Uniprocessor Virtual Memory without TLBs

Bruce Jacob, *Member*, *IEEE*, and Trevor Mudge, *Fellow*, *IEEE*

Abstract—We present a feasibility study for performing virtual address translation without specialized translation hardware. Removing address translation hardware and instead managing address translation in software has the potential to make the processor design simpler, smaller, and more energy-efficient at little or no cost in performance. The purpose of this study is to describe the design and quantify its performance impact. Trace-driven simulations show that software-managed address translation is just as efficient as hardware-managed address translation. Moreover, mechanisms to support such features as shared memory, superpages, fine-grained protection, and sparse address spaces can be defined completely in software, allowing for more flexibility than in hardware-defined mechanisms.

Index Terms—Virtual memory, virtual address translation, virtual caches, memory management, software-managed address translation, translation lookaside buffers.

1 Introduction

Changing trends in technologies, notably cheaper and faster memory hierarchies, have made it worthwhile to revisit many hardware-oriented design decisions made in previous decades. Hardware-oriented designs, in which one uses special-purpose hardware to perform some dedicated function, are a response to the high cost of executing instructions out of memory; when caches are expensive, slow, and/or in scarce supply, it is a perfectly reasonable reaction to build hardware state machines that do not compete with user applications for cache space and do not rely on the performance of the caches. In contrast, when the caches are large enough to withstand competition between the application and operating system, the cost of executing operating system functions out of the memory subsystem decreases significantly and software-oriented designs become viable. Software-oriented designs, in which one dispenses with special-purpose hardware and instead performs the same function entirely in software, can offer increased flexibility over hardware state machines at a modest cost in performance. One current example is the translation of x86 instructions by Transmeta's *code-morphing* software layer [11]; this performs the same type of function as the front-end of the Pentium Pro/II/III pipeline, which turns x86 instructions into RISC-like *uops* in hardware [16].

This paper describes a software-oriented design for a virtual memory management system. It shows that a software-oriented scheme can perform nearly as well as hardware schemes and it is more flexible. Eliminating dedicated special-purpose hardware from processor design can save chip area [25] and can reduce power consumption —e.g., the StrongARM TLB consumes 17 percent of the chip's power [26] and, so, eliminating the TLB would therefore reduce power consumption by a significant amount. Reducing die area and/or power potentially lowers the overall system cost. Moreover, a flexible design should aid in the portability of system software by allowing the operating system to dictate details to the hardware, as opposed to the other way around. A software-oriented design methodology should therefore benefit architects of many different microprocessor designs, from general-purpose processors in PC-class and workstation-class computers, to embedded processors where cost tends to have a higher priority than performance.

1.1 Efficient Virtual Memory without TLBs

In this paper, we demonstrate *software-managed address translation*, or *softvm* for short. The mechanism dispenses with the translation lookaside buffers (TLBs) found in nearly every modern microarchitecture and the page-table-walking state machines found in x86 and PowerPC architectures. It uses a software-handled cache miss, like the VMP multiprocessor [8], except that VMP used the mechanism to explore cache coherence in a multiprocessor, while softvm uses it to simplify memory management hardware in a uniprocessor. VMP also stored the cache-miss handler in a dedicated buffer next to the processor, while the *softvm* handler code is part of the operating system in main memory. *Softvm* also resembles the in-cache address translation mechanism of SPUR [32], [46] in its lack of TLBs, but it takes the design one step further by eliminating table-walking hardware.

The scheme is relatively simple and requires a minimum of hardware components: a virtually indexed, virtually tagged cache hierarchy and a mechanism to implement a software-managed cache miss at the lowest cache level (L2, for example). Because virtual caches do not require address translation when requested data is found in the cache, they obviate a TLB; further, if a miss in the L2 cache invokes the operating system's memory manager, the operating system is free to implement any type of page table, protection scheme, or replacement policy—even a software-defined

- *B. Jacob is with the Electrical and Computer Engineering Department, University of Maryland, College Park, MD 20742. E-mail: blj@eng.umd.edu.*
- *T. Mudge is with the Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI 48109. E-mail: tnm@eecs.umich.edu.*

Manuscript received 26 Oct. 1998; revised 24 Oct. 2000; accepted 23 Feb. 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 108114.

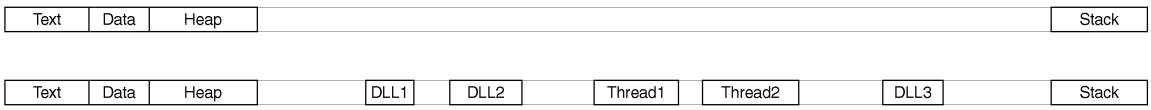

Fig. 1. **Space address spaces.** The top address space is that of a traditional 4.3BSD process, with contiguous text, data, and heap segments and a continuous stack against segment. The bottom address space contains modern features like dynamically loaded libraries and multiple threads of control, which leave holes with 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.

page size. This, then, is the extent of the mechanism studied in this paper: a virtual cache hierarchy where accesses to main memory are translated by the operating system, not the TLB. It represents one end of the continuum of design choices and, though simple, this configuration has not been previously studied.

We show the efficiency of the scheme by analyzing a specific implementation, thereby finding an upper bound on virtual-memory overhead. We find that, for large caches, the virtual-memory overhead is similar to that of hardware-managed schemes. The example we study adds software-managed translation to a conventional PowerPC memory management organization. PowerPC segments support address space protection and shared memory and provide access to a large virtual address space. Segments 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 global address space. However, the use of segments in conjunction with a virtual cache organization is one method to solve the consistency problems associated with virtual caches.

2 Memory-Management 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, 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. An alternative hardware approach is *paged segmentation*, as implemented in the PowerPC, PA-RISC, and IA-32 architectures, in which the virtual-physical translation occurs in two steps. The first step maps user addresses to a global address space at the granularity of segments; the second step maps addresses from the global space to physical memory at the granularity of pages. Finally, protection can be provided by software means [41].

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 application and library 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 [40].

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.

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 fine-grained 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 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. One can achieve significant performance gains by reducing the number of TLB entries to cover the current working set [38].

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 typically 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 may reduce performance by requiring the processor to go to main memory too often.

These represent a core set of features that any modern virtual memory system must offer through either hardware or software means. Therefore, it is fundamental to any hardware design that, if the design does not offer a particular feature, it must not preclude a software implementation of that feature.

3 Background and Related Work

This paper describes software-managed address-translation. 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 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

Our scheme depends on virtual caches, which are known to 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 [15] and this has been shown to cause significant overhead [44]; protection-bit modification is used to implement such features as copy-on-write [1], [31] and it can also cause significant overhead when used frequently.

The synonym problem has been solved in hardware using schemes such as dual tag sets [15] or back-pointers [42]; it can also be avoided by setting policy in the operating system. For example, OS/2 places all shared segments at identical virtual addresses in all process address spaces [10]. SunOS aligns shared pages on extremely large virtual boundaries to ensure that aliases map to the same cache block [7].[1] Single address space operating systems (SASOS) such as Opal [5] or Psyche [35] eliminate the need for virtual-address aliasing entirely: All shared data is referenced through global addresses, allowing pointers to be shared freely.

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.

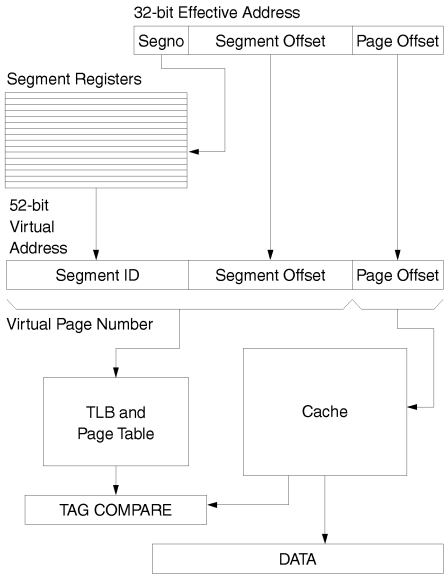

Fig. 2. **PowerPC segmented address translation.** Processes generate 32-bit effective addresses that are mapped onto a 52-bit address space via 16 segment registers, using the top four bits of the effective address as an index. It is this extended virtual address that is mapped by the TLB and page table. The segments provide address space protection and can be used for shared memory.

