtual address space at any specific time. A VMA may be *file-backed*, meaning that it is associated with a particular file that has been mapped into the virtual address space. VMAs that are not file-backed are *anonymous*; we refer to anonymous VMAs with read-write access permissions as *heap* VMAs. A VMA with no read, write, or execute access permissions set is known as a *guard region*.

3. Results

Our study aims to answer six basic questions about virtual memory usage by today's applications:

App.	Total VMAs	% Anon	% File	%Guard	% Libs
redis	34	38	24	6	32
proxy	44	20	30	0	50
dedup	51	45	14	20	22
graph	57	30	19	11	40
mysql	124	31	15	19	35
python	312	25	47	13	14
mcache	363	50	2	44	4
office	945	10	28	2	61
cass	1021	36	14	32	18
apache	1235	24	60	1	15
ffox	2724	29	22	3	46
chrome	7666	32	43	8	17

Table 2: Maximum number of VMAs mapped in each application's address space at any point during execution, and the category distribution.

1. How many and what type of VMAs do applications use?
2. How do VMA usage patterns change over time?
3. How do applications use differently-sized VMAs to map their virtual address space?
4. How are VMAs allocated and resized?
5. How are protection modes used?
6. How is physical memory used?

3.1. Number and Type of VMAs

Current systems can support an effectively infinite number of VMAs mapped into an application's virtual address space, limited only by the size of the address space and storage capacity for page tables. However, applications do not make use of this flexibility. Table 2 shows the maximum number of VMAs mapped by each application. While we see a three order of magnitude range in VMAs, even at the high end the number is relatively small. Over half of our applications use fewer than 400 VMAs. `chrome` and `ffox` are the highest – web browsers have much of the functionality and complexity of an OS and rely heavily on virtual memory management. `chrome` and `apache` have high VMA counts because they use multiple processes, each with its own private virtual memory map. Considered on a per-process basis, `apache` and `chrome` use only hundreds of VMAs per process.

Table 2 also classifies the VMAs according to the high-level type of data that each one maps. *Libs* includes non-writable VMAs used to map dynamically linked libraries. *Guard* VMAs are guard regions with no access permissions. *File* VMAs are used to map files (other than shared libraries) into the virtual address space, while *Anon* includes the remaining anonymous VMAs, primarily read-write heap VMAs.

Many applications make heavy use of memory-mapped files, and file-backed VMAs outnumber even anonymous heap VMAs in some applications. Surprisingly, our analysis shows the large number of VMAs used to map shared libraries and guard regions. For example, 46% of VMAs used in `ffox` are for libraries. When each shared library is dynamically linked into a process's address space at startup or during execution, four VMAs are typically allocated: one each for code, read-only data, and read-write data (e.g., global variables that the code may change) and a guard region with no access permissions. The VMAs included in the *Libs* category do not include those for read-write data since a write to these areas would trigger a copy-on-write, and the VMA would no longer be shared.

Summary. The number of virtual memory areas needed to hold all of a process's code and data items ranges across three orders of magnitude, from tens of VMAs to a few thousand. Many applications allocate large numbers of VMAs for memory-mapped files and guard regions. Additionally, applications may need to read data and execute code from tens or sometimes hundreds of shared libraries.

Design Implications. Although the number of VMAs supported is effectively unbounded, in practice applications use relatively few. The overhead of performing virtual address translation using pages and TLBs is growing; our analysis shows that this overhead results more from the breaking of virtual memory mappings into many small pages than from the total number of VMAs, which is modest. Therefore, future virtual memory systems should aim to increase page sizes to better accommodate large VMAs or even map each VMA to physical memory directly, i.e., without dividing it into pages (as with direct segments [6]).

3.2. VMAs Changes over Time

Figure 1 (left) plots the number of VMAs mapped over time. In all applications we studied, the number of VMAs increases dramatically when the application starts. This can be attributed to the dynamic linking of shared libraries or to the allocation of thread data. One VMA is initially allocated for each thread's stack, a substantial effect for heavily multi-threaded applications like `ffox` and `cass`. Applications may allocate other per-thread VMAs as well.

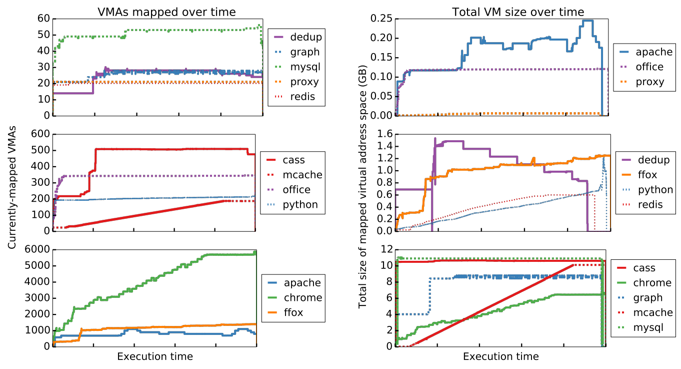

Figure 1: Total count of VMAs mapped in the application's virtual address space across its execution time (left). Time is normalized across the x-axis: from one end of the x-axis to the other end is the total execution time for the application. Total amount of virtual memory mapped by each application during its execution (right).

After startup, applications exhibit different allocation patterns. `proxy` and `redis` allocate *no* new VMAs after startup; however, existing VMAs may be resized, as described in Section 3.4.1. `mysql` and `office` allocate just a few new VMAs during their execution. Other applications exhibit clear phased behavior, e.g., `graph` has a stable period as it builds the graph in memory, followed by a series of graph traversals that allocate and deallocate VMAs; `dedup` shows a similar phase shift during its execution as well. Finally, `chrome`, `ffox`, `apache`, and `mcache` allocate new VMAs in response to new workload requests; for example, each new tab or window opened in the web browsers leads to the creation of many new VMAs. The particular web page that is visited also impacts allocation. For example, the heavy execution of JavaScript in the browser causes new VMAs to be allocated, and the largest jump in the `chrome` and `ffox` plots occurs when visiting a multimedia-heavy website that loads plugins, such as Flash.

