Software-Managed Address Translation

Bruce Jacob and Trevor Mudge
Advanced Computer Architecture Lab
EECS Department, University of Michigan
{blj,tnm}@eecs.umich.edu

Abstract

In this paper we explore software-managed address translation. The purpose of the study is to specify the memory management design for a high clock-rate PowerPC implementation in which a simple design is a prerequisite for a fast clock and a short design cycle. We show that software-managed address translation is just as efficient as hardware-managed address translation, and it is much more flexible. Operating systems such as OSF/1 and Mach charge between 0.10 and 0.28 cycles per instruction (CPI) for address translation using dedicated memory-management hardware. Software-managed translation requires 0.05 CPI. Mechanisms to support such features as shared memory, superpages, sub-page protection, and sparse address spaces can be defined completely in software, allowing much more flexibility than in hardware-defined mechanisms.

1 Introduction

In many commercial architectures the hardware support for memory management is unnecessarily complicated, places constraints on the operating system, and often frustrates porting efforts [37]. For example, the Intel *Pentium Processor User's Manual* devotes 100 of its 700+ pages to memory-management structures [31], most of which exist for backward compatibility and are unused by today's system software. Typical virtual memory systems exact a run-time overhead of 5-10% [4, 9, 41, 47], an apparently acceptable cost that has changed little in ten years [14], despite significant changes in cache sizes and organizations. However, several recent studies have found that the handling overhead of memory management hardware can get as high as 50% of application execution time [1, 28, 44]. Taken together these trends beg the question, *is dedicated memory-management hardware buying us anything—do its benefits outweigh its overhead?*

This work was supported by Defense Advanced Research Projects Agency under DARPA/ARO Contract Number DAAH04-94-G-0327.

In this paper we demonstrate a memory management design that stays within an acceptable performance overhead and that does not require complex hardware. It places few constraints on the operating system but still provides all the features of systems with more hardware support. The design is *software-managed address translation*, or *softvm* for short. It dispenses with hardware such as the translation lookaside buffers (TLBs) found in every modern microarchitecture and the page-table-walking state machines found in x86 and PowerPC architectures. It uses a software-handled cache miss, as in the VMP multiprocessor [11, 12, 13], except that VMP used the mechanism to explore cache coherence in a multiprocessor, while we use it to simplify memory management hardware in a uniprocessor. It also resembles the in-cache address translation mechanism of SPUR [26, 43, 56] in its lack of TLBs, but takes the design one step further by eliminating table-walking hardware.

Software-managed address translation supports common operating systems features such as address space protection, fine-grained protection, sparse address spaces, and superpages. Compared to more orthodox designs, it reduces hardware complexity without requiring unduly complex software. It has two primary components: a virtually indexed, virtually tagged cache hierarchy with a writeback cache at the lowest level (L2, for example), and a software-managed cache miss at the lowest level. Virtual caches do not require address translation when requested data is found in the cache, and so obviate the need for a TLB. A miss in the L2 cache invokes the operating system's memory manager, allowing the operating system to implement any type of page table, protection scheme, or replacement policy, as well as a software-defined page size. The migration of address-translation support from hardware to software increases flexibility significantly.

We show the efficiency of software-managed address translation by analyzing a specific implementation, thereby finding an upper bound on overhead. The example adds software-managed translation to a conventional PowerPC memory management organization. It forms the basis of the memory management design of the PUMA processor, a high clock-rate 32-bit PowerPC in which a simple design is

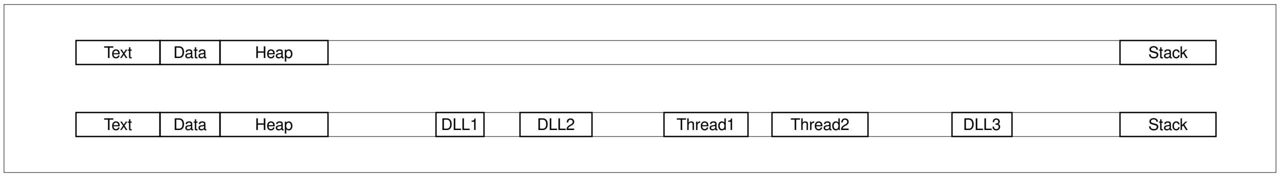

Figure 1: Sparse address spaces. The top address space is that of a traditional 4.3BSD process, with contiguous text, data, and heap segments and a continuous stack segment. The bottom address space contains modern features like dynamically loaded libraries and multiple threads of control, which leave holes within the address space, and thus would leave holes within a linear page table. A wired-down linear page table (as in 4.3BSD) would not be practical.

vital for a fast clock and a short design cycle.

The example implementation adds PowerPC segments [39] to the *softvm* design; these support address space protection, shared memory, and provide access to a large virtual address space. They are not an essential component of software-managed address translation—for example, they could be replaced by long address space identifiers or a 64-bit address space. However, the use of segments in conjunction with a virtual cache organization can solve the consistency problems associated with virtual caches.

2 Memory system requirements

There is a core set of functional mechanisms associated with memory management that computer users have come to expect. These are found in nearly every modern microarchitecture and operating system (e.g., UNIX [3], Windows NT [15], OS/2 [16], 4.3 BSD [34], DEC Alpha [17, 46], MIPS [23, 32], PA-RISC [25], PowerPC [29, 39], Pentium [31], and SPARC [52]), and include the following:

Address space protection. User-level applications should not have unrestricted access to the data of other applications or the operating system. A common hardware assist uses address space identifiers (ASIDs), which extend virtual addresses and distinguish them from addresses generated by different processes. Alternatively, protection can be provided by software means [5, 19, 50].

Shared memory. Shared memory allows multiple processes to reference the same physical data through (potentially) different virtual addresses. Space requirements can be reduced by sharing code between processes. Using shared memory for communication avoids the data-copying of traditional message-passing schemes. Since a system call is typically an order of magnitude faster than copying a page of data, many researchers have investigated zero-copy schemes, in which the operating system unmaps pages from the sender's address space and remaps them into the receiver's address space [18, 20, 35].

Large address spaces. Applications require increasingly large virtual spaces; industry has responded with 64-bit machines. However, a large address space does not imply a large address: large addresses are simply one way to implement large address spaces. Another is to provide each process a 4GB window into a larger global virtual address space, the approach used by the PA-RISC 1.X and 32-bit PowerPC architectures [25, 39].

Fine-grained protection. Fine-grained protection marks objects as read-only, read-write, execute-only, etc. The granularity is usually a page, though a larger or smaller granularity is sometimes desirable. Many systems have used protection to implement various memory-system support functions, from copy-on-write to garbage collection to distributed shared virtual memory [2].

Sparse address spaces. Dynamically loaded shared libraries and multithreaded processes are becoming commonplace, and these features require support for sparse address spaces. This simply means that holes are left in the address space between different objects to leave room for dynamic growth. In contrast, the 4.3BSD UNIX [34] address space was composed of two continuous regions, depicted in Fig 1. This arrangement allowed the user page tables to occupy minimal space, which was important because the original virtual memory design did not allow the page tables to be paged.

Superpages. Some structures must be mapped for virtual access, yet are very large. The numerous page table entries (PTEs) required to map them flood the TLB and crowd out other entries. Systems have addressed this problem with "blocks" or "superpages"—multiples of the page size mapped by a single TLB entry. For example, the Pentium and MIPS R4000 allow mappings for superpages to reside in the TLB alongside normal mappings, and the PowerPC defines a Block TLB to be accessed in parallel with the normal TLB. Several studies have shown significant performance gains for reducing the number of TLB entries to cover the current working set [33, 47, 49].