3.2 PowerPC: Segmented Translation

The IBM 801 introduced a segmented design that remained through the POWER and PowerPC architectures [4], [19], [29], [43]; it is illustrated in Fig. 2 and described in detail elsewhere [24], [23]. Applications generate 32-bit "effective" addresses that are mapped onto a larger "virtual" address space at the granularity of *segments*, 256 MB virtual regions. This extended address is used to index the TLB and page table. The operating system performs data movement and relocation at the granularity of pages.

One of the advantages of this arrangement is that it solves the virtual-cache synonym problem in a manner similar to SASOS organizations while still supporting private, protected address spaces that are not part of a larger whole. This is illustrated in Fig. 3: If memory is shared at a segment granularity, it is possible to use global addresses for shared items transparently. Therefore, no aliasing and no virtual-cache synonyms can occur and the TLB and cache need not be flushed on context switch.[2] Processes may map shared segments at arbitrary segment-aligned addresses or even at multiple locations. Enforcing a one-to-one correspondence between physical addresses and global virtual addresses ensures that a physical datum can exist in one and only block of a virtual cache in any given instant, even if the cache is set-associative.

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.

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 [12], [27]. If more processes are desired, identifiers must be constantly reassigned, requiring TLB and virtual-cache flushes.

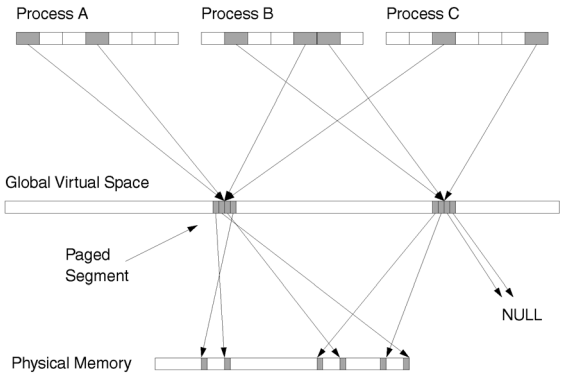

Fig. 3. **Virtual address aliasing in a segmented architecture.** The figure shows three processes sharing two segments. None of the processes use the same virtual address for the same physical data and two of the processes go so far as to map a segment at multiple locations within their address spaces. Nonetheless, these aliases will not result in any synonym problems in a virtual cache since there is a one-to-one correspondence between pages in the global virtual space and pages in physical memory.

3.3 MIPS: A Simple Page Table Design

MIPS 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. 4. The MIPS virtual memory system is described in detail elsewhere [23].

The architecture is one of the first designs to walk a page table bottom-up [24]. The VPN of an address that misses the TLB indexes the user-level page table as part of a virtual address. The architecture provides hardware support for this, storing the virtual base of the user page table in a register (*TLB Context*) and concatenating this with the VPN on TLB misses. This is illustrated in Fig. 5. The advantage is that, in most instances, a PTE lookup will require only one memory access—as opposed to top-down page tables and traditional inverted tables, which require a minimum of two. The software handler is less than 10 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.

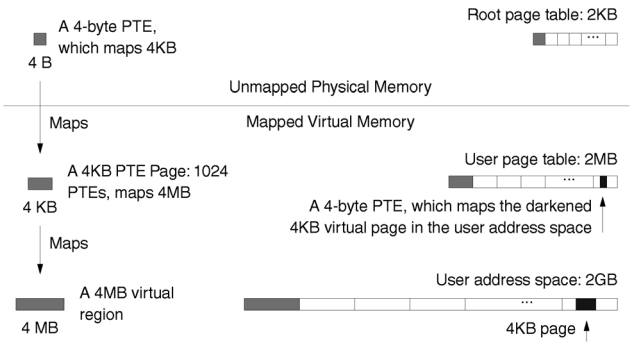

Fig. 4. **The MIPS 32-bit hierarchical page table.** MIPS hardware provides support for a 2 MB linear virtual page table that maps the 2 GB user address space by constructing a virtual address from a faulting virtual address that indexes the mapping PTE in the user page table. This 2 MB page table can easily be mapped by a 2 KB user root page table.

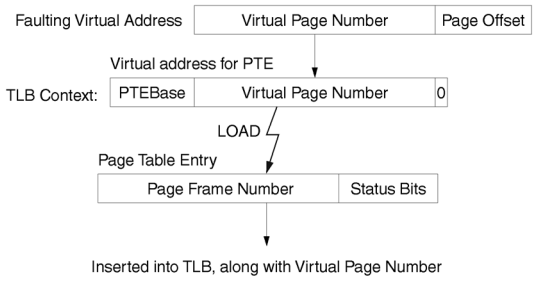

Fig. 5. **The use of the MIPS *TLB Context* register.** The VPN of the faulting virtual address of the mapping PTE. This PTE goes directly into the TLB.

3.4 SPUR: In-Cache Address Translation

The memory-management scheme of Berkeley's SPUR processor (Fig. 6) [32], [45], [46] demonstrated that the storage slots of the TLB are not a necessary component to address translation. The architecture uses a virtually indexed, virtually tagged cache to delay the need for address translation until a cache miss occurs. When a reference misses the cache, a hardware state machine generates a virtual address for the mapping PTE and searches the cache for that address. If this lookup misses, the state machine continues using virtual addresses for higher-level PTEs until the topmost level of the page table is reached, at which point the hardware requests the root PTE from physical memory. The walking of the page table is depicted in Fig. 7. Note its similarity to walking the MIPS table.

The SPUR design eliminates the specialized storage slots of the TLB and instead keeps the page table entries in the virtual cache (note that most operating systems cache the page tables anyway). However, it replaced the TLB with another specialized hardware translation mechanism: the state machine that searches for PTEs in general-purpose storage (the cache) instead of special-purpose storage (TLB slots). Our design is inspired by the SPUR mechanism, but moves this special state machine into software.

3.5 VMP: Software-Controlled Caches

The VMP multiprocessor (Fig. 8) [8] 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 hardware, and it simplifies the cache controller. Our mechanism is inspired by this design and distinguishes itself by eliminating the special buffer holding the cache-fill code (*local memory*), instead keeping all handler code and page-table data in main memory.

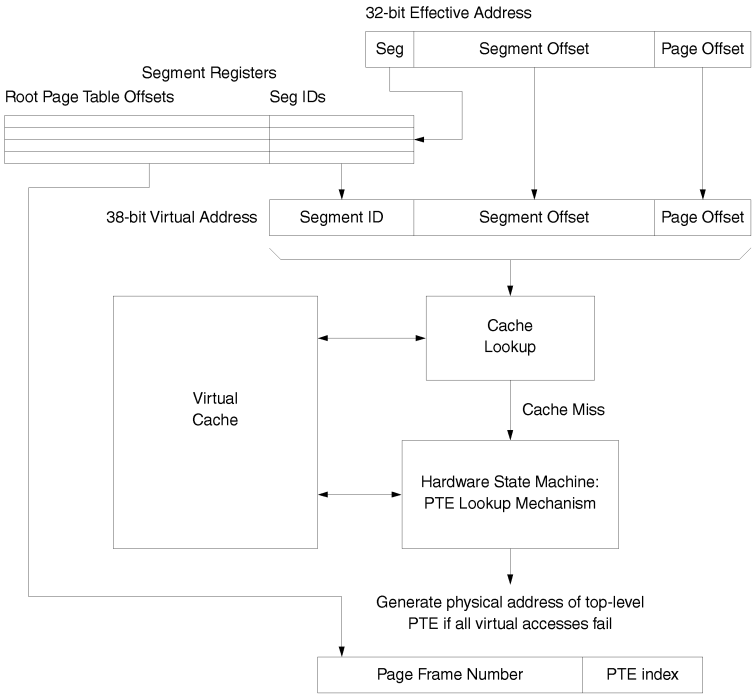

Fig. 6. **The SPUR address translation mechanism.** The SPUR processor was meant to be part of a multiprocessing system. Explicit TLB consistency management was avoided by eliminating the TLB from each processor. Simulation studies showed that this "in-cache translation" was as effective or more effective than a TLB. Like the PowerPC, SPUR used a segmented global space, assigning four 1 GB segments to each user-level address space.