The number of VMAs used by an application almost always increases over time. VMA deallocations are relatively rare and are often quickly followed by more allocations. Applications that deallocate VMAs include `dedup`, `apache`, and `chrome`. `apache` and `chrome` deallocations are caused by terminating processes, i.e., after a request has been handled or a browser window has been closed.

Figure 1 (right) shows the total amount of virtual memory over time, which varies across three orders of magnitude – from barely tens of megabytes for `proxy` to just over ten gigabytes for `mysql` and `cass`. The number of VMAs used does not reliably predict of the amount of virtual memory an application uses; while `chrome`, `ffox` and `cass` use both a large amount of virtual memory and many VMAs, `mysql` and `graph` use few VMAs to map a lot of virtual memory, and `apache` uses many VMAs for a small amount of virtual memory. In today's systems with demand paging, applications may allocate more virtual memory than they actually access and bring in to physical memory. We discuss demand paging and resident memory in Section 3.6.1.

Summary. Applications may allocate VMAs in response to client requests and application workload requirements. However, in many applications the number of VMAs allocated during execution is significantly smaller than the number allocated at startup. VMA deallocations occur infrequently. The number of VMAs used to map virtual memory does not always increase as the amount of mapped virtual memory increases.

Design Implications. At certain points during an application's execution (e.g., shortly after application startup or after a new phase of execution has begun) it would be beneficial for the OS to reconsider the application's memory mappings. At these points, the OS could aggressively optimize physical memory layout and the mappings from virtual to physical memory, for example, by compacting physical memory regions to make room for more superpage mappings. By enhancing the OS so it can recognize these points, or by adding an interface so applications can notify the OS when these points occur, optimizations could be performed that would have the greatest impact on the application's execution.

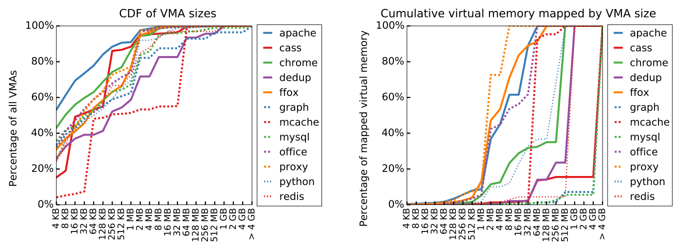

Figure 2: Cumulative distribution of of total mapped VMAs and their sizes.

3.3. VMA Size Distribution

Figure 2 shows how applications use differently-sized VMAs to map their virtual address space. Figure 2 (left) shows a CDF of VMA sizes. Figure 2 (right) divides the cumulative size of the VMAs by the total size of virtual memory that the application has mapped. Both of these figures are calculated when each application's execution has mapped the most virtual memory. They illustrate a pattern similar to file systems [17] and other domains: although most VMAs are small, the total amount of mapped virtual memory is dominated by a few large VMAs. In most applications, 4 KB VMAs (the base page size on x86, and hence the minimum VMA size) are the most frequent, but many other VMA sizes between 4 KB and 256 KB are widely used. Data-intensive applications (like `graph`, `dedup` and `mcache`) use larger VMAs. `mcache`, an outlier, has nearly 50% of its VMAs that exceed 32 MB.

Figure 2 (right) also shows that the majority of virtual memory in most applications is mapped using VMAs with sizes between 2 MB and 1 GB (the two x86-64 superpage sizes). The rightmost applications in this figure use just one or two large VMAs to map most of their data. `redis` keeps most of its key-value data in a 600 MB VMA; `dedup` uses two VMAs with the same size as the input file, about 700 MB each; `cass` uses one 10 GB VMA for its Java heap; `graph` uses two VMAs that are just over 4 GB to hold its large graph data; and `mysql` uses a 10 GB VMA for its buffer pool. These huge VMAs are all anonymous read-write heap regions.

Summary. Most VMAs are small, but large VMAs are responsible for mapping most of the virtual address space. Most virtual memory is mapped with VMAs whose sizes lie in between the superpage sizes that x86-64 hardware provides or with VMAs larger than the largest superpage size. Some, but not all, memory-intensive applications allocate one or two very large VMAs to hold their heap data.

Design Implications. The effectiveness of using fixed-size superpages to reduce address translation overheads is fundamentally limited. Most virtual memory is mapped using VMAs larger than 2 MB; thus, even if x86-64 superpages were used whenever possible, each of these VMAs would still require many 2 MB superpages, each with its own TLB entry. Similarly, the applications with the largest memory usage rely on VMAs that exceed 1 GB, the largest x86-64 superpage size; hence multiple 1 GB superpages would be required to map these VMAs. A small set of fixed superpage sizes will, therefore, fail to effectively map all VMAs for all applications. New virtual memory designs with more flexible address translation mechanisms would significantly reduce the number of mappings needed for address translation; for example, every VMA could be mapped to physical memory using exactly one variable-length segment, or perhaps more flexible paging implementations could allow new superpage sizes to be set for every application or on every reboot.

3.4. Memory Allocation and VMA Resizing

Linux applications can allocate virtual memory directly from the kernel using the `mmap` system call (or the legacy `sbrk`) or via a memory allocation library. Using source code analysis and the Linux `ltrace` library tracing facility, we observed that our applications generally did not allocate anonymous memory directly from the kernel. This applies even to high performance server applications, such as `mcache` and `redis`; these applications, like `proxy` or `dedup`, do not call `mmap` in their source code.

