Back to the Past: When Segmentation Is More Efficient Than Paging

Lauri P Laux Jr, Roberto A Hexsel

Departamento de Informática,
Universidade Federal do Paraná

lauripaulo@gmail.com,roberto@inf.ufpr.br

Abstract. *Virtual Memory was devised in a time of scarce resources. In the coming decade we expect to see physical memory systems populated with 2^{64} bytes of RAM, a fraction of which may be non-volatile. Demand paging is inefficient in such large memories because space (Page Tables) and time (Page Table walks) overheads are too high. We collected execution traces from six applications and characterized their virtual memory behavior with respect to miss rates in references to Translation Buffers (TLBs) and Segment Buffers (SBs). Our measurements indicate that the miss rates for SBs are 2-3 orders of magnitude smaller than for TLBs. In light of these results, we discuss some of the design implications of segmented systems and of SBs.*

1. Introduction

The first edition of "Computer Architecture, A Quantitative Approach" [HP90], stated, as a rule of thumb, that *the memory needed by the average program grows from 1/2 to 1 address bit per year*. The book was published in 1990, and around 1995 we saw the introduction of 64 bit microprocessors with memory buses that were 32-36 bits wide. A decade later, memory buses widened to 40 bits, and in the mid 2010's 50^{+} bit buses are available. If this trend continues, in the next decade we can expect to see processors which may reference the full 2^{64} byte address space, all of it populated with RAM, with possibly a large fraction comprised of non-volatile RAM.

The memory systems of the next decade will be rather different from those of the early 2000s, when Intel discontinued support for segmentation. We believe it might be convenient to reconsider the design of the virtual memory system, as the premises that held since the early 1960s are no longer valid. Some of the recent trends include (i) slow magnetic disks are being replaced by $\geqslant 100\times$ faster solid state disks (SSDs); (ii) installed RAM capacity is so large, and inexpensive, that swapping may become unnecessary; (iii) when the promise of non-volatile RAM is fulfilled, magnetic disks will become "second-class peripherals" and a complete redesign of file systems may be worthwhile, and this may entail the elimination of buffer caches, for instance [LH16].

Managing such gigantic address spaces with demand paging becomes rather expensive, both in terms of space – one huge page table per process – and time – paging traffic between the translation buffer and primary memory. For these large physical memories, we believe Multics style segmentation to be a better memory management model that demand paging [DD68].

This paper presents the characterization of the virtual memory behavior in terms of the miss rates when accessing the buffer that hold virtual-to-physical address mappings. We simulated translation buffers for demand paging and for segmented virtual memory. The simulated buffer organizations are not meant to represent what one may find in current products as we are interested in counting memory references rather than measuring the simulated performance of said buffers. As we are interested in characterizing memory behavior of systems which will be available ten years hence, we do not try to guess what will be the specific design parameters of the translation buffers, as the design space is rather large.

We performed measurements to assess the effectiveness of a segmented virtual memory system and compared that to a demand paging system, for translation buffers of similar complexity. Our results indicate that miss rates of the segmentation address translation buffer is 100 to 1000× smaller than its demand paging equivalent. This, plus the gains achievable in handling the smaller data structures, indicate that segmentation may be a worthy alternative to demand paging when one considers physical memories with 2^{64} bytes.

Traces were collected from the execution of six real programs, and with these traces we compare the performance of address translation with Translation Lookaside Buffers (TLB) for paging, and Segmentation Buffers (SB) for segmentation. Our tracing tool builds a Segment Table for the executing process and rewrites the linear addresses into segmentID-displacement tuples. These are then used to compute the TLB and SB hit rates. In this paper we consider single processor systems.

In the next section we define both Demand Paging and Segmented virtual memory systems. Section 3 describes our trace collection tools, and in Section 4 we present the simulation data collected. In Section 5 we compare the miss rates measured for TLBs and SBs; and in Section 6 we discuss system design issues in light of our results. Section 7 discusses related work, and Section 8 presents our conclusions.