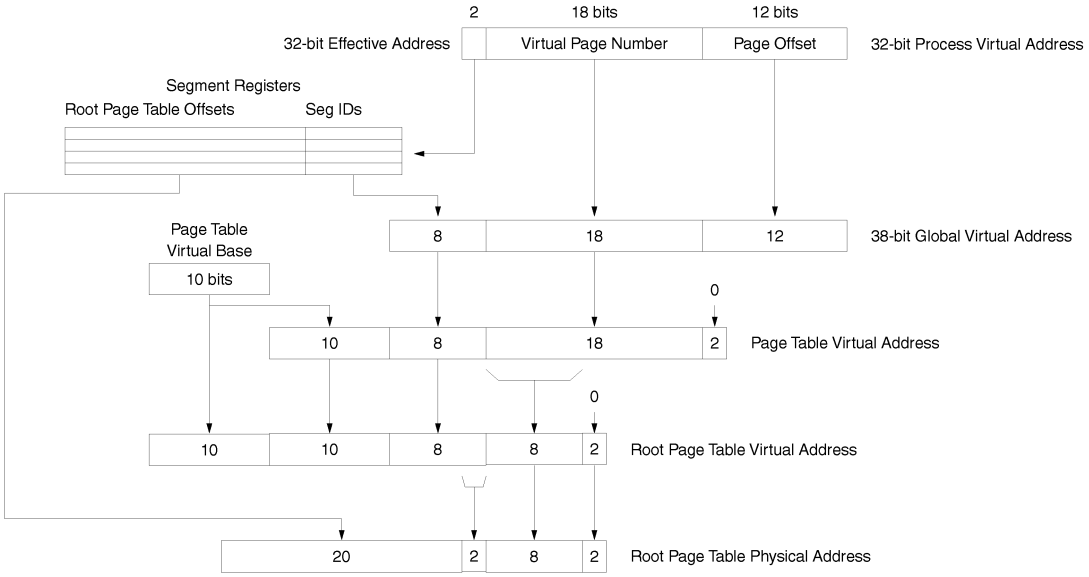

Fig. 7. **The SPUR in-cache translation algorithm.** The SPUR in-cache translation mechanism implements a two-level hierarchical page table where the root level of the table is a subset of the main table. Thus, the two page tables can use the same base address. This translation algorithm is performed by a hardware state machine when a reference misses the virtual cache. Diagram taken from Wood [45].

3.6 Multiprocessor Cache Coherency

In some sense, the software-managed address translation model can be seen as similar to software-managed coherency in multiprocessor cache systems; in both environments, software is responsible for the cache contents. The difference is that the goal in software-managed address translation is simply to translate virtual addresses to physical ones; the fact that the operating system performs cache fill is merely a by-product of this operation. In multiprocessor cache coherency systems, software is responsible for determining when a cached item would become stale with respect to the value in memory and performing corrective actions to prevent errors (block flush, block invalidate, TLB shootdown, etc.).

4 Software-Managed Address Translation

As mentioned earlier, the hardware mechanism explored in this study is simply a virtual cache hierarchy with a software-handled cache miss. When a virtual address fails

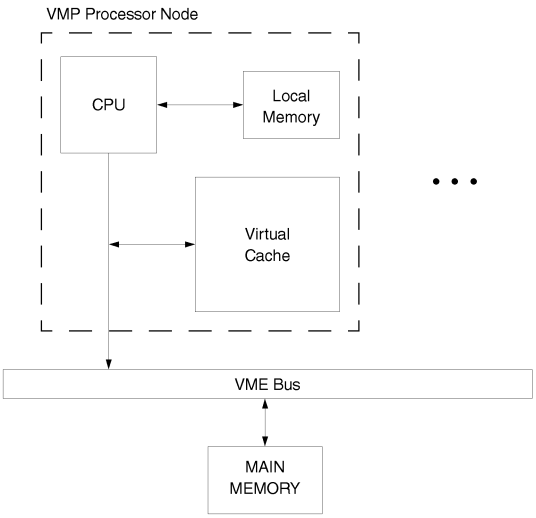

Fig. 8. **The VMP multiprocessor.** Each VMP processor node contains a CPU (68020), local cache outside the namespace of main memory, and a software-controlled virtual cache. The software-controlled cache made multiprocessor cache coherency easiter to explore as the protocols could be rewritten at will.

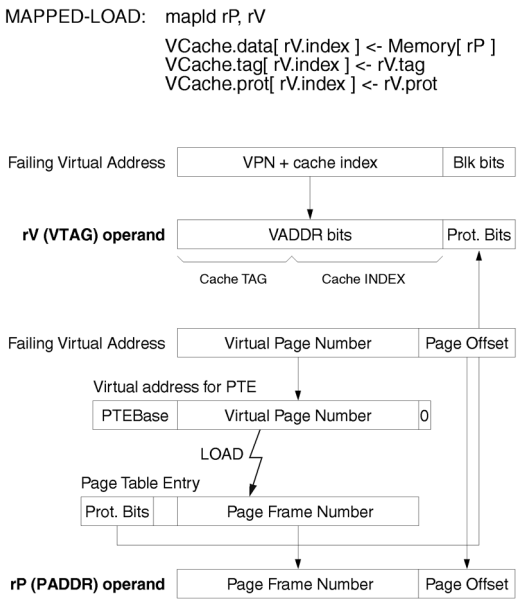

Fig. 9. **Mapped-load instruction.** The top portion gives the syntax and semantics; the bottom of the figure illustrates the source of the rV (VTAG+prot bits) and rP (physical address) operands. The example assumes a MIPS-like page table. Note: *page* = page offset bits.

to hit in the bottommost cache level, a *cache-miss exception* is raised. This invokes the operating system's cache-miss handler, which is responsible for translating the address and fetching the physical data it references. 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 many high performance systems have reasonably large L2 caches, from 256 KB found in even low-end 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 virtual memory is a mechanism to invoke a software cache miss handler 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 [45]. For the cache sizes we are considering, we reach the same conclusion (see Section 5 for details).

4.1 Handling the Cache-Miss Exception

On a cache-miss exception, the hardware saves the program counter of the instruction causing the cache miss and invokes the miss handler. The miss handler loads the data at the failing address on behalf of the user 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. For example, 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 new load instruction in which the operating system specifies a virtual tag and set of protection bits to apply to the incoming data as well as a physical address from which to load the data. The incoming data is inserted into the caches with the specified tag and protection information. This scheme requires a privileged instruction to be added to the instruction set architecture (ISA):[3] *mapped-load*, depicted in Fig. 9.

The mapped-load instruction is simply a load-to-cache instruction that happens to tell the virtual cache explicitly what bits to use for the tag and protection information. It does not load to the register file. Its has two operands, as shown in the following syntax:

```
mapld rP, rV  # register P contains physical
                address
              # register V contains VADDR and
                protection bits
```

The protection bits in rV come from the mapping PTE. The VADDR portion of rV comes from the failing virtual address; it is the topmost bits of the virtual address minus the size of the protection information (which should be no more than the bits needed to identify a byte within a cache block). Both the cache index and cache tag are recovered from VADDR; note that this field is larger than the virtual

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 implementation of this function.

page number. The hardware should not assume *any* overlap between virtual and physical addresses beyond the cache line offset. This is essential to allowing a software-defined page size.