Direct memory access. Direct memory access (DMA) allows asynchronous copying of data from I/O devices directly to main memory. It is difficult to implement with virtual caches, as the I/O space is usually physically mapped. The I/O controller has no access to the virtual-physical mappings, and so cannot tell when a transaction should first invalidate data in the processor cache. A simple solution performs DMA transfers only to uncached physical memory, but this could reduce performance by requiring the processor to go to main memory too often.

3 Background and previous work

Address translation is the mechanism by which the operating system provides virtual address spaces to user-level applications. The operating system maintains a set of mappings from per-process virtual spaces to the system's physical memory. Addresses are usually mapped at a *page* granularity—typically several kilobytes. The mappings are organized in a *page table*, and for performance reasons most hardware systems provide a *translation lookaside buffer* (TLB) that caches parts of the page table. When a process performs a load or store to a virtual address, the hardware translates this to a physical address using the mapping information in the TLB. If the mapping is not found in the TLB, it must be retrieved from the page table and loaded into the TLB before processing can continue.

3.1 Problems with virtual caches

Virtual caches complicate support for virtual-address aliasing and protection-bit modification. Aliasing can give rise to the *synonym problem* when memory is shared at different virtual addresses [22], and this has been shown to cause significant overhead [54]; protection-bit modification is used to implement such features as copy-on-write [1, 42], and can also cause significant overhead when used frequently.

The synonym problem has been solved in hardware using schemes such as dual tag sets [22] or back-pointers [51], but these require complex control logic that can impede high clock rates. Synonyms can be avoided by setting policy in the operating system—for example, OS/2 requires all shared segments to be located at identical virtual addresses in all processes so that processes use the same address for the same data [16]. SunOS requires shared pages to be aligned in virtual space on extremely large boundaries (at least the size of the largest cache) so that aliases will map to the same cache line [10, 24][1]. Single address space operating systems such as Opal [7, 8] or Psyche [45] solve the problem by eliminating the need for virtual-address aliasing entirely. In a single address space all shared data is referenced through global addresses; as in OS/2, this allows pointers to be shared freely across process boundaries.

Protection-bit modification in virtual caches can also be problematic. A virtual cache allows one to "lazily" access the TLB only on a cache miss; if so, protection bits must be stored with each cache line or in an associated page-protection structure accessed every cycle, or else protection is ignored. When one replicates protection bits for a page across several cache lines, changing the page's protection can be costly. Obvious but expensive solutions include flushing the entire cache or sweeping through the entire cache and modifying the affected lines.

1. Note that the SunOS scheme only solves the problem for direct-mapped virtual caches or set-associative virtual caches with physical tags; shared data can still exist in two different blocks of the same set in an associative, virtually-indexed, virtually-tagged cache.

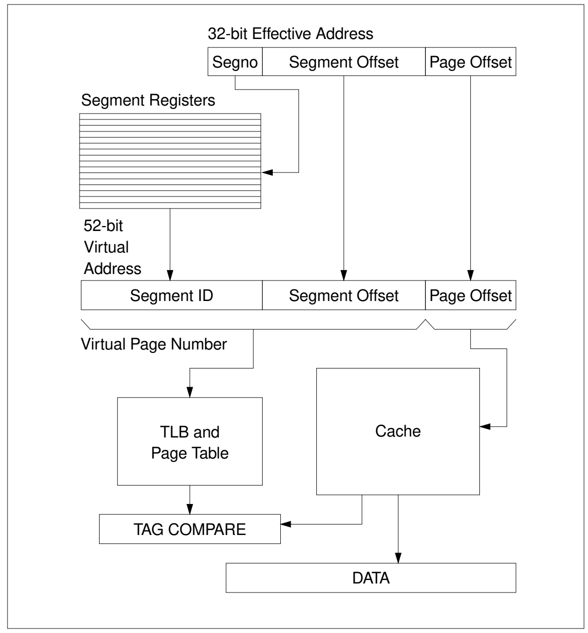

Figure 2: PowerPC segmented address translation. Processes generate 32-bit effective addresses that are mapped onto a 52-bit address space via sixteen segment registers, using the top four bits of the effective address as an index. It is this extended virtual address that is mapped by the TLB and page table. The segments provide address space protection and can be used for shared memory.

3.2 Segmented translation

The IBM 801 introduced a segmented design that persisted through the POWER and PowerPC architectures [6, 29, 39, 53]; it is illustrated in Fig 2. Applications generate 32-bit "effective" addresses that are mapped onto a larger "virtual" address space at the granularity of *segments*, 256MB virtual regions. Sixteen segments comprise an application's address space. The top four bits of the effective address select a segment identifier from a set of 16 registers. This segment ID is concatenated with the bottom 28 bits of the effective address to form an extended virtual address. This extended address is used in the TLB and page table. The operating system performs data movement and relocation at the granularity of pages, not segments.

The architecture does not use explicit address space identifiers; the segment registers ensure address space protection. If two processes duplicate an identifier in their segment registers they share that virtual segment by definition; similarly, protection is guaranteed if identifiers are *not* duplicated. If memory is shared through global addresses, no aliasing (and therefore no virtual-cache synonyms) can occur and the TLB and cache need not be flushed on context switch[2]. This solution to the virtual cache synonym problem is similar to that of single address space operating systems—global addresses cause no synonym problems.

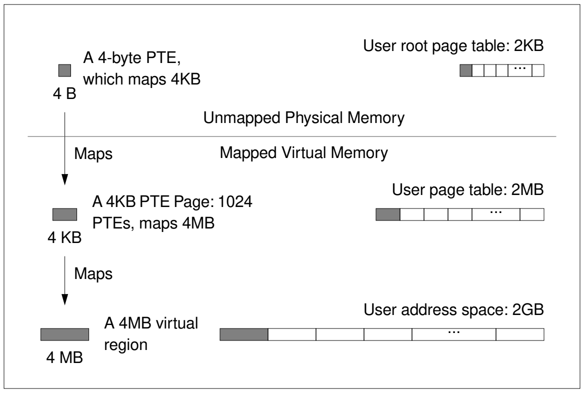

Figure 3: The MIPS 32-bit hierarchical page table. MIPS hardware provides support for a 2MB linear virtual page table that maps the 2GB user address space by constructing a virtual address from a faulting virtual address that indexes the mapping PTE in the user page table. This 2MB page table can easily be mapped by a 2KB user root page table.

3.3 MIPS: A simple 32-bit page table design

MIPS [23, 32] eliminated the page-table-walking hardware found in traditional memory management units, and in doing so demonstrated that software can table-walk with reasonable efficiency. It also presented a simple hierarchical page table design, shown in Fig 3. On a TLB miss, the hardware creates a virtual address for the mapping PTE in the user page table. The virtual page number (VPN) of the address that missed the TLB is used as an index into the user page table, which must be aligned on a 2MB virtual boundary. The base pointer, called *PTEBase*, is stored in a hardware register and is usually changed on context switch. This is illustrated as part of Fig 4. The advantage of this page table organization is that a small amount of wired-down memory (2KB) can map an entire user address space efficiently; in the worst case, a user reference will require two additional memory lookups: one for the root-level PTE, one for the user-level PTE. The TLB miss handler is very efficient in the number of instructions it requires: the handler is less than ten instructions long, including the PTE load. We base our page table and cache miss examples on this scheme for simplicity and clarity; however, any other organization could be used as well.

3.4 SPUR: In-cache address translation

SPUR [26, 43, 55, 56] demonstrated that the storage slots of the TLB are not a necessary component in address translation. The architecture uses a virtually indexed, virtually tagged cache to delay the need for address translation until a cache miss occurs. On a miss, a hardware state machine generates the virtual address for the mapping PTE and searches the cache for that address. If this lookup misses, the state machine continues until the topmost level of the page table is reached, at which point the hardware requests the root PTE (at a known address) from physical memory.