2. Paging and Segmentation

The need for virtualizing the memory system arose in the mid 1950's as the programs plus their data were becoming larger than the physical memories then available. The earlier systems made use of segmentation because it was intuitive: as a program is logically divided into three segments, text (code), data, and stack, portions of the physical memory should be allocated to these segments [Den70]. Segmentation had a serious drawback as the expensive physical memory could be underutilized because of fragmentation. The holes between the segments residing in memory would be too small for any new segment, whereas the scattered free space could accommodate one or more segments. This was called "external fragmentation" because the available spaces were external to the segments.

The computer architecture group at Manchester University designed the first demand paging system [KELS62]. Memory was divided into small chunks (pages) and the capacity of physical memory was augmented by "virtual memory", obtained by a clever combination of software with the use of secondary memory. This way, each segment wasted, on average, half of a page (internal fragmentation), and the

pages would be allocated onto physical memory only if the dynamics of program execution so dictated. Primary memory held only a small-ish subset of the program's address spaces. Memory utilization was greatly improved at the cost of increased traffic between primary and secondary memory. This memory management model has been successfully employed for half a century.

The Multics [DD68, BCD72] operating system was a very ambitious design, which combined the logic simplicity of segmentation while each segment was paged in or out of primary memory on demand. This design was much too complex for the hardware available in the early 1970's.

Figure 1 shows a conceptual model of a modern memory management system [LH16]. Two address streams are generated by the processor, one from the program counter (PC) and one from the address generation unit (AGU). These two units generate virtual addresses v_i and v_d to reference instructions and data. For each stream there is a "translation buffer" (TB) between the processor and primary memory. These TBs are implemented with fast associative memory and hold a small subset of the process' memory map, which is represented by the function $f(\,)$ in the diagram. The primary memory is referenced with physical addresses p_i and p_d.

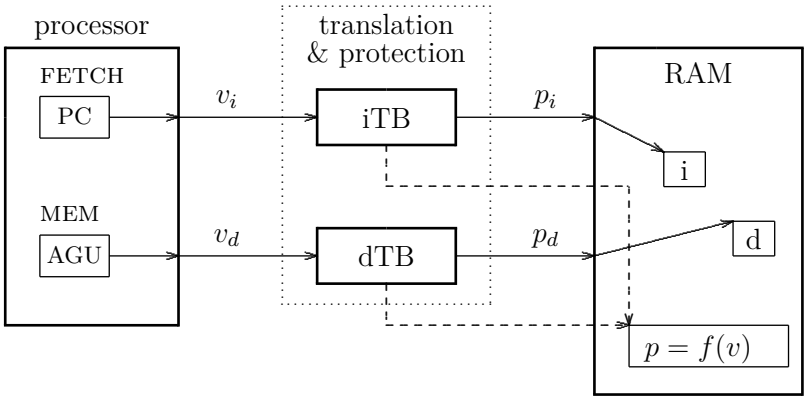

Figure 1. Conceptual model for the virtual memory system.

For a given process, its memory map holds all the valid associations between virtual and physical addresses. This map provides two important functions: (i) physical memory allocation which decouples virtual addresses from physical addresses, so that a process may use any free memory available; and (ii) security to stop a process from referencing addresses allocated to other processes. To improve performance, these two functions must be performed near the processor and the translations must be fast, hence the buffers are small.

When a process starts, its memory map is partially filled by the loader and a small subset of the instructions and data are loaded from secondary memory. As the execution progresses, a different subset of the address space may have to be loaded, and the memory map is updated accordingly. The domain of the mapping $f(\,)$ is the whole address space, typically 2^{64} bytes in current processors, whereas the image of $f(\,)$ is some subset of the physical address space. The main function of virtual memory is to provide the programmer with access to, essentially, infinite memory – 2^{64} is a rather large number.

2.1. Paging

The conceptual model in Figure 1 is valid for both, segmentation and demand paging. For paging, the chunks of memory are called pages, and the most popular page size is 4K bytes. Some systems provide super-pages. with sizes ranging from 64K to 1G bytes depending on processor model and operating system design. The mapping function is called "Page Table" (PT), and the size of its domain is, for pages with 2^n bytes, $|f(\,)| = 2^{64}/2^n$. For 4K bytes, the Page Table holds up to 2^{52} elements; for 1G byte pages, the domain is a hefty 2^{34} elements. As each PT element is encoded in 8 or 16 bytes, these tables are very large. Clever hierarchical designs are used to reduce table size. Yet for large processes such as database managers, the Page Tables are proportionally large. At least some subset of the PT must always be kept in primary memory.