The rP operand can contain a virtual or physical address. The datum identified by the rP operand is obtained from memory (or perhaps the cache itself) and then inserted into the cache at the cache block determined by the VADDR bits 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. To restart the thread, the failing load or store instruction is retried.

4.2 Correctness of Design

The single mapped-load instruction is an atomic operation. The fact that the load is not binding (i.e., no data is brought into the register file) means that it does not actually change the processor state; moreover, protections are not checked until the failing load or store instruction is retried (at which point, it could be found that a protection violation occurred), which simplifies the hardware requirements of the mapped-load instruction. Because the data is not transferred to the register file until a failing load is retried, we do not need different forms of the mapped-load for load word, load halfword, load byte, etc. There is no compiler impact on adding the mapped-load to an instruction set because the instruction is privileged and will not be used by user-level instructions; it will be found only in the cache-miss handler, which will most likely be written in assembly code.

The execution of the cache-miss handler raises some issues. The handler code should be cached for good performance, but it need not be cached—for example, if the instruction-fetch window is large, there is a good chance that the entire handler can be fetched from memory in the time it takes to obtain two cache blocks (the handler is ~10 instructions long). However, if the handler is cached, it could be overwritten if it attempts to bring in requested data that needs to reside in the same cache block. This is typically not a problem in modern architectures in which the instruction fetch and instruction execute are several cycles apart in time: By the time the mapped-load is executed, the handler code is no longer needed. This can, however, be a problem if the mapped-load executes before the trailing instructions in the handler (those that effect a return from exception) are fetched from the instruction cache. There are at least two possible solutions: One is that the cache-miss handler can be aligned so that the mapped-load instruction fits into the same cache block as the final instructions in the handler. This will ensure that the instructions all enter the pipeline before the mapped-load takes effect. However, this is nonportable and susceptible to human error. The second solution would be to alter the semantics of the mapped-load to include a return-from-exception: If it is the last instruction in the handler, then, by the time the mapped-load executes, the handler code is no longer needed in the instruction cache.

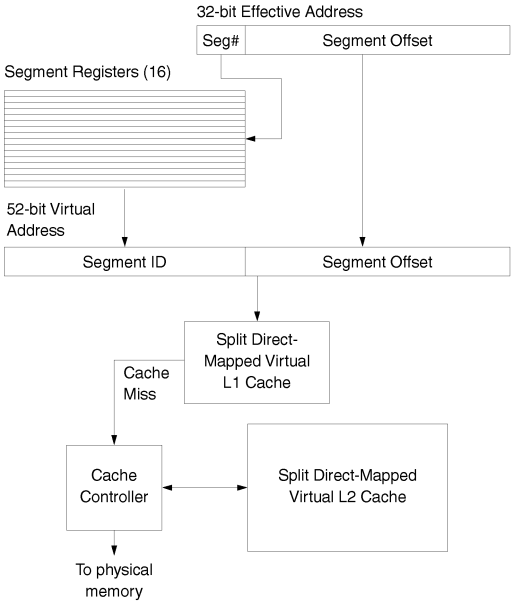

Fig. 10. **The example mechanism.** Segmentation extends a 32-bit user address into a 52-bit global address. The top 20 bits of the global address determine if the address is physical and/or cacheable.

4.3 An Example of *softvm* and Its Use

A PowerPC model of software-managed address translation is shown in Fig. 10, 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. In the analysis, we vary the L1 cache from 2 KB to 256 KB (1K to 128K per side) and the L2 cache from 1 MB to 4 MB.

We assume, for the sake of argument, a 4 GB maximum physical memory. To parallel the MIPS design, the top bits of the virtual address space (in this case, 20 of 52 bits) 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 2 GB of the 4 GB effective address space. Therefore, only the bottom eight of the 16 segment registers are used by applications; the user address space is composed of eight 256 MB virtual segments.

To demonstrate the implementation, 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 2 GB address space. Therefore, a process uses no more than 2 MB 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, 242 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. 11, is a two-

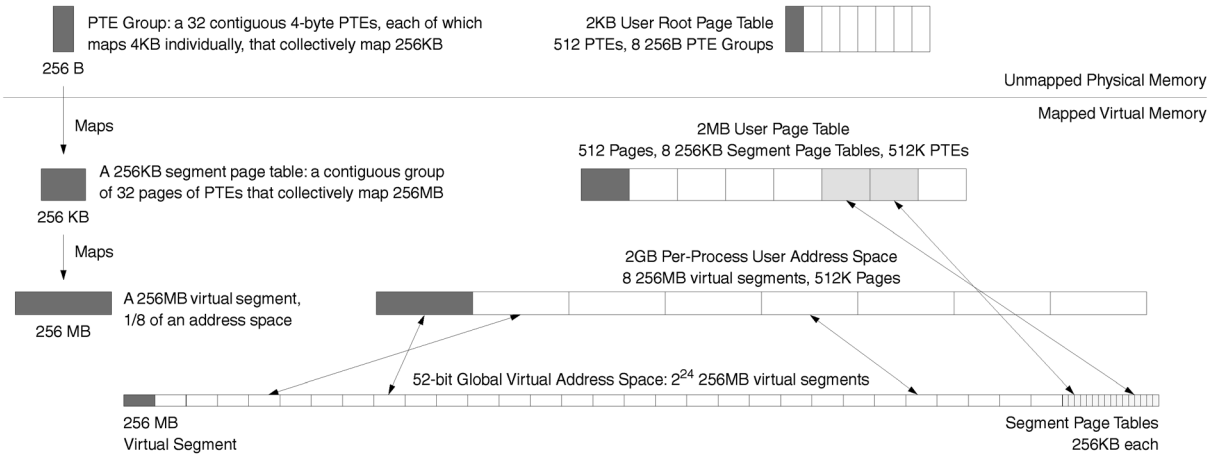

Fig. 11. **An example page table organization based on the PUMA disjunct page table.** There is a single linear page table at the top of the 52-bit address space that maps the entire global space. The 256 KB *Segment Page Tables* that comprise the user page table are taken directly from this global page 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 4 KB pages in the global space.

tiered hierarchy. It is based on the PUMA *disjunct page table* [20], so named because the user page table is a disjunct set of regions in the global space. The lower tier is a 2 MB virtual structure, divided into eight 256 KB segment page tables, each of which (collectively) maps one of the 256 MB 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 2 KB structure wired down in memory while the process is running; it is the bottom half of the process control block. It is divided into eight 256-byte PTE groups, each of which maps a 256 KB segment page table that in turn maps a 256 MB segment. PTE groups must be duplicated across user root page tables to share virtual segments.

We illustrate in Fig. 12 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 (Figs. 5 and 9). The bottom two bits of the address are 0s since the PTE size is four bytes. The top 10 bits of the address are 1s 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 cache-miss 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 1,024 PTEs that would map a 4MB user page table, which would in turn map a 4GB address space. Since the

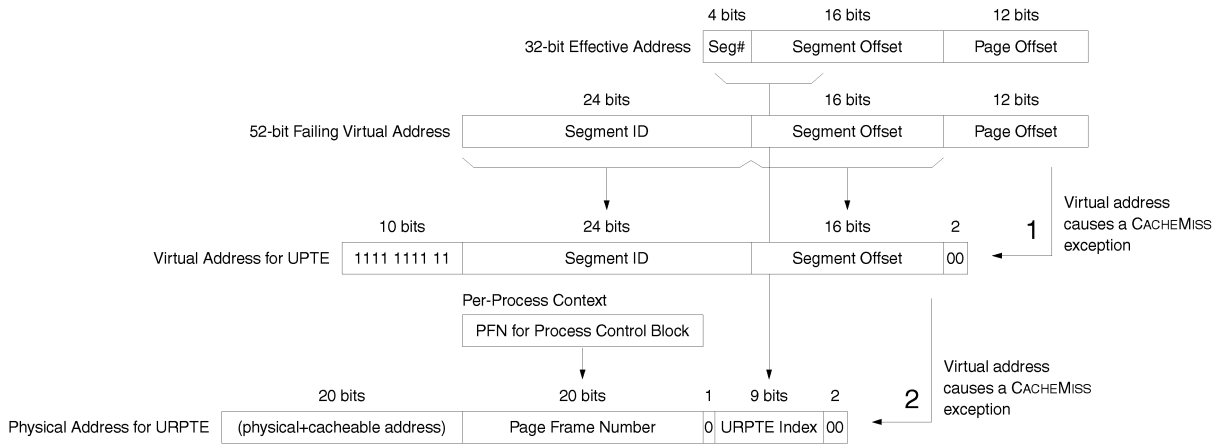

Fig. 12. **An example cache miss algorithm.** Step 1 is the result of a user-level L2 cache miss; the operating system builds a virtual address for a PTE in the global page table. If this PTE is not found in the L1 or L2 cache, a root PTE is loaded, shown in Step 2. One special requirement is a register holding the initial failing address. Another required hardware structure, the per-process context register, points to the process control block of the active process.