The SPUR design eliminated specialized, dedicated hardware to store mapping information. However, it replaced the TLB with another specialized hardware translation mechanism—a finite state machine that searched for PTEs in general-purpose storage (the cache) instead of special-purpose storage (TLB slots).

3.5 VMP: Software-controlled caches

The VMP multiprocessor [11, 12, 13] places virtual caches under software control. Each processor node contains several hardware structures, including a central processing unit, a software-controlled virtual cache, a cache controller, and special memory. Objects the system cannot afford to have causing faults, such as root page tables and fault-handling code, are kept in a separate area called *local memory,* distinguished by the high-order bits of the virtual address. Code in local memory controls the caches; a cache miss invokes a fault handler that locates the requested data, possibly causes other caches on the bus to invalidate their copies, and loads the cache.

The scheme reduces the amount of specialized hardware in the system, including memory management unit and cache miss handler, and it simplifies the cache controller hardware. However, the design relies upon special memory that lies in a completely separate namespace from the rest of main memory.

4 Software-managed address translation

The *softvm* design requires a virtual cache hierarchy. There is no TLB, no translation hardware. When a reference fails to hit in the bottommost virtual cache a CACHEMISS exception is raised. We will refer to the address that fails to hit in the lowest-level cache as the *failing address*, and to the data it references as the *failing data*.

This general design is based on two observations. The first is that most high performance systems have reasonably large L2 caches, from 256KB found in many PCs to several megabytes found in workstations. Large caches have low miss rates; were these caches virtual, the systems could sustain long periods requiring no address translation at all. The second observation is that the minimum hardware necessary for efficient virtual memory is a software-managed cache miss at the lowest level of a virtual cache hierarchy. If software resolves cache misses, the operating system is free to implement whatever virtual-to-physical mapping it chooses. Wood demonstrated that with a reasonably large cache (128KB+) the elimination of a TLB is practical [55]. For the cache sizes we are considering, we reach the same conclusion (see the *Discussion* section for details).

2. Flushing is avoided until the system runs out of identifiers and must reuse them. For example, the address space identifiers on the MIPS R3000 and Alpha 21064 are six bits wide, with a maximum of 64 active processes [17, 32]. If more processes are desired, identifiers must be constantly reassigned, requiring TLB & virtual-cache flushes.

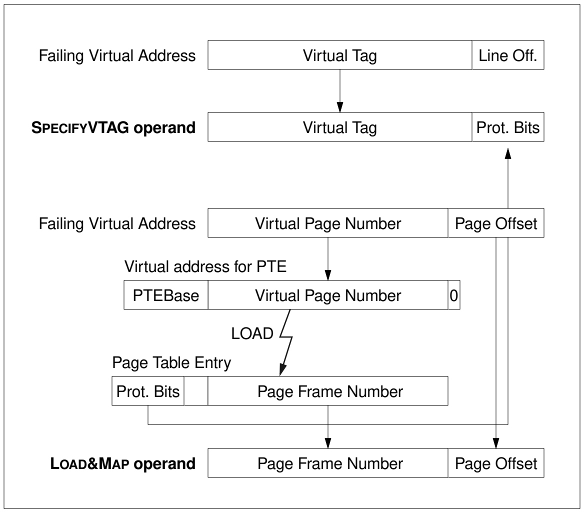

Figure 4: SPECIFYVTAG and LOAD&MAP. The top figure illustrates SPECIFYVTAG, the bottom figure illustrates LOAD&MAP. The Load&Map example assumes a MIPS-like page table.

4.1 Handling the CACHEMISS exception

On a CACHEMISS exception, the miss handler loads the data at the failing address on behalf of another thread. The operating system must therefore be able to load a datum using one address and place it in the cache tagged with a different address. It must also be able to reference memory virtually or physically, cached or uncached; to avoid causing a cache-miss exception, the cache-miss handler must execute using physical addresses. These may be cacheable, provided that a cacheable-physical address that misses the cache causes no exception, and that a portion of the virtual space can be directly mapped onto physical memory.

When a virtual address misses the cache, the failing data, once loaded, must be placed in the cache at an index derived from the failing address and tagged with the failing address's virtual tag, otherwise the original thread will not be able to reference its own data. We define a two-part load, in which the operating system first specifies a virtual tag and set of protection bits to apply to the incoming data, then loads the data with a physical address. The incoming data is inserted into the caches with the specified tag and protection information. This scheme requires two privileged instructions to be added to the instruction set architecture (ISA)[3]: SPECIFYVTAG and LOAD&MAP, depicted in Fig 4.

SPECIFYVTAG instructs the cache to insert future incoming data at a specific offset in the cache, tagged with a specific label. Its operand has two parts: the virtual tag (VTAG) comes from the failing virtual address; the protection bits come from the mapping PTE. The bottom half of the VTAG identifies a block within the cache, the top half is the tag.

3. Many ISAs leave room for such management instructions, e.g. the PowerPC ISA **mtspr** and **mfspr** instructions (move to/from special purpose register) would allow implementations of both functions.

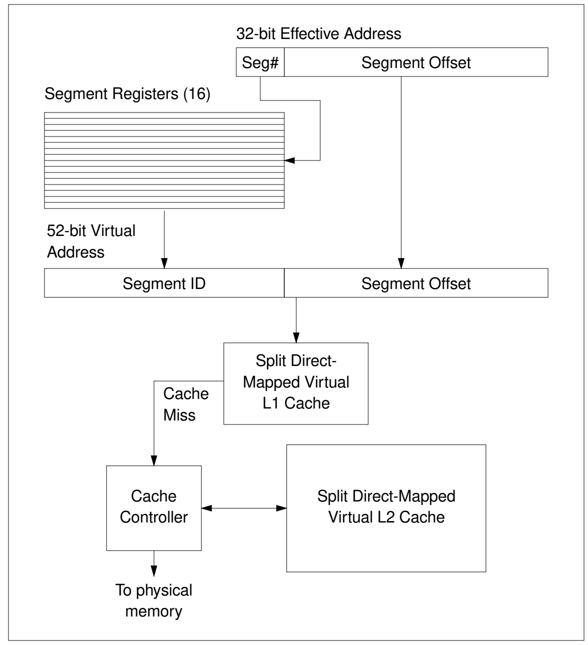

Figure 5: The example mechanism. Segmentation extends a 32-bit user address into a 52-bit global address. The top 20 bits of the global address determine if the address is physical and/or cacheable.

Note that the VTAG is larger than the virtual page number; the hardware should not assume *any* overlap between virtual and physical addresses beyond the cache line offset. This is essential to allow a software-defined page size.

The operand of a LOAD&MAP is a physical or virtual address. The datum identified by the operand is loaded from the cache or memory and then (re-) inserted into the cache at the cache block determined by the previously executed SPECIFYVTAG, and tagged with the specified virtual tag. Thus an operating system can translate data that misses the cache, load it from memory (or even another location in the cache), and place it in any cache block, tagged with any value. When the original thread is restarted, its data is in the cache at the correct line, with the correct tag. Note the operations can be performed out of order for performance reasons, as long as the tag arrives at the cache/s before the data arrives. Note also that without hardware support, the two-part load must not be interrupted by another two-part load.

4.2 An example of *softvm* and its use

A PowerPC implementation is shown in Fig 5, with a two-level cache hierarchy. Both caches in the hierarchy are virtual and split, to make the cost analysis clearer. Modification and protection bits are kept with each cache line, which should give a conservative cost estimate. In the cost analysis we vary the L1 cache from 2KB to 256KB (1K to 128K per side), and the L2 cache between 1MB and 2MB.