Although applications nearly always allocate their memory using a memory allocation library, they do not always use the same one. Most use the standard glibc `malloc`; `redis` uses `jemalloc` by default, and `ffox` uses its own custom Mozilla memory allocator. These allocators take different approaches to obtaining memory from the operating system. `mcache` uses numerous 64 MB VMAs to map nearly 10 GB of virtual memory. In contrast, `redis` uses a single large VMA (600 MB) to map nearly all its virtual memory. However, Figure 1 shows that the amount of virtual memory used by `redis` (like `mcache`) grows throughout its execution, indicating that `redis` continually expands the single VMA that holds its data. `mysql` takes a third approach, common for databases: it pre-allocates a single, huge (10 GB) VMA for its buffer pool, then manages memory itself within that region.

Summary. Even high-performance server applications rely on the `malloc` library; few applications `mmap` their own memory.

Design Implications. Memory allocators included in standard language libraries and runtimes should be considered part of the virtual memory system as a whole. Future virtual memory designs to change the interface between the OS and user-space could thus be kept within the memory allocation library, allowing most application code to remain unmodified. Changing the interface to `malloc`, however, would require extensive application changes.

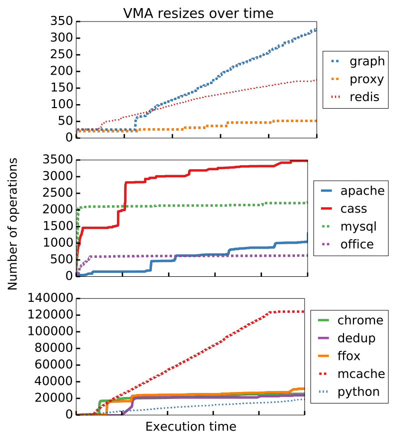

Figure 3: Count of resize operations that either expand or decrease the size of a VMA. Application execution times are normalized across the x-axis.

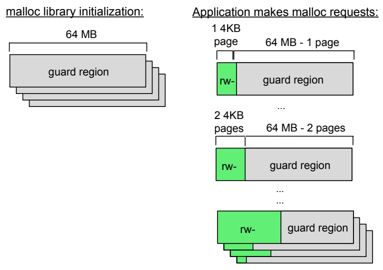

Figure 4: Demonstration of `malloc`'s *reservation* behavior, which relies on VMA resizing and page-granularity memory protection.

3.4.1. Resizing and Reserving VMAs

In addition to being allocated and deallocated, VMAs can be *resized*, e.g., by using the `mremap` system call. Figure 3 shows the number of VMA resizes performed by each application throughout its execution. Every application performs some resize operations, both expanding and decreasing its VMA sizes at process startup and when shared libraries are dynamically loaded.

In addition to this initial VMA resizing, many applications perform thousands of resize operations throughout their execution due to memory allocator behavior. For `jemalloc`, used by `redis`, resize operations are used to allocate more memory, as described above.

The standard glibc `malloc` performs many resize operations as well. Figure 4 demonstrates how `malloc` *reserves* contiguous 64 MB areas of virtual memory to serve an application's small object requests. Initially, these VMAs have no access permissions set. As the application makes repeated `malloc()` requests, these areas are converted from guard regions to read-write memory by extending a read-write VMA at one end by one page and shrinking the guard VMA at the other end by one page. Each of these reserved areas becomes a "free list" for a particular allocation size, and `malloc`'s metadata techniques require that these areas be contiguous in memory as much as possible; by reserving the virtual memory in advance with a guard region, `malloc` ensures that when the read-write VMA must be extended, the adjacent virtual memory will usually be available and will not collide with other VMAs.

Both `dedup` and `mcache` are heavy users of `malloc`, which explains why they have the most resize operations in Figure 3. The `malloc` behavior described here also explains why `mcache` keeps all of its virtual memory in 64 MB chunks, as this is the default size of `malloc`'s reserved areas; when the application fills up one of these areas, `malloc` simply allocates a new one and continues to fill it in the same way.

Summary. Applications commonly resize VMAs. glibc `malloc` performs resize operations frequently due to its behavior that reserves large contiguous areas of virtual memory, but only applies read-write permissions one page at a time.

Design Implications. Since paged virtual memory has been a feature of OSs and processor architectures for decades, it is not surprising that applications have evolved to take advantage of it. Resizing VMAs would be a challenging behavior for virtual memory systems that do not implement fine-grained paging for address translation, but the frequency of resizing could be reduced by modifying the standard `malloc` library, as well.

Having separate mechanisms for address translation and memory protection could be beneficial for the use of reserved virtual memory areas: the entire reserved VMA could be mapped into physical memory at once. Then, only the protection boundaries would need to be changed on a resize, eliminating the need to relocate physical memory.

3.5. Uses of Memory Protection

Virtual memory is used not just for memory translation but also for protection. This section analyzes the connection between application data types and their memory permissions, and the frequency of permission changes.

3.5.1. Types of Data in VMAs

Figure 5 helps us to visualize part of `ffox`'s virtual address space during its execution. The vertical axis indicates combinations of protection bits and VMA types: VMAs may have read, write and execute permissions set or unset; may be private to this process or shared with other processes; and may be file-backed or anonymous. Note that even "private" VMAs may still be shared across processes in physical memory, by using copy-on-write.

The `ffox` address space appears fragmented; this is typical of an application that dynamically links to numerous shared libraries. As described previously, loading each shared library creates four VMAs: code (`r-xpf`), read-write global data (`rw-pf`), read-only data (`r-pf`), and a guard region (`--pf`). These VMAs are typically contiguous in virtual memory; this is easy to see in Figure 5 where the bars for these four permis-

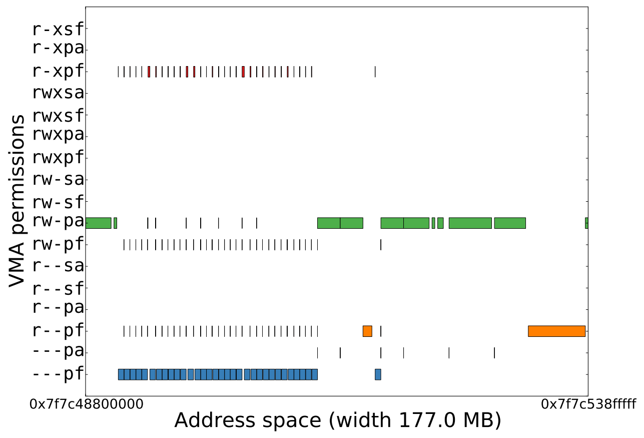

Figure 5: Visualization of part of `ffox`'s virtual address space. The horizontal axis is the process' linear virtual address space; each horizontal bar in the plot represents a VMA allocated with the particular permission combination listed on the vertical axis. The permission bits are read (r), write (w), and execute (x); shared (s) vs. private (p); and file-backed (f) vs. anonymous (a).

sion rows are usually aligned vertically. Shared library VMAs are usually small, often just one 4 KB page.