A virtual address is split into two fields, a virtual page number (VPN) and a displacement within the page. When the processor issues a reference, the TLB is searched for the corresponding VPN. If the translation is found (a TLB hit), the reference proceeds towards the primary memory (RAM), whence it then completes. If the address mapping is not found in the TLB (a TLB miss), then the Page Table is indexed with the VPN, and if the mapping is valid, the pair $\mathrm{VPN} \mapsto \mathrm{PPN}$ is loaded onto the TLB, possibly evicting some other mapping. These actions may be performed by a state machine – in Intel x86 processors – or by a short sequence of instructions – in MIPS processors.

If the mapping on the PT is not valid, it may be the case that the page is not resident in memory and has to be fetched from secondary memory (a page fault), or that the reference is to a non-mapped region of the address space (a segmentation fault), or the reference is deemed illegal (a protection fault). In any case, the operating system (OS) must take over and perform a series of operations to either recover from the fault, or to terminate the process.

2.2. Segmentation

As a minimum, a process comprises three logical segments: text, data and stack. Usually, the size of the text segment is fixed at compilation, whereas the size of the data and stack segments may vary during execution. In addition, each library dynamically linked to the process contributes with text and data segments which must be included in its address space.

A Segment Table (ST) is much smaller than a Page Table. The number of segments in a process is on the order of 100 to 1000. Even if the operating system considers open files as process segments, as in Multics, the ST would need some 2000 elements. Managing these smaller tables is far more efficient than the gargantuan Page Tables.

A virtual address is split into two fields, a virtual segment number (VSN) and a displacement into that segment. Segments can be of arbitrary sizes, hence both the VSN and the displacement are of arbitrary width. Usually, a certain number of the most significant address bits are reserved for indexing the Segment Table (or the segment buffer). The actions needed to translate a virtual address into the corresponding physical address are similar to those with paging. If the

mapping is found in the SB (a SB hit), the translated address is forwarded to the primary memory. If there is a SB miss, the segment table is searched for the physical segment number (PSN), and the mapping VSN $\mapsto$ PSN is loaded onto the SB, possibly evicting some other mapping.

If the mapping in the segment table is invalid, the three possibilities described above also apply: it may be a segment fault and the missing segment must be loaded from secondary memory, it can be a protection fault or a reference to an unmapped address. In any of these cases, the OS must take over to rectify the problem.

3. Trace Collection

We collected the execution traces on an Intel x86-64 processor, running Linux. The programs were instrumented with Valgrind 3.1.2 [NS07], and the traces generated with the Lackey tool (a Valgrind component). We wrote a program, called TraceTool, to read Lackey's output and store it in compressed files. Only the execution of the 'main' process is traced; the main process is the one that performs the bulk of the work for an application. Some of the applications are invoked by a shell script which sets up the environment, then starts the main process. If the main process spawns other processes, the child processes are not traced because their address spaces are disjoint from the main process'.

Our traces contain the addresses of all references to data, plus the addresses of the load-store instructions. The traces contain enough information to reproduce the memory behavior of the programs.

When the execution starts, most of the logical segments are mapped by the loader onto the process' address space: four segments for the program (text, initialized data, uninitialized data ("block started by symbol", or BSS) and the stack, plus three more segments for each library linked into the process (text, data and BSS). As the execution progresses, more segments may be added when the process requests heap space, usually via a call to `malloc`. To capture this behavior, we wrote a Python script that periodically scans the address space and records 'new' segments. The script makes use of `pmap` and takes a new snapshot every 100ms. Trace collection can last for several hours, and taking 10 samples per second seems to be a good compromise between precision and reduced interference on the process being traced. When the Lackey process ends, the memory map is complete.

The original x86-64 traces are suitable for simulating the linear (flat) address space of paged memory systems. To generate the two-dimensional addresses for a hypothetical segmented machine, we use the process memory map to create a Segment Table (ST). Each element of the Segment Table records the segment's base address (BaseAddr), top address (TopAddr), the segment size, access rights, and a string with the segment name. Table 1 shows four lines of an ST. As a new segment is added to the ST, if the topmost 16 address bits of the segment starting address are all zero then it receives the next available segment number (VSN). If the topmost 16 address bits are non-zero, the VSN becomes the value of the topmost 16 bits. In the table, the digits are grouped into quartets to improve readability.