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 10 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.

The operating system then performs a mapped-load instruction using the most significant bits of the failing 52-bit address and a physical address 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. A subsequent retry of the failing load or store instruction checks protection bits and, if the protection check is successful, loads the data from the cache into the register file (for a load).

4.4 Memory System Requirements, Revisited

We now revisit the memory management requirements listed earlier and discuss how a software-managed scheme supports them.

Address space protection and large address spaces. These memory management functions are not inherent to software-managed address translation, but a software-managed 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 provides 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.

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 [5], [35] 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) would provide easy support for superpages. A guarded page table [28] 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. [37].

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.

5 Performance

Many studies have shown that significant overhead is spent servicing TLB misses [1], [3], [18], [30], [33]. In particular, Anderson et al. [1] show TLB miss handlers to be among the most commonly executed primitives, Huck and Hays [18] show that TLB miss handling can account for more than 40 percent of total run time, and Rosenblum et al. [33] show that TLB miss handling can account for more than 80 percent of the kernel's computation time. These are extreme cases; typical measurements put TLB handling at 5 to 10 percent of a normal system's run time [3], [6], [30], [36]; this is an apparently acceptable cost that has changed little in 10 years [9], despite significant changes in cache sizes and organizations. The obvious question to ask is: Does the TLB buy us anything? Do its benefits outweigh its cost of management? We now discuss the performance costs of eliminating the TLB.

5.1 First-Order Comparison of Hardware-Oriented and Software-Oriented Designs

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 smallest of the three 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

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

MIPS-style TLB [27] with 64 entries, a random replacement policy, and eight 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 nonzero 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 Memory-Management Simulations

In this study, we compare softvm with Ultrix and Mach and simulate the memory-management performance of all three virtual memory systems. From our previous study in software-managed address translation [21], we choose to look at the two worst-performing benchmarks: gcc and vortex, as well as winword, a benchmark from the Etch suite [14], as Etch benchmarks tend to have larger footprints than SPEC benchmarks. We simulate the overheads of managing L2 cache misses in software for the softvm design and the overheads of managing TLB misses in software for Ultrix and Mach. The following sections describe the Ultrix and Mach simulations; more detail on the models can be found in [22].

5.2.1 Ultrix Memory-Management Simulation

The Ultrix page table as implemented on the MIPS processor is a two-tiered table [30]. The 2 GB user address space is mapped by a 2 MB linear table (the user page table) in virtual kernel space, which is in turn mapped by a 2 KB array of PTEs (the root page table). It requires at most two memory references to find the appropriate mapping information. The TLB (simulated as 256-entry, split into 128-entry fully-associative I-TLB and 128-entry fully-associative D-TLB; each TLB has 16 protected lower slots to hold kernel-level mappings) provides protection information; if the TLB misses on a reference, the page table is walked before the cache lookup can proceed. The TLB miss handler has two interrupt entry points: one for user-level misses, one for kernel-level misses. The handlers are located in unmapped space, so executing them cannot cause I-TLB misses. The user-level handler is 10 instructions long, the root-level handler is 20. The start of the handler code is page-aligned.

5.2.2 Mach Memory-Management Simulation

The Mach page table as implemented on the MIPS processor is a three-tiered table [30], [3]. The 2 MB user page tables are located in kernel space, the entire 4 GB kernel space is mapped by a 4 MB kernel structure (the kernel page table), which is in turn mapped by a 4 KB kernel structure (the root page table). It requires at most three memory references to find the appropriate mapping information. The Mach TLB-miss handler on actual MIPS hardware is comprised of two main interrupt paths. There is a dedicated interrupt vector for user-level misses (those in the bottom half of the 4GB address space) and all other TLB misses go through the general interrupt mechanism. This general-purpose vector contains a large amount of administrative

TABLE 2
Components of VMCPI

Tag	Cost per	Description
uhandler	variable	A TLB miss (or an L2 cache miss in the case of a SOFTVM simulation) that occurs during application-level processing invokes the user-level miss handler
upte-L2	20 cycles	The UPTE lookup during the user-level handler misses the L1 data cache; reference goes to the L2 data cache
upte-MEM	500 cycles	The UPTE lookup during the user-level handler misses the L2 data cache; reference goes to main memory
khandler	variable	A TLB miss that occurs during the user-level miss handler invokes the kernel-level miss handler
kpte-L2	20 cycles	The KPTE lookup during the kernel-level handler misses the L1 data cache; reference goes to the L2 data cache
kpte-MEM	500 cycles	The KPTE lookup during the kernel-level handler misses the L2 data cache; reference goes to main memory
rhandler	variable	A TLB miss (or an L2 cache miss in the case of a SOFTVM simulation) that occurs during the user-level or kernel-level miss handler invokes the root-level miss handler
rpte-L2	20 cycles	The RPTE lookup during the root-level handler misses the L1 data cache; reference goes to the L2 data cache
rpte-MEM	500 cycles	The RPTE lookup during the root-level handler misses the L2 data cache; reference goes to main memory
handler-L2	20 cycles	During execution of a miss handler, code misses the L1 instruction cache; reference goes to L2 instruction cache
handler-MEM	500 cycles	During execution of a miss handler, code misses the L2 instruction cache; reference goes to main memory

code that adds an enormous cost to interrupts that cannot be handled by the dedicated vectors. Like the Ultrix simulation, the handlers for the first two tiers of the page table are 10 and 20 instructions long, respectively. Root-level misses take a long path of 500 instructions and perform a number of additional loads to simulate the effect of administrative code. The handlers are located in unmapped space (executing them cannot cause I-TLB misses).

5.3 Description of Statistics Gathered

This study uses trace-driven simulation to measure virtual memory overhead. The unit of measurement we use is cycles per instruction (CPI), calculated as execution cycles divided by the number of user-level instructions. This is a direct measure of performance, given that the number of user-level instructions is constant and the processor cycle time will remain constant for different VM simulations. Our definition of VMCPI is the number of execution cycles imposed by the VM system divided by user-level instructions and so represents the additional burden of the virtual memory system on top of program execution. When simulating TLB-miss handlers, the contents of the I-caches are overwritten with the handler code and PTE loads overwrite the D-caches. VMCPI is, therefore, that portion of CPI attributable to management of the TLB or cache.

VMCPI is subdivided into the categories shown in Table 2. The *uhandler*, *khandler*, and *rhandler* components refer to the cost of executing the instructions that make up the TLB-miss handlers for the different levels of the page table. The *upte-L2* and *upte-MEM* components represent instances where the PTE in the user-level page table is not in the L1 data-cache and has to go to either the L2 data cache or main memory, respectively. The analogous *kpte-* and *rpte-* components correspond to PTE loads for the kernel- and root-level page tables, respectively. The *handler-L2* and *handler-MEM* components refer to the cost of instruction cache misses while executing TLB-miss handlers. Note that not all of the categories apply to all simulations; for instance, the *softvm* and Ultrix simulations have no kernel-level miss handlers (*khandler*, *kpte-L2*, and *kpte-MEM* events will not happen). The costs of the page-table accesses that can occur in each of the simulations (labeled variable in Table 2) are summarized in Table 3.