Heap VMAs with `rw-pa` permissions comprise most of the address space for `ffox`, like most applications we studied. Many of `ffox`'s heap VMAs are near each other in virtual memory, but `ffox`'s memory allocator prevents them from being contiguous. Likely reasons for this behavior are to cause unintentional accesses beyond a VMA boundary to hit unmapped memory and segfault rather than hitting adjacent memory and causing corruption, or to make it easier to unmap these VMAs. Each heap VMA is allocated using `mmap`; no applications use the legacy `sbrk` interface.

Figure 5 also shows some anonymous guard regions with `--pa` permissions; these VMAs may be used to protect thread stacks (which are allocated with `rw-pa` permissions), or to detect overflows in other kinds of VMAs. While these anonymous guard regions are small, other applications use large ones, e.g., to reserve contiguous memory as described in Section 3.4.1.

We have observed other types of data being mapped into the virtual address space of some applications that are not shown in Figure 5. Every process has one `rw-pa` VMA for its stack, located at the extreme upper end of the address space. Some applications use shared memory areas. Applications use shared file-backed mappings both to access persistent data and to create a shared region for inter-process communication via the mapping of temporary files. Anonymous shared VMAs are also used for inter-process communication.

Summary. The Linux dynamic linker "fragments" virtual address space by splitting up the code and data regions for each shared library into separate VMAs. Applications use guard regions to detect a variety of invalid memory accesses and to prevent writeable VMAs from being placed adjacent to each other in virtual memory. Sharing of virtual memory across processes is fairly uncommon.

Design Implications. Figure 5 demonstrates that some applications could coalesce their VM into a small number of VMAs. As described above, `ffox` allocates many heap VMAs that are *nearly* adjacent to each other; coalescing these into just one VMA could improve translation performance by leading to better use of superpages. Future virtual memory systems may need to reconsider whether guard regions and memory placement are the optimal mechanisms for avoiding corruption due to overflow because these mechanisms prevent coalescing. A virtual memory system with separate ways to configure address translation and memory protection could use larger VMAs for address translation while still preserving the guard regions between heap allocations.

3.5.2. Frequency of Protection Operations

We also examined the occurrence of protection change operations over time; we do not present the data here, but note that the occurrence of protection changes tends to mirror the occurrence of VMA allocation and VMA resize operations. In other words, protection changes appear to occur in conjunction with other VMA modifications, and not in isolation.

Design Implications. While our analysis suggests that current applications do not rely heavily on frequent or sophisticated use of memory protection, that does not mean they would not *like* to. The memory protection provided by Linux on x86-64 is limited in many ways: memory protection checking and address translation are coupled, the minimum granularity for protection changes is the 4 KB base page, and hardware protection domains are at the process level rather than the thread level. Today's applications likely avoid using memory protection because of these limitations; for example, `mysql` and `redis` cannot use different memory protections within their buffer pools without forcing their large VMAs to be broken up into multiple smaller VMAs. Future virtual memory designs should consider decoupling memory protection from address translation to allow for finer-grained memory protection and more flexible protection domains. Previous work has shown that this flexibility would be valuable to many applications [13, 35].

3.6. Connecting Virtual memory To Physical Memory

In this section we shift from application behavior in the virtual address space to usage and mapping of physical memory.

3.6.1. Demand Paging and Resident Physical Memory

Demand paging, a feature of current virtual memory systems, does not establish the mapping from a virtual page to a physical page until the application accesses the virtual page for the first time. We used our tracing infrastructure to track the amount of *resident* memory, or virtual memory that has been demand-paged into physical memory, for each application over its execution.

App	Ratio	Resident	Virtual	Diff.
`apache`	19.5%	38.41 MB	206.94 MB	168.53 MB
`cass`	96.9%	10.19 GB	10.62 GB	441.36 MB
`chrome`	3.9%	254.98 MB	6.63 GB	6.38 GB
`dedup`	95.2%	1.40 GB	1.48 GB	86.14 MB
`ffox`	39.5%	489.72 MB	1.24 GB	781.68 MB
`graph`	99.4%	8.66 GB	8.79 GB	140.74 MB
`mcache`	99.6%	9.96 GB	10.11 GB	146.96 MB
`mysql`	8.3%	824.91 MB	11.03 GB	10.23 GB
`office`	94.1%	80.16 MB	86.02 MB	5.87 MB
`proxy`	93.1%	1.27 MB	1.36 MB	100.00 KB
`python`	98.4%	1.17 GB	1.21 GB	31.71 MB
`redis`	92.2%	558.77 MB	612.23 MB	53.47 MB

Table 3: Ratio of resident physical memory to mapped virtual memory. The 95th percentile ratio during each application's execution time is shown, along with the absolute difference between the virtual and physical memory sizes at that time.

Table 3 shows the extent to which our applications rely on demand paging to avoid making certain virtual memory resident in physical memory. After every virtual or physical memory allocation or deallocation, we calculated the ratio of resident physical memory to mapped virtual memory. We then took the 95th percentile of these ratios across the application's entire execution to approximate the "maximum" ratio while filtering out instantaneous spikes or drops in the ratio. At this 95th percentile ratio, the table also displays the absolute amounts of resident and virtual memory, and the difference between these values.