Once the ST is populated, the two-dimensional addresses can be generated for a linear address (LA). To find the segment to which a linear address belongs to,

Table 1. Segment table (fragment).

VSN	BaseAddr	TopAddr	Size	Rights	Name
0x0000	0x0040.0000	0x0061.0fff	2M	r-x--	lsblk
0x0001	0x0061.1000	0x0061.1fff	4K	r----	lsblk
0x0002	0x0061.2000	0x03ff.ffff	58M	rw---	lsblk
0x0003	0x0400.0000	0x0402.5fff	152K	r-x--	ld-2.23.so

the address is compared to the starting address and size of each ST element. If the address is in range of a segment (BaseAddr $\leqslant$ LA $\leqslant$ TopAddr), then the VSN replaces the topmost 16 address bits of the linear address, thus generating a segmented address. For instance, if the linear address is `0x0061.10ff`, then it belongs in segment `0x0001` and the segmented address is `0x`*`0001`*`.0000.0061.10ff`.

4. Reference Counts

We collected traces from six real applications. In what follows we briefly describe the application, the data sets, and the reference counts obtained from the traces. The programs were simulated for 10 billion instructions, and each simulation yields traces with roughly 13-15 million memory references.

MySQL SysBench [Ubu17] performs random operations on a table with 1,000,000 lines, implemented as an MySQL database. The operations are distributed across eight internal MySQL threads, and run during 480 seconds. Each thread's stack is held in a separate segment, and there is an anonymous segment below the bottom of the stack to catch references outside of the allocated area.

Firefox The web browser was traced while opening a dynamic web page pointed to by a command line argument. Most of the segments mapped are dynamic libraries but just a few of them are used to display the (relatively simple) web page. There are 91 segments labeled as 'cache', with sizes ranging from 1 to 256 pages.

Python Python was traced executing `pip`, to fetch and install tools used in the development of Python programs.

Tomcat Tomcat is a Java Servlet Container and web server. The server was traced while serving Tomcat's initial web page, then shutting down. There are 319 segments labeled as `anon`.

QEMU + FreeDOS QEMU is a hardware virtualization platform. The traces were obtained while emulating an Intel x86 processor, with 32 Mbytes of RAM, hard disk, and text console. FreeDOS was run on this virtual machine from the boot sequence until the first prompt. At the prompt, FreeDOS was shutdown from the text console. There are 242 segments labeled as `anon`.

LibreOffice The trace records the `writer` application reading a 314 Kbytes Microsoft Word document. There are 91 segments labeled as 'cache', with sizes from 1 to 256 pages, and 163 anonymous segments.

Table 2 shows the number of libraries linked into the application, the segments mapped, and the segments actually referenced. Many segments, mostly from libraries, are mapped but never referenced, as the libraries' functions are not invoked.

Table 2. Libraries, number of segments, usage ratio.

application	libs	segs	used	ratio
MySQL	75	310	151	48%
Firefox	485	979	633	64%
Python	111	162	115	71%
Tomcat	90	416	272	65%
QEMU	601	845	567	67%
LibreOffice	785	1,085	830	76%

All libraries, with two exceptions, map a 2 Mbytes segment with no '`r,w,x`' permissions, between the executable and the read-only segment. This buffer zone is used to catch any references to addresses below the segment with initialized variables, which is copy-on-write and remains untouched unless its contents are changed by the process. The two exceptions are the loader `ld-2.23.so`, and the locale's tables `libc.mo` – these libraries do not make use of the 2 Mbyte buffers.

Memory allocated as heap is mapped on segments labeled `anon`, and have `r,w` permissions. Some `anon` segments have no permission bits set, and probably are heap regions that were allocated but never touched during execution – the traces show no references to these segments.

Figure 2 shows a plot of segment frequency versus segment size for the applications, and the average for the six programs. The most popular segment is a single 4 Kbytes page; the second most popular size is 2 Mbytes. Segment sizes were rounded up to the nearest power of two. The numbers shown are for allocated segments, and not for segments actually referenced – Table 2 indicates that some 20 to 50% of the segments are allocated yet never touched during execution.