As an example, Table 4 gives a detailed breakdown of costs for one of the benchmarks: GCC with 8K/8K L1 caches and 512K/512K L2 caches. Our example miss handler from the previous section requires 10 instructions, including one PTE load. It is very similar to the MIPS TLB refill handler that requires less than 10 instructions, including one PTE load, taking 10 cycles when the load

TABLE 3
Simulated Page-Table Events

VM Sim	User Handler	Kernel Handler	Root Handler
ULTRIX	10 instrs, 1 PTE load	n.a.	20 instrs, 1 PTE load
MACH	10 instrs, 1 PTE load	20 instrs, 1 PTE load	500 instrs, 10 “admin” loads + 1 PTE load
SOFTVM	10 instrs, 1 PTE load	n.a.	20 instrs, 1 PTE load

TABLE 4
Breakdown of GCC Overhead

Event	VMCPI Component	Frequency (per instr.)	Penalty per Occurrence	Overhead (CPI)
L2 D-Cache Ld/St miss	uhandler	0.0021	10 cycles (i-execute)	0.0210
L2 I-Cache I-fetch miss	uhandler	0.0019	10 cycles (i-execute)	0.0194
Miss handler L1 D-miss	upte-L2	0.0006	20 cycles (mem stall)	0.0126
Miss handler L2 D-miss	upte-MEM	0.0001	500 cycles (mem stall)	0.0427
Miss handler L1 I-miss	handler-L2	0.0007	20 cycles (mem stall)	0.0145
Miss handler L2 I-miss	handler-MEM	< 0.0001	500 cycles (mem stall)	0.0085
Miss handler Recursion	rhandler	0.0001	20 cycles (i-execute)	0.0017
Recursive L1 D-miss	rpte-L2	0.0001	20 cycles (mem stall)	0.0014
Recursive L2 D-miss	rpte-MEM	< 0.0001	500 cycles (mem stall)	0.0004
			Total CPI:	0.1222

hits in the cache or 40+ when the load misses in the cache, thereby forcing the reference to main memory [3]. In our model, the L2 cache miss handler always takes 10 cycles and runs whenever we take an L2 cache miss while executing in user mode (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 or the address is physical and goes straight through to main memory (*Miss handler L2 D-miss*, 500 cycles). 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 500-cycle penalty to reference physical memory. Cache misses that cause recursive invocations of the cache-miss handler are represented by the *Cache-miss Recursion*, *Recursive L1 D-miss*, and *Recursive L2 D-miss* components. These represent the handler execution, PTE loads that miss the L1 cache, and PTE loads that miss the L2 cache, respectively. Instruction-cache misses for recursive handlers are accounted for in the components described earlier: *Miss-handler L1/L2 I-miss*.

The cost of interrupts is not included here because it is dependent on pipeline organization and is therefore orthogonal to the mechanisms we have been describing; however, we have covered it in detail (including presentation of experimental results) elsewhere [22]. That study shows the interrupt overhead, if the number of instructions flushed per interrupt is high, can be on a par with the overhead of managing the TLB or cache.

We assume the memory system is large enough to hold all pages used by an application and all pages required to hold the page tables. To put the organizations on an even footing, we ignore the cost of initializing the process address space. This includes demand-paging data from disk and initializing the page tables; a realistic measurement is likely to be extremely dependent on implementation; therefore, this cost is not factored into the simulations or the measurements given. It is also likely to be the same for all virtual memory mechanisms and, therefore, its inclusion would simply serve to blur performance distinctions between the mechanisms. Though our measurements do include the cold-start cost of warming the caches, they do not include the cost of initializing the page table entries as this overhead would be identical in every simulation. Our measurements are intended to highlight only the differences between the page table organizations, the TLB implementations (hardware- or software-managed), and the presence or absence of memory-management hardware. Thus, the only part of the OS that is simulated is the TLB-refill mechanism.

5.4 Results: VMCPI as a Function of Cache Organization

We present the VMCPI overheads as a function of L1 and L2 cache sizes and linesizes, based on three applications executing on three different operating systems and instruction-set architectures. The benchmark/platform combinations are GCC (a C compiler) executing on a Digital Unix/Alpha machine, VORTEX (a database) executing on an IBM AIX/PowerPC machine, and WINWORD (a word processor) executing on a Microsoft Windows/x86 machine. GCC and VORTEX are SPEC benchmarks, and the traces were generated in-house; WINWORD is in the Etch suite of benchmark traces [14]. The L2 instruction and data-cache misses per 1,000 user instructions are given in Table 5 for these benchmarks. The numbers represent L2 caches with 32-byte linesizes.

TABLE 5
Level-2 Cache Misses per 1,000 Instructions

Benchmark	L2 I/I cache size	L2 icache misses per 1000 instrs	L2 dcache misses per 1000 instrs
GCC/Alpha	512K/512K	2.10	1.94
	1024K/1024K	1.37	1.88
	2048K/2048K	1.18	1.85
VORTEX/PowerPC	512K/512K	1.10	8.80
	1024K/1024K	0.83	7.96
	2048K/2048K	0.71	7.54
WINWORD/x86	512K/512K	1.01	6.61
	1024K/1024K	0.41	5.84
	2048K/2048K	0.19	5.78

In general, the number of L2 misses per 1,000 instructions tended to be between 1 and 10 for the range of cache organizations simulated. This means that, for every 1,000 user instructions executed, roughly five of them caused the *softvm* cache-miss handler to execute, resulting in the execution of 10-30 instructions, part of the instruction caches being overwritten by handler instructions, and part of the data caches being overwritten by page-table entries. All told, this represents anywhere from 10 cycles to several hundred cycles in VM overhead per L2 cache miss, depending on the number of cache misses incurred by the handler. Measured execution times for the cache-miss handler averaged between 10 and 40 cycles per invocation. This translates to 50-200 cycles per 1,000 user instructions, assuming five L2 misses per 1,000 instructions. As the graphs will show, the bulk of the *softvm* measurements were in the range of 10-200 VM cycles per 1,000 user instructions and, given appropriate cache configurations, were comparable in scale to overheads measured for both ULTRIX and MACH. Recall that, for these latter simulations, the sizes of the TLBs were modeled as 128 fully associative entries per side (128-entry ITLB, 128-entry DTLB), which is on the aggressive end of today's TLB sizes.

Fig. 13 gives the VMCPI results for GCC, Fig. 14, and Fig. 15 gives the results for WINWORD. We note several things. First, the overheads are in the right ballpark to represent a 5-10 percent overhead for a 1 CPI machine, even without considering address space and page table initialization, paging, I/O, etc. Recall that typical virtual memory systems exact a run-time overhead of 5-10 percent in TLB management [3], [6], [30], [36] (the figure does not including paging, I/O, etc.). We conclude that these results are appropriate for our stated target domain of desktop-class workstations. Second, for the Alpha and PowerPC studies (GCC and VORTEX), the ULTRIX and MACH virtual memory systems have surprisingly similar overheads, despite the extremely high cost of managing the root-level table in the MACH simulation. This would suggest that even fairly complicated page tables can have low overheads provided that the common-case portions of the structure are efficient. However, given the factor-of-two difference in performance between ULTRIX and MACH on the WINWORD benchmark, we have to conclude that this result is benchmark-dependent. Remember that the only difference between ULTRIX and MACH VM organizations is the addition of an expensive third level of the page table in MACH. Though WINWORD misses the L2 cache just as often as VORTEX, it evidently requires significant management at this third level in the page table, judging from the differences in VMCPI overhead. This indicates a larger, more sparsely populated address space. Third, the software-managed cache (*softvm*) tends to do about as well as the other schemes once the L2 cache is large enough and once a suitable linesize is chosen ($\text{L2 linesize} \geq 64$ bytes); however, its performance tends to be more sensitive to choices of linesize and cache size than the other virtual memory organizations. This is not surprising; the software-oriented scheme places a much larger dependence on the cache system than the other virtual memory organizations, which are much more dependent on the performance of the TLBs. We note that decreasing the TLB sizes to 64 entries per side increases the overhead of the hardware schemes by roughly an order of magnitude, while the software-managed scheme would be unaffected. Last, we note that the curves for the TLB-based schemes are roughly grouped by L1 linesize: For small L1 caches, linesize is more important than cache size. In contrast, the softvm curves are roughly grouped by L2 linesize: For this scheme, L2 linesize is more important than L1 cache size.