Table 3 shows a clear division between those applications that access and load into physical storage nearly all of the virtual memory that they allocate, and those that do not. Of the non-intensive applications, `chrome` and `mysql` access less than 10% of their virtual memory during execution, while `apache` and `ffox` access less than 40%. The data is somewhat misleading for `apache` and `chrome` because they are multi-process applications: each new process inflates the total virtual memory size because it gets its own virtual memory map, but some of this memory is marked as copy-on-write and is shared in physical memory across processes. Unfortunately, our current tracing infrastructure cannot tell us how much of each application's resident physical memory is shared across multiple processes.

We suspect that `ffox`'s reliance on hundreds of shared libraries results in its sparse use of its virtual memory, i.e., many of the libraries in its address space are never accessed during execution. Similarly, `mysql` allocates more than 10 GB of virtual memory that is never accessed and therefore never loaded into in physical memory. The reason for this discrepancy is that `mysql` sets its buffer pool size statically to 10 GB, but our current workload accesses only a small portion of the data kept in the database.

When viewed as a timeseries across the application's execution (not shown here), *the ratio of resident memory to virtual memory generally remains constant after application startup*. The only applications with significant changes in this ratio during execution are `apache`, as processes are started and killed to handle requests, and the `chrome` and `ffox` web browsers, because of their dynamic workloads.

Finally, we note that even in applications that access nearly all of their virtual memory, making it resident in physical memory, the amount of non-resident virtual memory is still sizable. For example, `cass`, `graph` and `mcache` all have a resident-to-virtual ratio greater than 96%, but still have hundreds of megabytes of virtual memory that has not yet been accessed and made resident.

Summary. Many applications load into DRAM more than 90% of their virtual memory. The use of paged VM with on-demand page loading benefits applications with large address spaces and unpredictable dynamic workloads, such as web browsers and web servers.

Design Implications. Demand paging is clearly a useful feature for desktop applications with large, sparsely used address spaces. However, in many environments today, demand paging is not nearly as critical, and its usefulness will likely decrease further as physical memory capacities grow. Most applications that run on data center servers load nearly all of the virtual memory that they allocate; this is demonstrated by our data for `cass`, `mcache` and `redis`. For data center servers, it is not useful to defer backing virtual memory with physical memory: these machines are provisioned to ensure the correct amount of physical memory is available so they will never swap. In fact, many of these applications would benefit from having all their virtual memory allocations backed by physical memory immediately to avoid introducing page fault latency at unpredictable points in the future.

It may be useful for future virtual memory systems to provide efficient alternatives to demand paging. This would let applications that plan to use all of their virtual memory avoid unpredictable page faults later, and it could potentially ease the use of superpages.

3.6.2. TLB Miss Handling

To measure the impact of TLB misses during the application execution, we use the hardware performance counters in our server's Intel Westmere CPU. We determined the percentage of execution time that is spent handling misses in the data TLB by enabling the DTLB_MISSES.WALK_CYCLES counter, which counts the number of cycles that the x86 hardware page table walker spends walking page tables after a second-level TLB miss, then dividing by the total cycles spent executing each application's processes. To best understand the impact of TLB misses, we enabled this performance counter only after each application had been started, and only during the most intensive period of the application's workload (e.g., for `mcache` and `cass`, we only measured TLB misses that occurred while

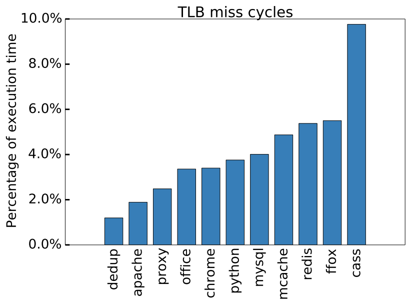

Figure 6: Percentage of total execution cycles spent walking each application's page table to handle data TLB misses. The percentage of execution time for `graph` is 58%, outside the y-axis scale; we have omitted it from this plot.

a mix of gets and puts were being performed, and not while the store was initially being loaded with data).

Figure 6 shows the percentage of execution time spent handling TLB misses for each application. Nearly all page table walk cycles result from load misses in the TLB, rather than store misses. `graph` is off the charts because of the extremely poor memory locality of its graph analysis workload; Basu et al. [6] found similar results for this application and reported that the time spent on TLB misses dropped to about 10% with 2 MB superpages enabled and to about 1.5% with 1 GB superpages enabled.

The other applications spend a few percent of their execution cycles waiting for the page table walker to handle TLB misses. The data storage applications – `mysql`, `mcache`, `redis` and `cass`– spend more time handling TLB misses because of their large data sets and the random requests to keys or records that our workloads make. Particularly in virtualized environments, where the cost of handling a TLB miss may be as much as four times higher [7], even this relatively low miss rate can be significant. There is a weak positive correlation between the amount of virtual memory that an application uses and the time spent handling TLB misses. Besides this, there does not appear to be a strong relationship between the data in Figure 6 and the other virtual memory data that we have presented. For example, using more VMAs does not necessarily mean that an application will spend more time handling TLB misses, as `graph`, `redis` and `mysql` are among the applications with the fewest VMAs.

Finally, we note that none of our applications was explicitly compiled or configured to take advantage of superpages. Using superpages would likely reduce the amount of time spent handling TLB misses to some degree. Other recent research has explored TLB misses, the impact of superpages, and other optimizations in more detail [6, 9, 10, 11, 26, 27].

Summary. The amount of time that applications spend handling TLB misses is non-trivial, generally a few percent of the application's execution cycles, but as high as 58% for certain workloads. If used effectively, superpages would likely reduce the miss handling time.