We assume for the sake of argument a 4GB maximum physical memory. To parallel the MIPS design, the top bits of the virtual address space (in this case, 20 of 52 bits)

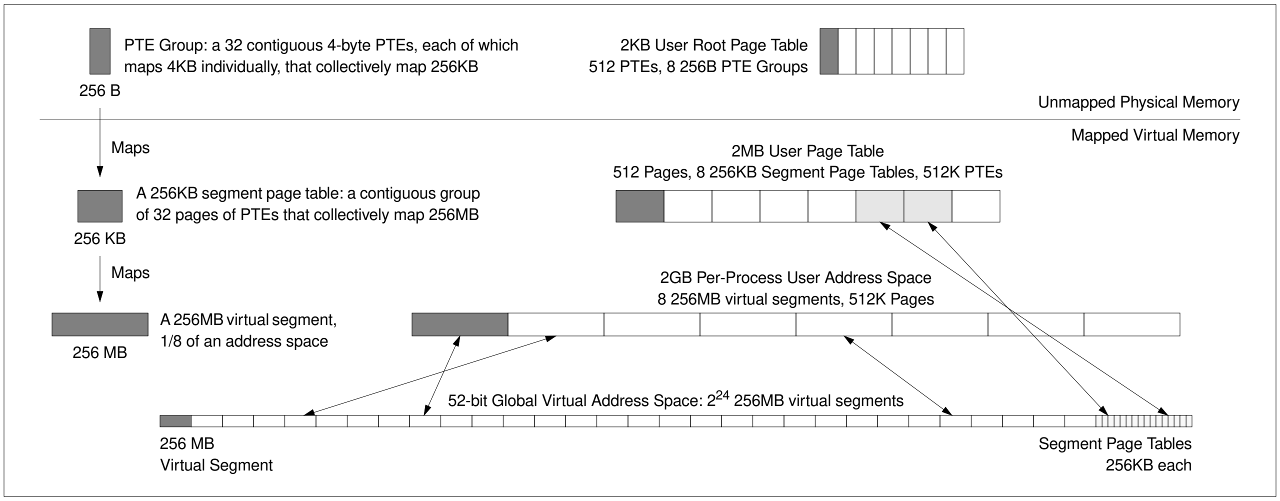

Figure 6: An example page table organization. There is a single linear page table at the top of the 52-bit address space that maps the entire global space. The 256KB *Segment Page Tables* that comprise the user page table are taken directly from this global page table. Therefore, though it may seem that there is a separate user page table for every process, each page table is simply mapped onto the global space; the only per-process allocation is for the user root page table. Though it is drawn as an array of contiguous pages, the user page table is really a disjunct set of 4KB pages in the global space.

determine whether an address is physical and/or cacheable; this is to allow physical addresses to be cached in the virtually indexed, virtually tagged caches. Also like MIPS, a user process owns the bottom 2GB of the 4GB effective address space. Therefore only the bottom 8 of the 16 segment registers are used by applications; the user address space is composed of 8 256MB virtual segments.

To demonstrate the use of *softvm*, we need also define a page table and cache-miss handler. We would like something similar to the MIPS page table organization, as it maps a 32-bit address space with a minimum of levels and supports sparse address spaces easily. A global virtual address space, however, suggests the use of a global page table, which *cannot* be mapped by a small, wired-down piece of memory, meaning that we might need more than two levels in our page table. However, each process need only map enough of the global page table to in turn map its 2GB address space. Therefore, a process uses no more than 2MB of the global table at any given time, which can be mapped by a 2KB user root page table.

A virtual linear table is at the top of the global address space, 2^{42} bytes long, mapping the entire global space (pages are software-defined at 4K bytes, PTEs are 4 bytes). The page table organization, shown in Fig 6, is a two-tiered hierarchy. The lower tier is a 2MB virtual structure, divided into 8 256KB *segment page tables*, each of which (collectively) maps one of the 256MB virtual segments in the user address space. The segment page tables come directly from the global table, therefore there is no per-process allocation of user page tables; if two processes share a virtual segment they share a portion of the global table. The top tier of the page table is a 2KB structure wired down in memory while the process is running; it is the bottom half of the process control block. It is divided into 8 256-byte *PTE groups*, each of which maps a 256KB segment page table that in turn maps a 256MB segment. PTE groups must be duplicated across user root page tables to share virtual segments.

We illustrate in Fig 7 the algorithm for handling misses in the L2 cache. Processes generate 32-bit effective addresses that are extended to 52 bits by segmentation, replacing the top four bits of the effective address. In step 1, the VPN of a 52-bit failing global virtual address is used as an index into the global page table to reference the PTE mapping the failing data (the UPTE). This is similar to the concatenation of PTEBase and VPN to index into the MIPS user page table (Fig 4). The bottom two bits of the address are 0's, since the PTE size is four bytes. The top ten bits of the address are 1's since the table is at the very top of the global space.

If this misses in the L2 cache, the operating system takes a recursive CACHEMISS exception. At this point, we must locate the mapping PTE in the user root page table. This table is an array of PTEs that cannot be indexed by a global VPN. It mirrors the structure of the user's perceived address space, not the structure of the global address space. Therefore it is indexed by a portion of the original 32-bit effective address. The top 10 bits of the effective address index 1024 PTEs that would map a 4MB user page table, which would in turn map a 4GB address space. Since the top bit of the effective address is guaranteed to be zero (the address is a user reference), only the bottom nine bits of the top ten are meaningful; these bits index the array of 512 PTEs in the user root page table. In step 2, the operating system builds a physical address for the appropriate PTE in the user root page table (the URPTE), a 52-bit virtual address whose top 20 bits indicate physical+cacheable. It then loads the URPTE, which maps the UPTE that missed the cache at the end of step 1. When control is returned to the miss handler in step 1, the UPTE load retry will complete successfully.

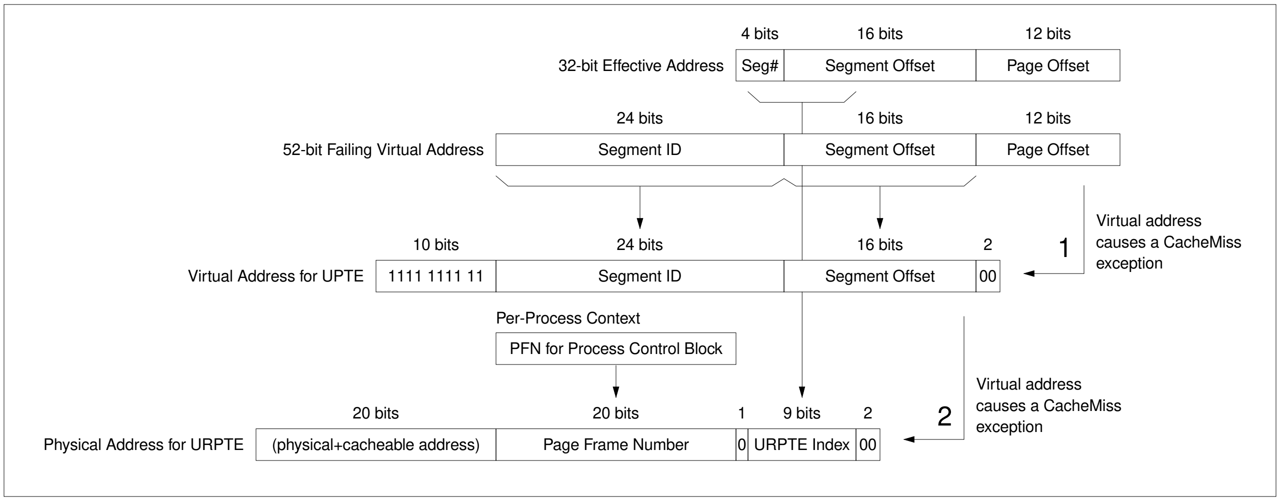

Figure 7: An example cache miss algorithm. Step 1 is the result of a user-level L2 cache miss; the operating system builds a virtual address for a PTE in the global page table. If this PTE is not found in the L1 or L2 cache a root PTE is loaded, shown in step 2. One special requirement is a register holding the initial failing address. Another required hardware structure, the per-process context register, points to the process control block of the active process.