5. A Comparison of Paging and Segmentation

Three TLB/SB configurations were simulated, with 32, 64 and 128 entries. All buffers are fully associative and were simulated with perfect least-recently used (LRU) replacement. Except for the LRU replacement, the buffers are of a complexity similar to those available on current x86-64 processors. By 'similar' we mean the simulated buffers are of similar capacity to the last level TLB of x86-64 processors. As we are merely counting references, it is unnecessary to simulate the rather complex TLB hierarchies of existing processors.

Table 3 displays the number of references on each trace (refs). The columns under the heading TLB show the miss rates for the three TLB organizations, while those under the heading SB show the miss rates for the three SBs. Some of the applications encounter a number of misses that are less than one in 10,000 references – typically, these are buffers that accommodate all the active segments referenced during execution, and the few misses observed are compulsory misses.

The miss rates shown in Table 3 are not surprising. For the demand paging systems, the 32 entry TLB covers just $32 \times 4096 = 128$ Kbytes, and the 128 entry covers 512 Kbytes. Given that the applications have working sets larger than the TLB reach, high miss rates are to be expected.

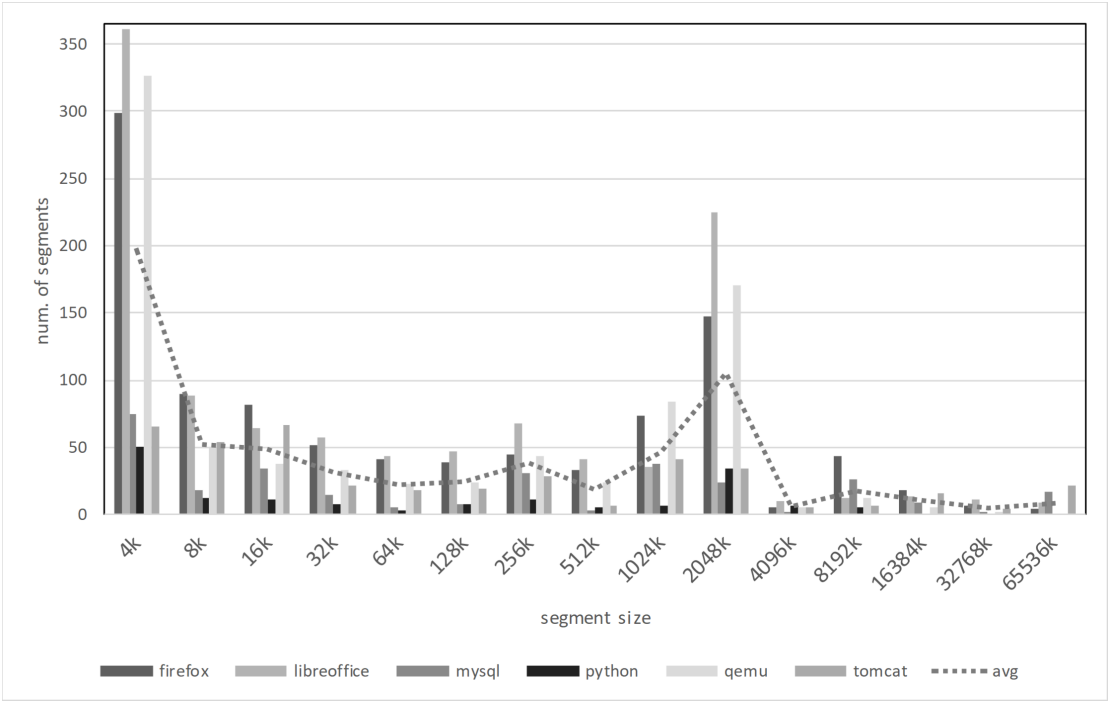

Figure 2. Segment size distribution.

Suppose that a RAM reference takes 200 cycles (this is optimistic), that the PT is organized in four levels (realistic), and that the PT lines are never present in the L2 or L3 TLBs (pessimistic). Under these assumptions, each TLB miss costs some 800 cycles. Hornyack, Basu, *et al.* [HCG+15, BGC+13] present measurements for the percentage of execution cycles spent walking the Page Table on TLB misses for large scale applications. They found that the applications spend from 2 to 10% of the execution cycles filling the TLB from the PT. One graph application with bad locality spends over 50% of its cycles on PT walks to refill the TLB.