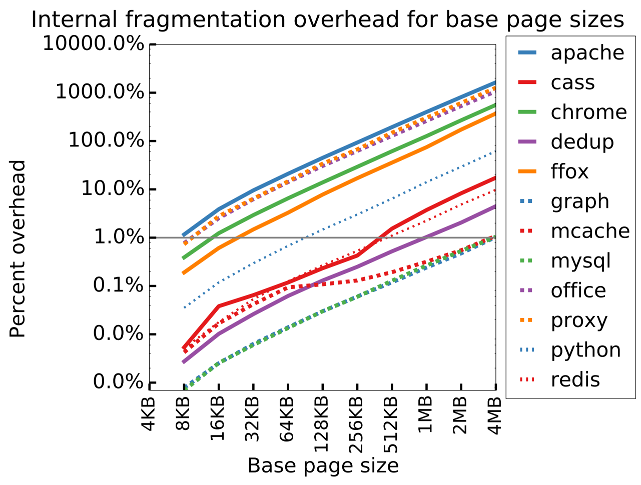

Figure 7: Additional internal fragmentation that would be introduced by increasing the base page size beyond 4 KB. This overhead is in addition to any internal fragmentation that currently exists with 4 KB base pages.

Design Implications. The frequency of TLB misses for an emerging class of large-memory data analysis applications motivate the need for a virtual memory system that does not rely so heavily on the TLB. We make such a proposal in Section 4, based on the results of our study.

4. Future Virtual Memory Systems

Based on the results of our measurement study of virtual memory behavior, we offer two high-level recommendations for the design of future virtual memory systems:

1. flexible, segment-based address translation capability
2. decoupled, page-based protection mechanisms

4.1. Variable-Sized Address Translation

One problem we observed with today's virtual memory systems is the use of rigid, fixed-size pages for performing virtual-to-physical address translation. The 4 KB base page size is a relic of the 1960s; even combined with larger superpages, it results in a tremendous number of mappings that must be tracked by the virtual memory system. This number only increases as the size of physical memory continues to grow.

The result is that TLBs map an increasingly small fraction of the virtual address mappings. Despite much effort to improve TLB performance – e.g., using fully-associative caches that consume up to 15% of chip power [19] – it has been a losing battle. We have seen that TLB misses comprise more than half the cycle cost of a graph benchmark that represents a class of emerging data analytics applications. Others have reported that page-based virtual address translation imposes a considerable cost in virtualized environments [7].

Application	4KB	4KB, 2MB	4KB, 2MB, 1GB	VMAs
`apache`	62,991	15,468	–	1,235
`cass`	2,798,628	31,563	26,453	1,018
`chrome`	947,939	244,292	–	7,666
`dedup`	400,855	3,297	–	46
`ffox`	327,201	93,674	–	2,716
`graph`	2,304,846	6,368	2,280	56
`mcache`	2,648,846	81,582	–	362
`mysql`	2,892,282	7,687	2,577	98
`office`	31,890	3,785	–	950
`proxy`	1,646	624	–	44
`python`	319,837	15,281	–	300
`redis`	156,732	877	–	33

Table 4: Minimum number of virtual to physical memory mappings needed when each application's VMAs are broken up using x86 base pages and superpages. Dashes indicate that 1 GB superpages do not help for this application.

4.1.1. Why Not Larger Pages?

Possible ways to reduce the number of virtual to physical memory mappings include increasing the base page size, or using existing architectural support for multiple page sizes. The former wastes memory due to internal fragmentation. Figure 7 shows the amount of additional fragmentation that would be introduced in our test applications by increasing the base page size. Because many of these applications create numerous small mappings, the benefits are limited: a modest increase in the base page size to perhaps 16 KB or 32 KB would be feasible, but larger base page sizes would introduce too much additional fragmentation. The latter is not likely to significantly reduce the total number of virtual to physical memory mappings required, either. To illustrate the limitations of fixed-size pages and superpages, Table 4 counts the minimum number of mappings needed for each application when the largest page sizes available are used for each VMA. The final column shows the total number of VMAs. In every case, the resulting number of virtual-to-physical mappings is at least one order of magnitude greater than the number of VMAs.

The results in Table 4 assume that superpages are used perfectly: every memory region is mapped using the largest page sizes possible. In practice, achieving this requires substantial support from operating systems and applications [33, 24], and Linux application and kernel developers have repeatedly complained of the complexity of implementing and using superpages [14, 15, 16].

4.1.2. Segment-Based Address Translation

We believe that future virtual memory systems should look to perform virtual address translation *without* using fixed-size pages in order to move the number of virtual to physical mappings closer to the actual number of VMAs that applications allocate. By doing this, reliance on the small TLB cache will be reduced and application performance will improve.

Figure 2 shows that the sizes of applications' VMAs vary widely, and that for each application may use different VMA sizes to map the majority of its virtual memory (e.g., 64 MB VMAs for `mcache`, but 512 MB VMAs for `chrome`). Because these VMA sizes do not correspond well to the x86 page sizes, a large number of virtual to physical mappings are required and TLB performance suffers. This demonstrates an inherent limitation of trying to use a small set of fixed page sizes to map applications' VMAs and suggests that a more flexible approach to virtual address translation is needed.

We advocate variable-sized *segments* for address translation in future virtual memory systems. Recent research has already demonstrated the benefits of using one large segment for heap allocations in certain server-class applications [6]; we believe that extending this approach to allow applications to use many segments could be fruitful for the more general applications we have studied here. Managing physical memory using segments has some well-known drawbacks, such as increased external fragmentation and the difficulty of expanding segments, but *operating systems today are already dealing with these issues again due to the use of superpages*. In today's systems with huge amounts of physical memory that may be running latency-sensitive web workloads, trading off some amount of memory fragmentation to eliminate TLB misses when segments are accessed is likely worthwhile.

Virtual memory features that rely on small page granularity, such as demand paging, would have reduced effectiveness with segment-based address translation because they would have to operate on entire segments at a time. However, our results suggest that many important applications do not rely heavily on demand paging. We believe that the importance of demand paging and other page-based virtual memory features will continue to decrease in future systems; for example, if future computers use only non-volatile memory [2] with no block storage devices, then demand paging no longer makes sense: all data are already resident in memory at all times.