The operating system then performs a SPECIFYVTAG using the most significant bits of the failing 52-bit address, and performs a LOAD&MAP using the physical address for the failing data, built from the PFN in the UPTE and the page offset from the failing address. This loads the failing data and inserts it into the cache using the user's virtual tag.

4.3 Memory system requirements, revisited

We now revisit the memory management requirements listed earlier, and discuss how *softvm* supports them.

Address space protection and large address spaces. These memory management functions are not inherent to software-managed address translation, but a *softvm* design does not preclude their implementation. They are satisfied in our example through the use of PowerPC segments. As described earlier, segments provide address space protection, and by their definition provide a global virtual space onto which all effective addresses are mapped. A process could use its 4GB space as a window onto the larger space, moving virtual segments in and out of its working set as necessary. This type of windowing mechanism is used on the PA-RISC [27].

Shared memory. The sharing mechanism is defined by the page table. One can simplify virtual cache management by sharing memory via global addresses, a scheme used in many systems [7, 8, 16, 18, 20, 21, 45], and shown to have good performance. Alternatively, one could share memory through virtual-address aliasing.

Fine-grained protection. One can maintain protection bits in the cache, or in an associated structure like a TLB. If one could live with protection on a per-segment basis, one could maintain protection bits in the segment registers. For our discussion we maintain protection bits in the cache line. Protection granularity therefore becomes a software issue; the page size can be anything from the entire address space down to a single cache line. Note the choice of this granularity does not preclude one from implementing segment-level protection as well. The disadvantage is that if one chooses a page size larger than a single cache line, protection information must be replicated across multiple cache lines and the operating system must manage its consistency. We analyze this later.

Sparse address spaces. Sparse address space support is largely a page table issue. Hardware can either get out of the way of the operating system and allow any type of page table organization, or it can inhibit support for sparse address spaces by defining a page table organization that is not necessarily suitable. By eliminating translation hardware, one frees the operating system to choose the most appropriate structure.

Superpages. By removing the TLB one removes hardware support for superpages, but as with sparse address spaces one also frees the operating system to provide support through the page table. For instance, a top-down hierarchical page table (as in the x86 [31]) would provide easy support for superpages. A guarded page table [36, 38] would also provide support, and would map a large address space more efficiently, as would the inverted page table variant described by Talluri, et al. [48].

Direct memory access. While software-managed address translation provides no explicit support for DMA, and actually makes DMA more difficult by requiring a virtual cache, direct memory access is still possible. For example, one could perform DMA by flushing affected pages from the cache before beginning a transfer, and restricting access to the pages during transfer.

Table 1: Qualitative comparison of cache-access/address-translation mechanisms

Event	Frequency of Occurrence		Actions Performed by Hardware and Operating System per Occurrence of Event	
	I-side	D-side	TLB + Virtual cache	Software-Mgd Addr Translation
L1 hit, TLB hit	96.7%	95.8%	L1 access (*w/ TLB access in parallel*)	L1 access
L1 hit, TLB miss	0.01%	0.06%	L1 access + page table access + TLB reload	L1 access
L1 miss, L2 hit, TLB hit	3.2%	3.9%	L1 access + L2 access	L1 access + L2 access
L1 miss, L2 hit, TLB miss	0.03%	0.09%	L1 access + page table access + TLB reload + L2 access	L1 access + L2 access
L1 miss, L2 miss, TLB hit	0.008%	0.12%	L1 access + L2 access + memory access	L1 access + L2 access + page table access + memory access
L1 miss, L2 miss, TLB miss	0.0001%	0.0009%	L1 access + page table access + TLB reload + L2 access + memory access	L1 access + L2 access + page table access + memory access

5 Discussion

Many studies have shown that significant overhead is spent servicing TLB misses [1, 4, 9, 28, 41, 44, 47]. In particular, Anderson, et al. [1] show TLB miss handlers to be among the most commonly executed primitives, Huck and Hays [28] show that TLB miss handling can account for more than 40% of total run time, and Rosenblum, et al. [44] show that TLB miss handling can account for more than 80% of the kernel's computation time. Typical measurements put TLB handling at 5-10% of a normal system's run time.

The obvious question to ask is *does the TLB buy us anything?* Do its benefits outweigh its overhead? We now discuss the performance costs of eliminating the TLB.

5.1 Performance overview

The SPUR and VMP projects demonstrated that with large virtual caches the TLB can be eliminated with no performance loss, and in most cases a performance gain. For a qualitative, first-order performance comparison, we enumerate the scenarios that a memory management system would encounter. These are shown in Table 1, with frequencies obtained from SPECint95 traces on a PowerPC-based AIX machine (frequencies do not sum to 1 due to rounding). The model simulated has 8K/8K direct-mapped virtual L1 caches (in the middle of the L1 cache sizes simulated), 512K/512K direct-mapped virtual L2 caches (the smaller of the two L2 cache sizes simulated), and a 16-byte linesize in all caches. As later graphs will show, the small linesize gives the worst-case performance for the software-managed scheme. The model includes a simulated MIPS-style TLB [32] with 64 entries, a random replacement policy, and 8 slots reserved for root PTEs.

The table shows what steps the operating system and hardware take when cache and TLB misses occur. Note that there is a small but non-zero chance a reference will hit in a virtual cache but miss in the TLB. If so, the system must take an exception and execute the TLB miss handler before continuing with the cache lookup, despite the fact that the data is in the cache. On TLB misses, a software-managed scheme should perform much better than a TLB scheme. When the TLB hits, the two schemes should perform similarly, except when the reference misses the L2 cache. Here the TLB already has the translation, but the software-managed scheme must access the page table for the mapping (note that the page table entry may in fact be cached). Software-managed translation is not penalized by placing PTEs in the cache hierarchy; many operating systems locate their page tables in cached memory for performance reasons.

5.2 Baseline overhead

Table 2 shows the overheads of TLB handling in several operating systems as percent of run-time and CPI. Percent of run-time is the total amount of time spent in TLB handlers divided by the total run-time of the benchmarks. CPI overhead is the total number of cycles spent in TLB handling routines divided by the total number of cycles in the benchmarks. The data is taken from previous TLB studies [4, 40, 41] performed on MIPS-based DECstations, which use a software-managed TLB. CPI is not directly proportional to run-time overhead for two reasons: (1) the run-time overhead contains page protection modifications and the CPI overhead does not, and (2) memory stalls make it

Table 2: TLB overhead of several operating systems

System	Overhead (% run-time)	Overhead (CPI)
Ultrix	2.03%	0.042
OSF/1	5.81%	0.101
Mach3	8.21%	0.162
Mach3+AFSin	7.77%	0.220
Mach3+AFSout	8.88%	0.281

difficult to predict total cycles from instruction counts.

Table 3 gives the overheads of the software-managed design, divided by benchmark to show a distribution. The values come from trace-driven simulation of the SPEC95 integer suite. The simulations use the same middle-of-the-line cache organization as before (8K/8K L1, 512K/512K L2, 16-byte linesize throughout), but replace the TLB with software address translation. Our memory penalties are 1 cycle to access the L1 cache, 20 cycles to access the L2 cache, and 90 cycles to access main memory.