The miss rates for the segmented memory are 100–1000× smaller than those for TLBs of similar organization, and this is also not surprising. As the number of segments is much smaller than the number of virtual pages, the miss ratios are bound to be better. Furthermore, all processes have at least one segment of 8–128 Mbytes that holds a large portion of the actively referenced addresses.

Table 3. TLB and SB reference counts and miss rates (#refs×10^6).

		TLB			SB		
applic.	refs	32	64	128	32	64	128
MySQL	15,272	0.718	0.152	0.058	0.0033	0.0006	$1 \cdot 10^{-8}$
Firefox	14,649	0,354	0.137	0.053	0.0251	0.0178	0.0129
Python	14,499	0.855	0.193	0.032	0.0001	$2 \cdot 10^{-8}$	$8 \cdot 10^{-9}$
Tomcat	14,512	0.258	0.066	0.018	0.0011	$3 \cdot 10^{-7}$	$3 \cdot 10^{-8}$
QEMU	13,618	0.278	0.036	0.004	0.0146	0.0137	0.0033
LibreOf	13,509	0.216	0.073	0.029	0.0529	0.0353	0.0267

For some applications, such as Firefox, LibreOffice and QEMU, dozens of libraries are linked into the address map yet their segments are never referenced, thus improving locality, hence yielding better miss ratios. In a segmented system, these libraries would be mapped on the Segment Table but allocation of physical memory would be postponed until the segment is referenced, further improving performance.

6. Design Considerations

The diagram in Figure 3 shows a 'traditional' organization for a segment buffer [LH16]. The virtual address is split into the virtual segment number (VSN) and a displacement. The VSN is associatively compared to the virtual tags in the SB; in case of a hit, the displacement is added to the physical address of the segment base, and the physical address is posted to memory. This completes the translation phase.

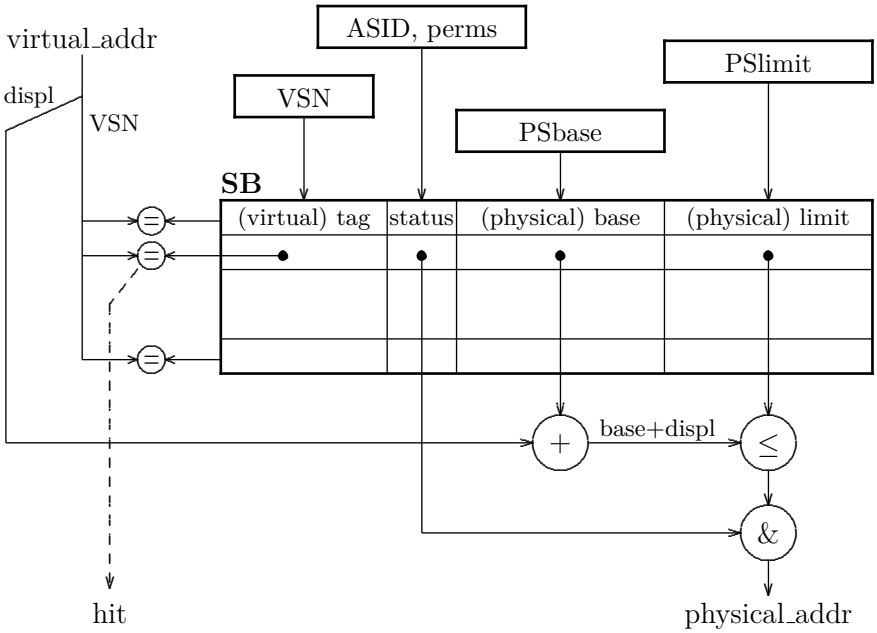

Figure 3. Segment Buffer.

The resulting physical address is then compared to the segment limit; if the comparison results in a segment violation, the exception is signaled to the processor. Some form of process, or address space, identification is kept in the ASID field, and the ASID is also checked for illegal references. This completes the reference validation phase, or the protection phase.