4.2. Decoupling Protection from Address Translation

We have seen many cases where memory protection requirements cause applications to allocate more, smaller VMAs than would otherwise be necessary. For example, Table 2 shows that applications frequently use guard regions to protect against overflows and invalid memory accesses. Figure 5 shows that `ffox` allocates its heap VMAs with small gaps between them to avoid data corruption on buffer overflows; it also shows that the Linux dynamic linker breaks up every shared library into multiple adjacent VMAs with different permissions.

These examples arise because current architectures combine memory protection checks and virtual address translation into the same mechanism: both are specified in the page table entries. We advocate rethinking CPU architectures to separate these two concerns. This could be achieved by using a separate page-based mechanism to specify access permissions along with our proposed segment-based address translation mechanism.

Decoupling protection checks from address translation would provide an opportunity to specify memory protection on individual pages *within* a wider virtual-to-physical memory mapping. This could reduce the number of address mappings required even beyond the number of VMAs shown in Table 4. For example, instead of allocating numerous 1 MB VMAs that are not directly adjacent, `ffox` could allocate a single larger VMA but still specify guard protections on pages within the VMA to detect buffer overflows. This approach could potentially be pushed even further to enable finer-grained protection (e.g., at cache-line granularity).

One might be concerned that this protection mechanism would introduce many of the same complexities as existing page-based virtual memory translation, such as requiring a TLB-like protection cache. An important difference is that the address translation must be performed early in the CPU pipeline because the memory access depends on the result; however, the protection check does not need to happen this early. It can be performed in parallel with the memory access and other processing, as it can be completed any time before the instruction commits.

5. Related Work

An extensive body of research on virtual memory systems and optimizations extends back more than fifty years. In this section, we briefly focus on architectural research related to the learnings and implications of our measurement results. Particularly relevant in this context are recent studies of enhanced TLB design, extensions to address translation for big memory applications, and single-address space systems.

Enhanced TLB designs. As awareness of the performance overheads of TLBs has grown in recent years, many research projects have attempted to make microarchitectural enhancements to improve TLB performance in three broad categories: increasing TLB coverage (or "reach") to reduce miss rates, prefetching TLB entries to reduce miss rates, and reducing the cost of handling TLB misses.

Coalesced Large-Reach TLBs [27] adds logic to TLBs that merges the entries for up to 8 contiguous pages into one entry, leaving room for more entries to increase TLB reach. Pham et al. [26] extend this approach to coalesce mappings that are not perfectly contiguous but do exhibit weakly clustered spatial locality. Others have attempted to share TLB cache structures more effectively [9]. Basu [5] explored a mechanism to allow a TLB entry to be used for a superpage of any size. Swanson et al. [34] instead propose adding another layer of indirection inside of the memory controller itself to increase TLB reach.

Recency prefetching [30] extends page table entries to keep track of which pages are typically referenced together temporally and then uses this history to prefetch TLB entries when one of the pages is referenced. Kandiraju and Sivasubramaniam [20] surveyed the effectiveness of prefetching techniques previously proposed for data caches and proposed a new prefetching mechanism that reduces the spatial overhead of recency prefetching. With the increasing prevalence of multicore processors, Bhattacharjee and Martonosi proposed inter-core cooperative TLB prefetchers [11] to take advantage of common TLB reference patterns across cores.

Most current Intel and AMD x86 architectures use MMU caches to reduce the cost of the page table walk that must be performed after a TLB miss. These structures have not received much research attention until recently, however. Barr et al. [3] surveyed the different MMU cache structures used by Intel and AMD processors and evaluated several alternate structures. This work demonstrated that radix tree page tables with MMU caches should require far fewer total memory accesses for address translation than hash-based inverted page tables. Bhattacharjee [8] proposed two further improvements to the MMU cache: a combined OS and hardware technique to coalesce MMU cache entries, and a shared MMU cache rather than private, per-core MMU caches. SpecTLB [4] uses speculation rather than more caching to hide the latency of most page table walks on TLB misses.

While these techniques to enhance TLBs improve performance, they do not address the underlying cause of address translation overhead: the rigidity of a page-based approach that tries to map all applications' virtual memory areas using a small set of page sizes. These techniques also do not address the second pressing problem with current virtual memory systems, the complexity of managing multiple page sizes; in fact, they might be a step backwards in this regard because they add even more complex logic to the CPU, MMU or OS. Moreover, even with these enhancements to improve TLB reach, the effectiveness of a TLB's cache-based approach is still limited for applications that use large amounts of memory but have poor locality. This limitation is reflected in the results for some of this related work; for example, [27] is least effective for `Tigr` (DNA sequence assembly) because of its poor temporal locality in the coalesced TLB, and [8] notes that `Canneal` (simulated annealing) has high TLB overheads because of its "pseudo-random access patterns."

With a segment-based approach, these techniques may become unnecessary because TLBs can be eliminated.

Direct segments for big memory applications. Recent research by Basu et al. [6] proposed adding support for a single "direct segment" in hardware and the OS. Applications that use a large amount of memory can be configured to place all of their heap allocations in the direct segment, which is contiguous in both virtual and physical memory. All accesses to virtual addresses within the direct segment are then intercepted and translated to physical addresses with a simple offset calculation from the base address of the segment, bypassing the TLB entirely.

Direct segments are a step in the right direction towards more flexible virtual memory: they eliminate most of the need to manage superpages, and they eliminate most of the address translation overhead of TLBs. However, they still have some limitations. First, the size of the segment must be decided before the application starts running. This is acceptable for certain applications that are configured by an administrator to use most of the physical memory in the server, but it may not be appropriate for all big-memory applications, especially those that will contend for resources with other memory-intensive applications on the same server, or for applications running in a virtualized guest OS where it is difficult to know how much physical memory is truly available. Second, direct segments are also limited to storing heap data; they do not currently work with `mmap`-ed files, which are another source of large virtual memory allocations and TLB misses for some applications (e.g., databases like MongoDB [23]).