Table 3: Overhead of software-managed address translation

Workload	Overhead (CPI)
m88ksim	0.003
li	0.003
go	0.004
compress95	0.009
perl	0.019
ijpeg	0.052
vortex	0.060
gcc	0.097
Weighted Average:	0.033

Table 4 gives a more detailed breakdown of costs for one of the benchmarks: gcc. Our example miss handler from the previous section requires 10 instructions including two loads. It is very similar to the MIPS TLB refill handler that requires less than 10 instructions including one load, taking 10 cycles when the load hits in the cache, or 40+ when the load misses in the cache, thereby forcing the reference to main memory [4]. In our model, the L2 cache miss handler always takes 10 cycles, and runs whenever we take an L2 cache miss (labeled *L2 I-Cache miss* or *L2 D-Cache miss* in the table). When the PTE load in the handler misses the L1 cache (*Miss handler L1 D-miss*) we take an additional 20 cycles to go to the L2 cache to look for the PTE. If that load misses we either take a recursive cache miss (if handling a user-miss, therefore the PTE address is virtual, accounted for in *L2 D-Cache miss*), or the address is physical and goes straight through to main memory (*Miss handler L2 D-miss*, 90 cycles). When the miss handler is handling a miss from the handler itself, we need also load the failing UPTE on

Table 4: Breakdown of GCC overhead

Event	Frequency (per instr.)	Penalty per Occurrence	Overhead (CPI)
L2 D-Cache Ld/St miss	0.000697	10 cycles	0.006970
L2 I-Cache I-fetch miss	0.004756	10 cycles	0.047560
Miss handler L1 D-miss	0.000596	20 cycles	0.011920
Miss handler L2 D-miss	0.000032	90 cycles	0.002880
Miss handler Load UPTE	0.000053	90 cycles	0.004770
Miss handler L1 I-miss	0.000985	20 cycles	0.019700
Miss handler L2 I-miss	0.000035	90 cycles	0.003150
		Total CPI:	0.096950

behalf of the handler (*Miss handler Load UPTE*, 90 cycles).

Additionally, the miss-handler code can miss in the L1 or L2 I-caches; since it is mapped directly onto physical memory it does not cause a cache miss itself. However, for every instruction fetch that misses in the L1 cache we take a 20-cycle penalty to reference the L2 cache; for every L2 miss we take a 90-cycle penalty to reference physical memory.

The average overhead of the scheme is 0.033 CPI. This is about the overhead measured of Ultrix on MIPS, considered to be an example of an efficient match between OS and architecture. This CPI is several times better than that of Mach, which should result in a run-time savings of at least 5% over Mach. However, the number does not take into account the effect of writebacks.

5.3 Writebacks

When a cache miss occurs in a writeback cache, a common rule of thumb says that half the time the line expelled from the cache will be dirty, requiring it to be written back to main memory. This case must be dealt with at the time of our CacheMiss exception. There are two obvious solutions. The translation is available at the time a cache line is brought into the cache; one can either discard this information or store it in hardware. If discarded, the translation must be performed again at the time of the writeback. If one wishes to throw hardware at the problem one can keep the translation with the cache line, simplifying writeback enormously but increasing the size of the cache without increasing its capacity. This also introduces the possibility of having stale translation information in the cache. We do not discuss the hardware-oriented solution further, as the purpose of this paper is to investigate reducing address translation hardware to its simplest.

If writebacks happen in 50% of all cache misses, then 50% of the time we will need to perform two address translations: one for the data to be written back, one for the data to be brought into the cache. This should increase our overhead by roughly 50%, from 0.033 CPI to 0.050 CPI, which is about the overhead of Ultrix and still far less than that of OSF/1 or Mach. The problem this introduces is that the writeback handler can itself cause another writeback if it touches data in cacheable space, or if the handler code is in cacheable space and the caches are unified.

5.4 Fine-grained protection

As mentioned earlier, managing protection information can be inefficient if we store protection bits with each cache line. If the protection granularity is larger than a cache line, the bits must be replicated across multiple lines. Keeping the protection bits consistent across the cache lines can cause significant overhead if page protection is modified frequently. The advantage of this scheme is that the choice of protection granularity is completely up to the operating system. In this section, we determine the overhead.

We performed a study on the frequency of page protection modifications in the Mach operating system. The benchmarks are the same as in [41], and the operating system is Mach3. We chose Mach as it uses copy-on-write liberally, producing 1000 times the page-protection modifications seen in Ultrix [41]. We use these numbers to determine the protection overhead of our system; this should give a conservative estimate for the upper bound. The results are shown in Table 5.

Table 5: Page protection modification frequencies in Mach3

Workload	Page Protection Modifications	Modifications per Million Instructions
compress	3635	2.8
jpeg_play	12083	3.4
IOzone	3904	5.1
mab	27314	15.7
mpeg_play	26129	19.0
gcc	35063	22.3
ousterhout	15361	23.8
	Weighted Average:	11.3

Page-protection modifications occur on the average of 11.3 for every million instructions. At the very worst, for each modification we must sweep through a page-sized portion of the L1 and L2 caches to see if lines from the affected page are present. Overhead therefore increases with larger page sizes (a software-defined parameter) and with smaller linesizes (a hardware-defined parameter). On a system with 4KB pages and a 16-byte linesize, we must check 256 cache lines per modification. Assuming an average of 10 L1 cache lines and 50 L2 caches lines affected per modification[4], if L1 cache lines can be checked in 3 cycles and updated in 5 cycles (an update is a check-and-modify), and L2 cache lines can be checked in 20 cycles and updated in 40 cycles, we calculate the overhead as follows. Of 256 L1 cache lines, 10 must be updated (5 cycles), the remaining 246 need only be checked (3 cycles); of 256 L2 cache lines 50 must be updated (40 cycles), the remaining 206 need only be checked (20 cycles); the overhead is therefore 6908 cycles per page-protection modification (10 * 5 + 246 * 3 + 50 * 40 + 206 * 20). This yields between 0.019 and 0.164 CPI ($6908 * 2.8 * 10^{-6}$ and $6908 * 23.8 * 10^{-6}$). This is in the range of Ultrix and OSF/1 overheads and at the lower end of Mach's overhead. This translates to a worst case of 2-7% total execution time. If the operating system uses page-protection modification as infrequently as in Ultrix, this overhead decreases by three orders of magnitude to 0.0001 CPI, or about 0.01% execution time.

We can improve this by noting that most of these modifications happen during copy-on-write. Often the protections are being increased and not decreased, allowing one to update protection bits in each affected cache line lazily—to delay an update until a read-only cache line is actually written, at which point it would be updated anyway.

5.5 Sensitivity to cache organization

The graphs in Fig 8 show the sensitivity of software-managed address translation to cache size and cache linesize. The benchmarks are from SPEC95 as before; we only show graphs for the two worst-performing benchmarks—gcc and vortex. The numbers differ slightly from those presented in Table 3; the benchmarks were not run to completion for this study, but were stopped after 1 billion references for each.

Besides the familiar signature of diminishing returns from increasing linesize (e.g., the two largest overheads in Fig 8a are from the smallest and largest linesizes), the graphs show that cache size has a significant impact on the overhead of the system. For gcc, overhead decreases by an order of magnitude when the L2 cache is doubled, and decreases by a factor of three as the L1 cache increases from 1KB to 128KB (2KB to 256KB total L1 cache size); for vortex, overhead decreases by a factor of two as the L2 cache doubles, and decreases by a factor of three as L1 increases from 1KB to 128KB. Within a given cache size, linesize choice can affect performance by a factor of two or more (up to ten for some configurations).