Notice that all addresses are 64 bits wide, hence the programming model does not need to change and the segmented memory is hidden from the programmer, as it should [LH16]. The management of logical segments is performed by the linker, the segment table is setup by the loader and is maintained by the OS. To improve physical memory utilization, the compiler could allow the programmer to give an estimate for segment size, or to declare a segment as 'elastic' if it is known that it can grow or shrink during execution.

The address translation ($\text{VA} \mapsto \text{PA}$) must be performed as early as possible so the memory reference may complete without delay. With demand paging, the translation is obtained by simply indexing the TLB, whereas with segmentation two additions must be performed, the first to compute the PA (PSbase + displ), and the second to compare the PA to the segment limit. The first addition must be performed early and the adder is on the critical path. As a first approximation, the average memory access time could be elongated by 1–2% without significant performance loss when compared to a paging system. Recall that TLB-page table walks may cost 2-10% additional execution cycles, according to Hornyack's measurements.

Any additional delay introduced by translation may be compensated by the reduction in traffic at the processor-RAM interface: as the SB miss ratios are 100-1000× smaller, there is less demand for RAM bandwidth. Also, the smaller segment tables incur in less management overhead, which translates into thousands of instructions, and secondary memory accesses, *not* performed to manage a hierarchical memory mapping table.

The adder that compares the PA to the segment limit can be taken off the critical path because addressing/protection exceptions do not need to be signaled until the instruction is about to commit, and that is late in the pipeline. The same applies to access permissions.

There is one additional condition that must be verified. If the programming model is to remain unchanged, then the the PA must also be checked against the segment base. A programming error (or malice) could generate a 'negative' VA, which when added to the segment base could result in a PA that is smaller than the physical segment base, thus generating an addressing fault. This comparison can also be performed late in the pipeline and kept off the critical path. The third adder is not shown in Figure 3.

External fragmentation may be a problem, albeit not so serious because memory is 'infinite'. Trading physical memory utilization for shorter execution time may be worthwhile for time-critical applications. Internal fragmentation is a problem in systems that use super-pages as their sizes are either too large for the smallish segments (libraries), or too small for the large heap areas or in-memory tables [HCG+15]. In systems which support long running applications, the cost of relocating segments in order to free physical memory may be a small fraction of the total execution time. There is a relatively inexpensive solution to the relocation dilemma: a DMA-like engine can steal bus cycles to move inactive segments onto more crowded regions in physical memory, thus freeing unused space [LH16].

7. Related Work

Single address space systems [KCE92] are an alternative to segmentation: all processes are mapped on a single virtual address space. If the physical memory is large enough, address translation may be unnecessary. Protection is enforced at the level of logical segments, or with paging. In either case, a structure similar to a TLB or an SB must be used to capture the illegal references.

Basu *et al.* [BGC+13] introduce the "direct segment" to support *one* arbitrarily large segment while retaining demand paging for the rest of the memory

space. Their solution is justified by the way large-scale applications use memory: one very large chunk of memory is allocated at initialization, then the application itself manages the allocation of smaller chunks for buffers, tables, etc. The direct segments imply in changes the programming model, by adding one programmer visible segment, yet is less disruptive than variable size super-pages, from the micro-architecture point of view.

Pham *et al.* [PBEL14] state that applications typically exhibit 'contiguous' spatial locality in which tens of consecutive virtual pages are mapped to consecutive physical pages. This behavior generates many instances of contiguous page table entries (PTE), though typically not enough for large page generation. Pham proposes a low-overhead, multi-granular TLB organization that exploits PTE clustering on a hierarchical TLB architecture with enhancements regarding replacement policies and prefetching to eliminate 46% of L2 TLB misses. Our results show that segmentation may provide the same benefits in exploiting spatial locality, with less complex hardware and software implementation.

Hornyack *et al.* [HCG+15] present strong evidence that several large-scale applications would perform better if the memory allocation were segment-based instead of the current paging systems. These applications spend a great deal of time handling TLB misses and the performance loss ranges from a few percent of the execution cycles to 58% for some workloads with poor locality. Hornyack also shows that server-class memory-hungry applications use "virtual memory areas" which represent an item of code (or data) in a contiguous region of memory that spans from one to several contiguous virtual pages. They claim that, rather than using fixed size pages, hence large page tables, a segmented system would substantially reduce the amount of state needed to keep the protection and mapping information. While our results are in agreement with their findings, we extend Hornyack's work by characterizing program behavior in terms of segmentation. Our work, and theirs likewise, indicate that segmentation can bring substantial performance gains in current systems, and specially for the systems of the coming decade.