Besides the familiar signature of diminishing returns from increasing linesize (e.g., for GCC and VORTEX, the largest overheads in the 1 MB *softvm* results are from the smallest and largest linesizes), the *softvm* results show that cache size clearly has a significant impact on the overhead of the system. Overhead decreases by between a factor of two and an order of magnitude when the L2 cache is doubled and decreases by between a factor of two and a factor of five as the L1 cache increases from 1 KB to 128 KB (2 KB to 256 KB total L1 cache size). Within a given cache size, linesize choice can affect performance by a factor of two or more (up to an order of magnitude for some configurations).

In general, the software-managed scheme performs similarly to the other schemes and the fact that we see similar trends across three very different OS/hardware platforms is encouraging.

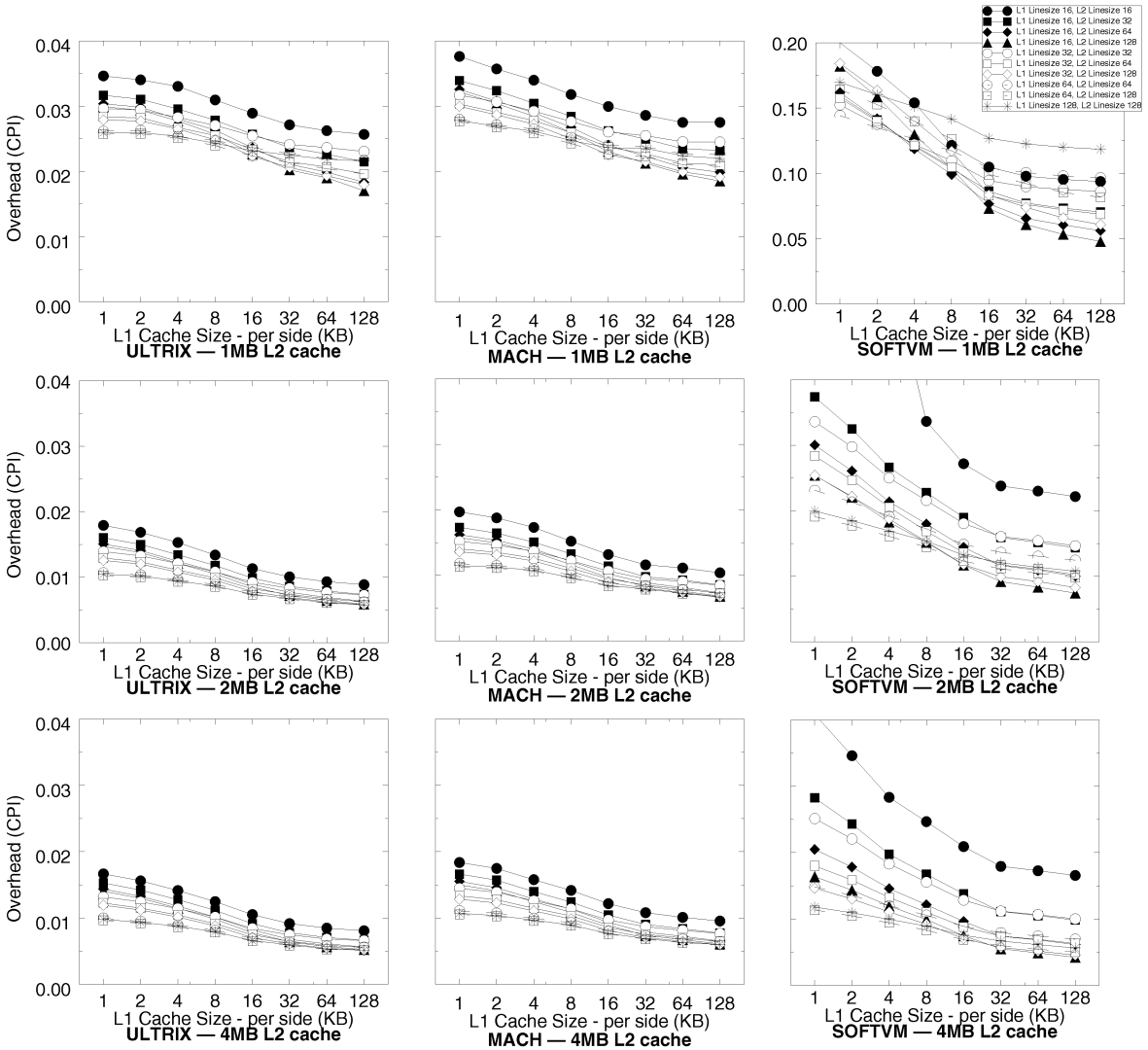

Fig. 13. **VMCPI vs. L1 and L2 cache size and linesize—GCC.** These are the VMCPI totals for each of the VM simulations. This overhead represents only the cost of walking the page table and refilling the TLB (or, in the case of the SOFTVM simulation, filling a cache block). Each data point represents one run of the simulator; each curve represents a different L1/L2 linesize configuration. Note that the scale differs for the SOFTVM graph with 1 MB L2 cache size.

5.5 Handling Writes

We discuss two options in handling writes: One can either use a software-managed translation scheme with a write-back cache or a write-through cache.

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 cache-miss 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. A hardware-intensive solution to the problem is to 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 in conjunction with a writeback cache, because it is not necessary. If writebacks happen in 50 percent of all cache misses, then 50 percent 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 overhead (VMCPI) by roughly 50 percent. 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.

To implement stores to a write-through cache, there must be some mechanism that translates the virtual reference to a physical address. This can be done if the translation for each cache line is held in the cache, as mentioned earlier. This requires more hardware, but does not impose excess performance overhead on the mechanism. However, it does raise the issue of having potentially