The best organization should result in an overhead an order of magnitude lower than that calculated earlier—to less than 0.01 CPI, or a run-time overhead far less than 1%. This suggests that software-managed address translation is viable today as a strategy for faster, nimbler systems.

4. We chose these numbers after inspecting individual SPEC95 benchmark traces, which should give conservative estimates: (1) SPEC working sets tend to be smaller than normal programs, resulting in less page overlap in the caches, and (2) individual traces would have much less overlap in the caches than multiprogramming traces.

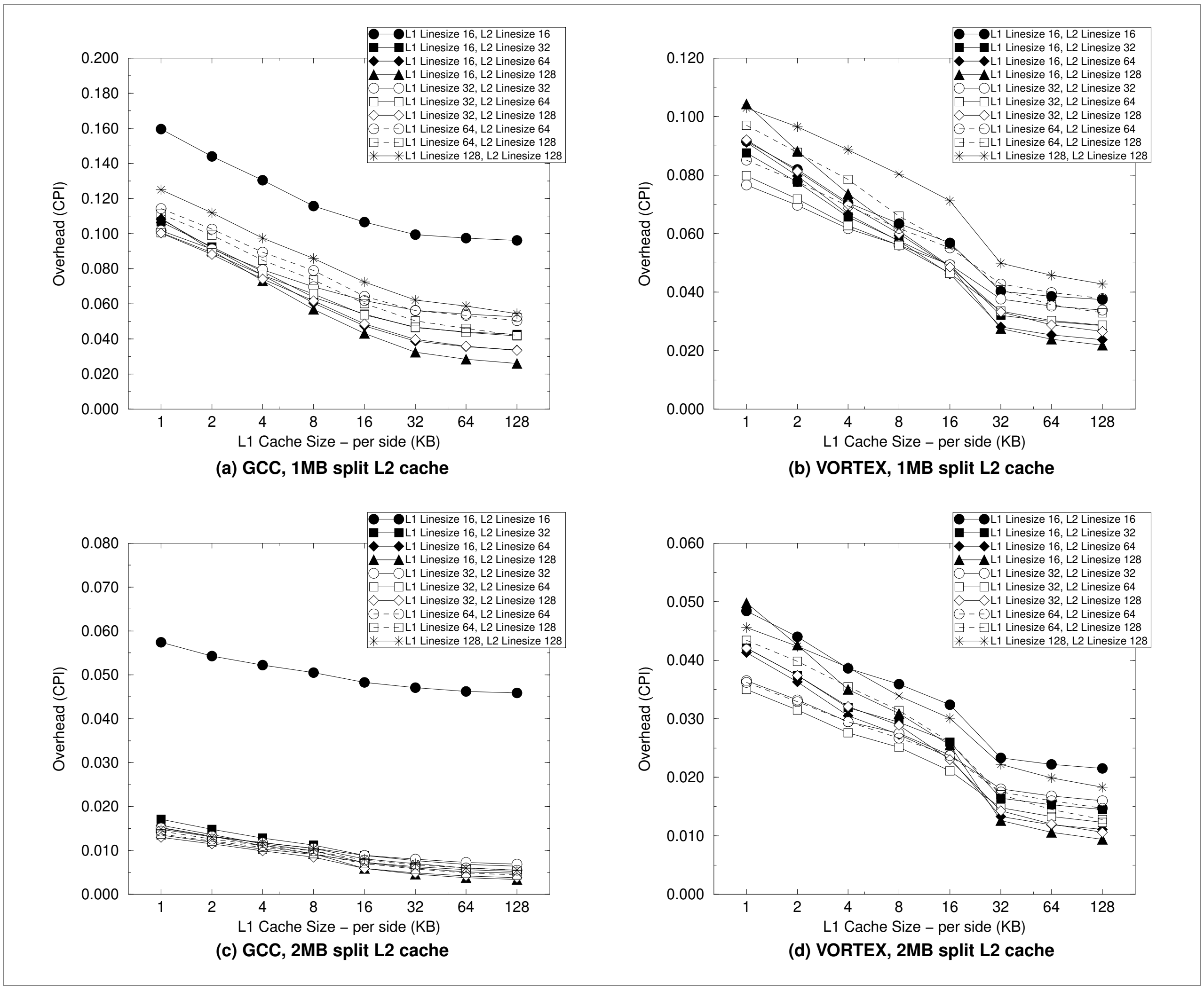

Figure 8: The effect of cache size and linesize on software-managed address translation. The figure shows two benchmarks—gcc and vortex. All caches are split. L1 cache size is varied from 1K to 128KB per side (2KB to 256KB total), and L2 cache size is varied from 512KB to 1024KB per side (1MB to 2MB total). Linesizes are varied from 16 bytes to 128 bytes; the L2 linesize is never less than the L1 linesize. In each simulation, the I-caches and D-caches have identical configurations. We apologize for using different y-axis scales; however, they better show the effects of linesize for a given cache size.

6 Summary

We are building a high clock-rate 32-bit PowerPC. For the design of the memory management system, we have returned to first principles and discovered a small set of hardware structures that provide support for address space protection, shared memory, large sparse address spaces, and fine-grained protection at the cache-line level. This set does not include address-translation hardware; we show that address translation can be managed in software efficiently. Current virtual memory systems such as Mach exact an overhead of 0.16 to 0.28 cycles per instruction to provide address translation; a software scheme requires 0.05 CPI (2% run-time, with a 16KB L1 cache, and 1MB L2), about the same as the overhead of Ultrix on MIPS. If copy-on-write and other page-protection modifications are used as frequently as in Mach, protection-bit management can increase this overhead to that of OSF/1 or Mach. However, the number of page-protection modifications in Ultrix represent a negligible overhead. With slightly larger caches (2MB L2, common in today's systems), the overhead of software-managed address translation should reduce to far less than 1% of run-time. Therefore software-managed address translation is a viable strategy for high-end computing today, achieving better performance with less hardware.

Beyond the performance gains suggested by our simulations, the benefits of a minimal hardware design are threefold. First, moving address translation into software creates a simpler and more flexible interface; as such it supports much more innovation in the operating system than would a fixed design. Second, a reduction in hardware will leave room for more cache structures, increasing performance. Last, simpler hardware should be easier to design and debug, cutting down on development time.

References

[1] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. "The interaction of architecture and operating system design." In *Proc. Fourth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS 4)*, April 1991, pp. 108–120.

[2] A. W. Appel and K. Li. "Virtual memory primitives for user programs." In *Proc. Fourth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS 4)*, April 1991, pp. 96–107.

[3] M. J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1986.

[4] K. Bala, M. F. Kaashoek, and W. E. Weihl. "Software prefetching and caching for translation lookaside buffers." In *Proc. First USENIX Symposium on Operating Systems Design and Implementation*, November 1994.

[5] B. N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer. "SPIN – an extensible microkernel for application-specific operating system services." Tech. Rep. 94-03-03, University of Washington, February 1994.

[6] A. Chang and M. F. Mergen. "801 storage: Architecture and programming." *ACM Transactions on Computer Systems*, vol. 6, no. 1, February 1988.

[7] J. S. Chase, H. M. Levy, M. Baker-Harvey, and E. D. Lazowska. "How to use a 64-bit virtual address space." Tech. Rep. 92-03-02, University of Washington, March 1992.

[8] J. S. Chase, H. M. Levy, E. D. Lazowska, and M. Baker-Harvey. "Lightweight shared objects in a 64-bit operating system." Tech. Rep. 92-03-09, University of Washington, March 1992.

[9] J. B. Chen, A. Borg, and N. P. Jouppi. "A simulation based study of TLB performance." In *Proc. 19th Annual International Symposium on Computer Architecture (ISCA 19)*, May 1992.