Virtual memory was devised to hide the idiosyncrasies of primary and secondary memories from the programmer, by providing a flat address space, and a programming model that abstracted away all details regarding the timing of memory references, and the capacity of the installed memory. Large pages are an interesting idea but represent an unwelcome break with the original, neat and tidy, programming model provided by virtual memory. Large pages expose the physical memory to the programmer and introduce more complexity to a rather intricate component of the operating system.

Virtual memory systems were devised in an era of scarcity, and have served us well since the late 1960s. Petabyte physical memories are becoming affordable and the first products supporting non-volatile memory (NVRAM) are reaching the market. In [LH16] we examine several changes to the way operating systems might be designed and built, when one considers the effects of solid state disks (SSDs) and NVRAM on system performance and complexity.

8. Conclusion

In the next decade we may see systems equipped with 2^{64} bytes of physical memory, possibly with NVRAM comprising a large fraction of the primary memory. We believe serious consideration should be given to a complete re-design of our long serving and trusted demand paging virtual memory systems. To support such a bold claim, we characterized the behavior of real applications with respect to the virtual-to-physical translation buffers. We performed trace driven simulations of translation buffers of similar complexity for demand paging systems (TLBs) and segmented systems (SBs).

Our results indicate that the miss rates for SBs are two to three orders of magnitude smaller than for TLBs. This is not surprising since the number of logical segments is much less than the number of logical pages for any non-trivial application. The management of the smaller segment tables is bound to be less expensive than the huge page tables, yielding further improvements in performance. We show how the SBs can be designed so the segmented virtual memory remains hidden from the programmer and compiler writer.

Hard data notwithstanding, people in our community have grown so accustomed to demand paging that the simpler and more efficient alternative, segmentation, is commonly discarded offhand.

References

[BCD72] A Bensoussan, C T Clingen, and R C Daley. The Multics virtual memory: Concepts and design. *Comm of the ACM*, 15(5):308–318, May 1972.

[BGC+13] A Basu, J Gandhi, J Chang, M D Hill, and M M Swift. Efficient virtual memory for big memory servers. In *ISCA'13: 40th Intl Symp on Computer Arch*, pages 237–248, 2013.

[DD68] Robert C Daley and Jack B Dennis. Virtual memory, processes, and sharing in MULTICS. *Comm of the ACM*, 11(5):306–312, May 1968.

[Den70] Peter J Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, Sep 1970.

[HCG+15] P Hornyack, L Ceze, S Gribble, D Ports, and H M Levy. A study of virtual memory usage and implications for large memory. In *Proc Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, 2015.

[HP90] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1st edition, 1990. ISBN 1558600698.

[JNW08] B L Jacob, S W Ng, and D T Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008. ISBN 0123797513.

[KCE92] E J Koldinger, J S Chase, and S J Eggers. Architecture support for single address space operating systems. In *ASPLOS'92: 5th Intl Conf on Arch Support for Progr Lang and Oper Sys*, pages 175–186, 1992.

[KELS62] T Kilburn, D B G Edwards, M J Lanigan, and F H Sumner. One-level storage system. In *IRE Trans on Electronic Computers*, EC-11, pages 223–235, 1962.

[LH16] Lauri P Laux Jr and R A Hexsel. Back to the past: Segmentation with infinite and non-volatile memory. In *WSCAD-SSC'16: XVII Workshop em Sistemas Computacionais de Alto Desempenho*, pages 278–289, 2016.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI'07: Proc 28th ACM SIGPLAN Conf on Programming Language Design and Implementation*, pages 89–100, 2007.

[PBEL14] B Pham, A Bhattacharjee, Y Eckert, and G H Loh. Increasing TLB reach by exploiting clustering in page translations. In *HPCA'14: 20th Int Symp on High-Performance Comp Arch*, pages 558–567, 2014.

[Ubu17] Ubuntu. SysBench – a modular, cross-platform and multi-threaded benchmark tool, Aug 2017.