Single address space systems. In systems such as Opal [13], all applications share a single vitual address space. Variable-width segments are the unit of virtual memory allocation and protection, while virtual addresses are translated to physical addresses at page granularity. It is possible to use a virtually-indexed, virtually-tagged cache, because virtual addresses have the same translation, independent of the process. Protection checks are process based and can be performed in parallel with cache lookups; these checks are performed using a Protection Lookaside Buffer [21], whose operation is similar to a typical TLB, or by using page-group hardware like that implemented in the HP PA-RISC architecture.

6. Conclusions

Technological advances have led to enormous changes in our systems and applications over the last 50 years. However, virtual memory mechanisms have evolved very little over that time. To a large extent, today's virtual memory architectures are still based on the assumptions of small physical memories and slow moving head disks.

We believe that a re-examination of virtual memory is warranted in the face of today's trends, such as enormous physical memories and rapidly evolving storage technologies (e.g., NVRAM and SSDs). As a basis for this reexamination, this paper measures the memory behavior of set of modern desktop, server, and data processing applications. Our measurements reveal the wide range of modern applications with respect to virtual memory usage, while our results show a number of opportunities for memory system optimization through both software and hardware means – e.g., to re-examine segment-based address translation with a decoupled finer-grained protection mechanism.

References

[1] K. Albayraktaroglu, A. Jaleel, Xue Wu, M. Franklin, B. Jacob, Chau-Wen Tseng, and D. Yeung. Biobench: A benchmark suite of bioinformatics applications. In *Proceedings of the IEEE International Symposium on*

Performance Analysis of Systems and Software, 2005, ISPASS '05, 2005.

[2] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Operating System Implications of Fast, Cheap, Non-Volatile Memory. In *HotOS*, 2011.

[3] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: skip, don't walk (the page table). In *ISCA*, pages 48–59. ACM, 2010.

[4] Thomas W. Barr, Alan L. Cox, and Scott Rixner. SpecTLB: a mechanism for speculative address translation. In *ISCA*, pages 307–318. ACM, 2011.

[5] Arkaprava Basu. *Revisiting Virtual Memory*. PhD thesis, University of Wisconsin - Madison, 2013.

[6] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *ISCA*, 2013.

[7] Ravi Bhargava, Ben Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *ASPLOS*, pages 26–35. ACM, 2008.

[8] Abhishek Bhattacharjee. Large-reach memory management unit caches. In *MICRO*, pages 383–394. ACM, 2013.

[9] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared last-level TLBs for chip multiprocessors. In *HPCA*, pages 62–63. IEEE Computer Society, 2011.

[10] Abhishek Bhattacharjee and Margaret Martonosi. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. In *PACT*, pages 29–40. IEEE Computer Society, 2009.

[11] Abhishek Bhattacharjee and Margaret Martonosi. Inter-core cooperative TLB for chip multiprocessors. In *ASPLOS*, pages 359–370. ACM, 2010.

[12] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[13] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.*, 12(4):271–307, 1994.

[14] Jonathan Corbet. Huge page issues. http://lwn.net/Articles/592011/, March 2014. Accessed: 2014-04-19.

[15] Jonathan Corbet. Postgresql pain points. https://lwn.net/Articles/591723/, March 2014. Accessed: 2014-04-21.

[16] Jonathan Corbet. Various page cache issues. http://lwn.net/Articles/591690/, March 2014. Accessed: 2014-04-21.

[17] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of apple desktop applications. *ACM Trans. Comput. Syst.*, 30(3):10, 2012.

[18] Linux Journal. Distributed caching with memcached. `http://www.linuxjournal.com/article/7451`.

[19] I. Kadayif, A. Sivasubramaniam, M. Kandemir, G. Kandiraju, and G. Chen. Generating physical addresses directly for saving instruction TLB energy. In *Proceedings of the 35th IEEE/ACM International Symposium on Microarchitecture (MICRO '02)*, Istanbul, Turkey, December 2002.

[20] Gokul B. Kandiraju and Anand Sivasubramaniam. Going the distance for TLB prefetching: An application-driven study. In *ISCA*. IEEE Computer Society, 2002.

[21] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architectural support for single address space operating systems. In *ASPLOS*, pages 175–186. ACM Press, 1992.

[22] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan W. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[23] MongoDB. FAQ: MongoDB storage. http://docs.mongodb.org/manual/faq/storage/. Accessed: 2014-04-19.

[24] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, USA, November 2002.

[25] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *SIGOPS Operating Systems Review*, 43(4), December 2009.

[26] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. Increasing TLB reach by exploiting clustering in page translations. In *International Symposium On High Performance Computer Architecture*, 2014.

[27] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. Colt: Coalesced large-reach TLBs. In *MICRO*, pages 258–269. IEEE Computer Society, 2012.

[28] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. Phase-change random access memory: A scalable technology. *IBM J. Res. Dev.*, 52(4):465–479, July 2008.

[29] Redis. http://redis.io/.

[30] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. Recency-based TLB preloading. In *ISCA*, pages 117–127. IEEE Computer Society, 2000.

[31] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, Vienna, Austria, September 2007.

[32] Karin Strauss and Doug Burger. What the future holds for solid-state memory. *IEEE Computer Magazine*, January 2014.

[33] Indira Subramanian, Cliff Mather, Kurt Peterson, and Balakrishna Raghunath. Implementation of multiple pagesize support in HP-UX. In *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, USA, June 1998.

[34] Mark R. Swanson, Leigh Stoller, and John B. Carter. Increasing TLB reach using superpages backed by shadow memory. In *ISCA*, pages 204–213. IEEE Computer Society, 1998.

[35] Emmett Witchel, Josh Cates, and Krste Asanovic. Mondrian memory protection. In *ASPLOS*, pages 304–316. ACM Press, 2002.