[10] R. Cheng. "Virtual address cache in UNIX." In *Proceedings of the Summer 1987 USENIX Technical Conference*, June 1987.

[11] D. R. Cheriton, H. A. Goosen, and P. D. Boyle. "Multi-level shared caching techniques for scalability in VMP-MC." In *Proc. 16th Annual International Symposium on Computer Architecture (ISCA 16)*, June 1989.

[12] D. R. Cheriton, A. Gupta, P. D. Boyle, and H. A. Goosen. "The VMP multiprocessor: Initial experience, refinements and performance evaluation." In *Proc. 15th Annual International Symposium on Computer Architecture (ISCA 15)*, May 1988.

[13] D. R. Cheriton, G. A. Slavenburg, and P. D. Boyle. "Software-controlled caches in the VMP multiprocessor." In *Proc. 13th Annual International Symposium on Computer Architecture (ISCA 13)*, January 1986.

[14] D. W. Clark and J. S. Emer. "Performance of the VAX-11/780 translation buffer: Simulation and measurement." *ACM Transactions on Computer Systems*, vol. 3, no. 1, February 1985.

[15] H. Custer. "Inside Windows/NT." Tech. Rep., Microsoft Press, 1993.

[16] H. Deitel. *Inside OS/2*. Addison-Wesley, Reading MA, 1990.

[17] Digital. *DECchip 21064 and DECchip 21064A Alpha AXP Microprocessors Hardware Reference Manual*. Digital Equipment Corporation, Maynard MA, 1994.

[18] P. Druschel and L. L. Peterson. "Fbufs: A high-bandwidth cross-domain transfer facility." In *Proc. Fourteenth ACM Symposium on Operating Systems Principles*, December 1993, pp. 189–202.

[19] D. Engler, R. Dean, A. Forin, and R. Rashid. "The operating system as a secure programmable machine." In *Proc. 1994 European SIGOPS Workshop*, September 1994.

[20] W. E. Garrett, M. L. Scott, R. Bianchini, L. I. Kontothanassis, . R. A. McCallumm, J. A. Thomas, R. Wisniewski, and S. Luk. "Linking shared segments." In *USENIX Technical Conference Proceedings*, January 1993.

[21] W. E. Garrett, R. Bianchini, L. Kontothanassis, . R. A. McCallum, J. Thomas, R. Wisniewski, and M. L. Scott. "Dynamic sharing and backward compatibility on 64-bit machines." Tech. Rep. TR 418, University of Rochester, April 1992.

[22] J. R. Goodman. "Coherency for multiprocessor virtual address caches." In *Proc. Second Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2)*, October 1987, pp. 72–81.

[23] J. Heinrich, Ed. *MIPS R10000 Microprocessor User's Manual, version 1.0*. MIPS Technologies, Inc., Mountain View CA, June 1995.

[24] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[25] Hewlett-Packard. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*. Hewlett-Packard Company, 1990.

[26] M. D. Hill, et al. "Design Decisions in SPUR." *IEEE Computer*, vol. 19, no. 11, November 1986.

[27] J. Huck. *Personal communication*. 1996.

[28] J. Huck and J. Hays. "Architectural support for translation table management in large address space machines." In *Proc. 20th Annual International Symposium on Computer Architecture (ISCA 20)*, May 1993.

[29] IBM and Motorola. *PowerPC 601 RISC Microprocessor User's Manual*. IBM Microelectronics and Motorola, 1993.

[30] J. Inouye, R. Konuru, J. Walpole, and B. Sears. "The effects of virtually addressed caches on virtual memory design and performance." Tech. Rep. CS/E 92-010, Oregon Graduate Institute, 1992.

[31] Intel. *Pentium Processor User's Manual*. Intel Corporation, Mt. Prospect IL, 1993.

[32] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs NJ, 1992.

[33] Y. A. Khalidi, M. Talluri, M. N. Nelson, and D. Williams. "Virtual memory support for multiple page sizes." In *Proc. Fourth Workshop on Workstation Operating Systems*, October 1993.

[34] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, 1989.

[35] J. Liedtke. "Improving IPC by kernel design." In *Proc. Fourteenth ACM Symposium on Operating Systems Principles*, December 1993, pp. 175–187.

[36] J. Liedtke. "Address space sparsity and fine granularity." *ACM Operating Systems Review*, vol. 29, no. 1, pp. 87–90, January 1995.

[37] J. Liedtke. "On micro-kernel construction." In *Proc. Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[38] J. Liedtke and K. Elphinstone. "Guarded page tables on MIPS R4600." *ACM Operating Systems Review*, vol. 30, no. 1, pp. 4–15, January 1996.

[39] C. May, E. Silha, R. Simpson, and H. Warren, Eds. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, San Francisco CA, 1994.

[40] D. Nagle. *Personal communication*. 1995.

[41] D. Nagle, R. Uhlig, T. Stanley, S. Sechrest, T. Mudge, and R. Brown. "Design tradeoffs for software-managed TLBs." In *Proc. 20th Annual International Symposium on Computer Architecture (ISCA 20)*, May 1993.

[42] R. Rashid, A. Tevanian, M. Young, D. Young, R. Baron, D. Black, W. Bolosky, and J. Chew. "Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures." *IEEE Transactions on Computers*, vol. 37, no. 8, pp. 896–908, August 1988.

[43] S. A. Ritchie. "TLB for free: In-cache address translation for a multiprocessor workstation." Tech. Rep. UCB/CSD 85/233, University of California, May 1985.

[44] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. "The impact of architectural trends on operating system performance." In *Proc. Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[45] M. L. Scott, T. J. LeBlanc, and B. D. Marsh. "Design rationale for Psyche, a general-purpose multiprocessor operating system." In *Proc. 1988 International Conference on Parallel Processing*, August 1988.

[46] R. L. Sites, Ed. *Alpha Architecture Reference Manual*. Digital Equipment Corporation, Maynard MA, 1992.

[47] M. Talluri and M. D. Hill. "Surpassing the TLB performance of superpages with less operating system support." In *Proc. Sixth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS 6)*, October 1994.

[48] M. Talluri, M. D. Hill, and Y. A. Khalidi. "A new page table for 64-bit address spaces." In *Proc. Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[49] M. Talluri, S. Kong, M. D. Hill, and D. A. Patterson. "Tradeoffs in supporting two page sizes." In *Proc. 19th Annual International Symposium on Computer Architecture (ISCA 19)*, May 1992.

[50] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. "Efficient software-based fault isolation." In *Proc. Fourteenth ACM Symposium on Operating Systems Principles*, December 1993, pp. 203–216.

[51] W.-H. Wang, J.-L. Baer, and H. M. Levy. "Organizaiton and performance of a two-level virtual-real cache hierarchy." In *Proc. 16th Annual International Symposium on Computer Architecture (ISCA 16)*, June 1989, pp. 140–148.

[52] D. L. Weaver and T. Germand, Eds. *The SPARC Architecture Manual version 9*. PTR Prentice Hall, Englewood Cliffs NJ, 1994.

[53] S. Weiss and J. E. Smith. *POWER and PowerPC*. Morgan Kaufmann Publishers, San Francisco CA, 1994.

[54] B. Wheeler and B. N. Bershad. "Consistency management for virtually indexed caches." In *Proc. Fifth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS 5)*, October 1992.

[55] D. A. Wood. *The Design and Evaluation of In-Cache Address Translation*. PhD thesis, University of California at Berkeley, March 1990.

[56] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, J. M. Pendleton, S. A. Ritchie, G. S. Taylor, R. H. Katz, and D. A. Patterson. "An in-cache address translation mechanism." In *Proc. 13th Annual International Symposium on Computer Architecture (ISCA 13)*, January 1986.