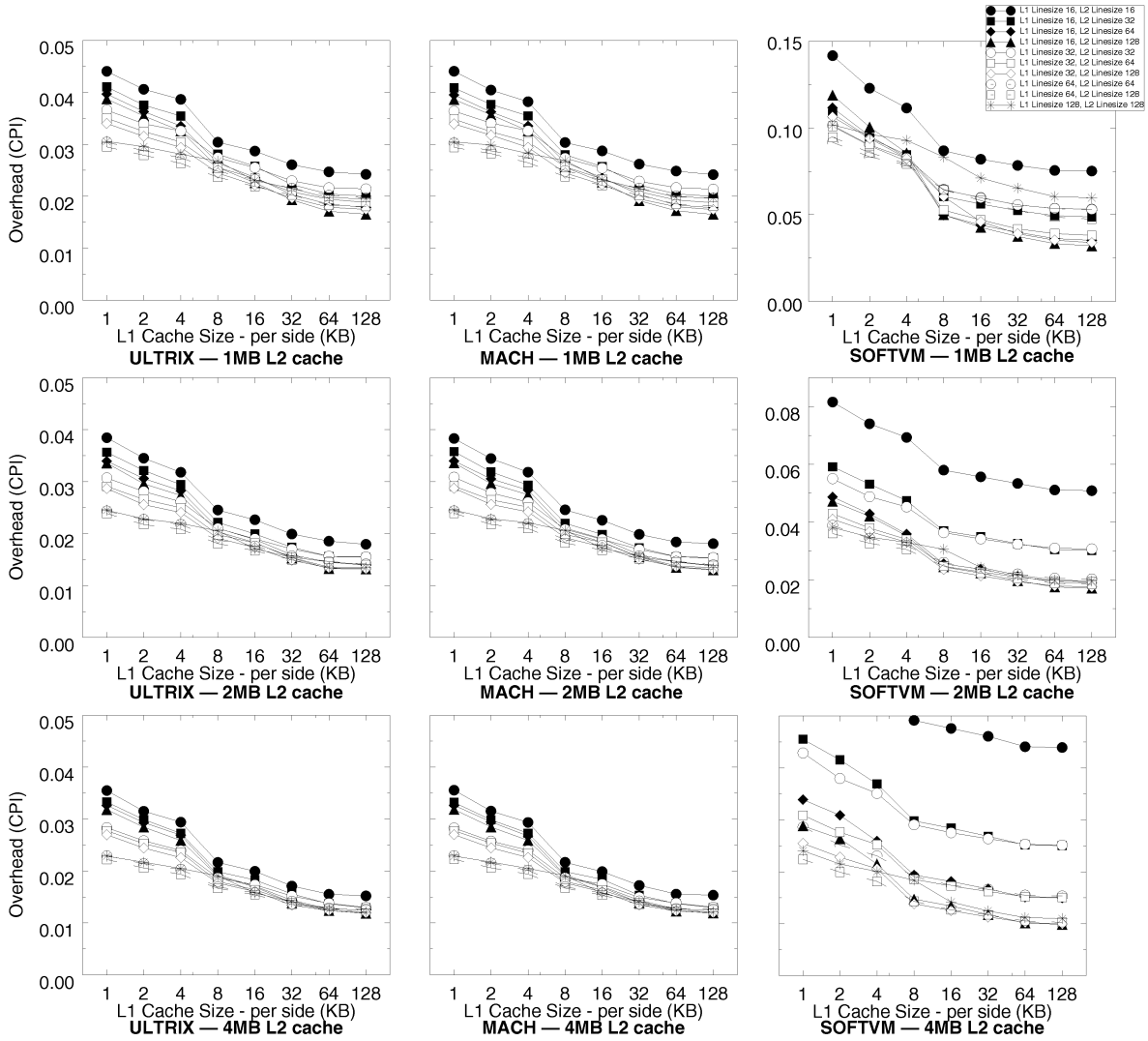

Fig. 14. **VMCPI vs. L1 and L2 cache size and linesize—VORTEX.** These are the VMCPI totals for each of the VM simulations. This overhead represents only the cost of walking the page table and refilling the TLB (or, in the case of the SOFTVM simulation, filling a cache block). Each data point represents one run of the simulation; each curve represents a different L1/L2 linesize configuration. Note that the scale differs for the SOFTVM graphs with 1 MB and 2MB L2 cache sizes.

stale translation information in the cache. Keeping this translation information consistent is the subject of the next section, which looks at the problem of maintaining protection-bit consistency—very similar to maintaining translation consistency if the translation for each cache line is stored with that cache line.

5.6 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 [30] and the operating system is Mach3. We chose Mach as it uses copy-on-write liberally, producing 1,000 times the page-protection modifications seen in Ultrix [30]. 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 6.

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 4 KB pages and a 16-byte linesize (smaller linesizes give more conservative results), we must check 256 cache lines per modification. Assuming an average of 10 L1 cache lines and 50 L2 caches lines affected per

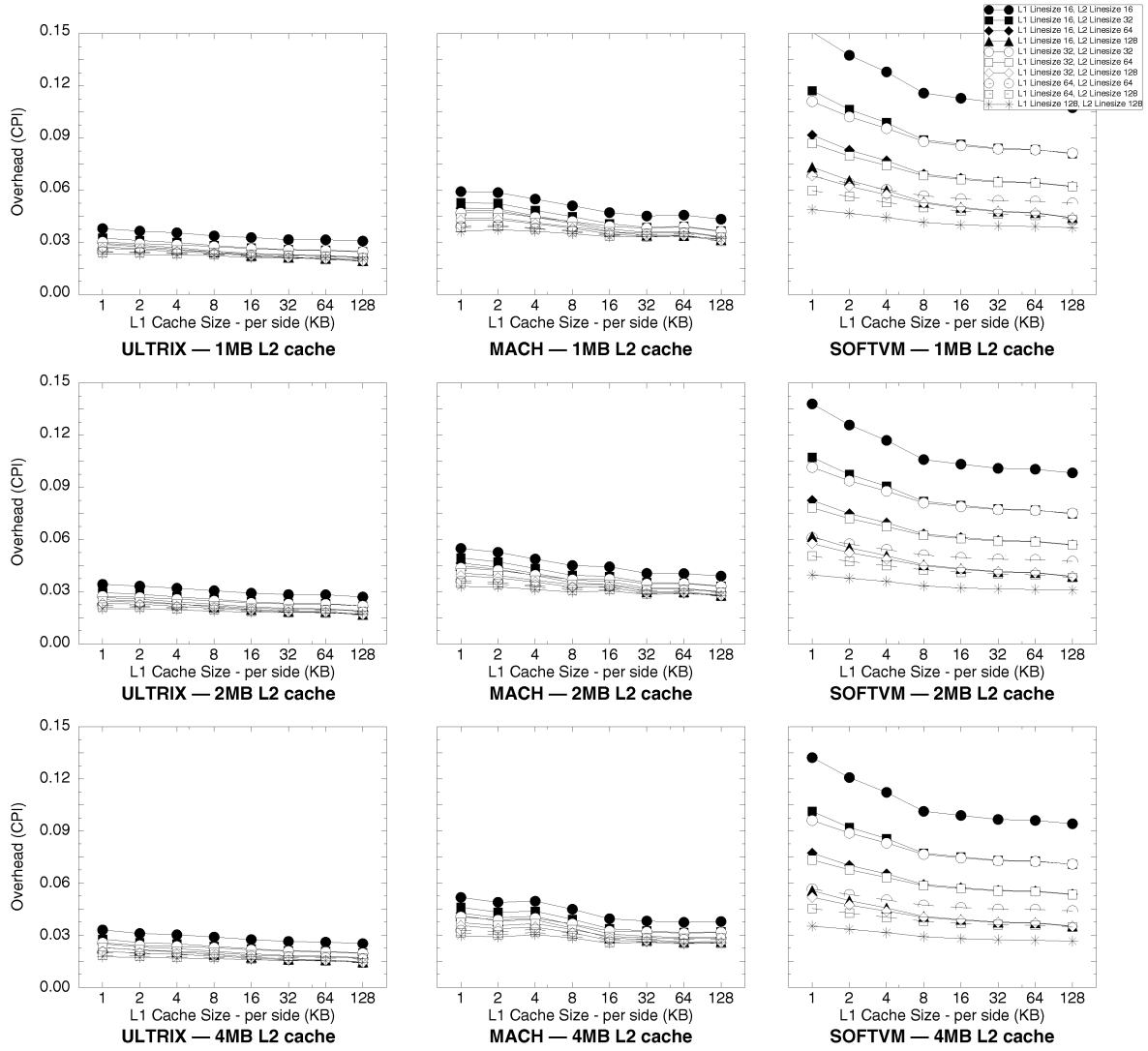

Fig. 15. **VMCPI vs. L1 and L2 cache size and linesize—WINWORD.** These are the VMCPI totals for each of the VM simulations. This overhead represents only the cost of walking the page table and refilling the TLB (or, in the case of the SOFTVM simulation, filling a cache block). Each data point represents one run of the simulator; each curve represents a different L1/L2 linesize configuration. All graphs use the same y-axis scale.

modification,[4] if L1 cache lines can be checked in three cycles and updated in five 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 (five cycles), the remaining 246 need only be checked (three 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 6,908 cycles per page-protection modification ($10 * 5 + 246 * 3 + 50 * 40 + 206 * 20$). This yields between 0.019 and 0.164 CPI ($6,908 * 2.8 * 10^{-6}$ and $6,908 * 23.8 * 10^{-6}$). This should translate to a worst case of 2-7 percent 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 percent execution time.

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.

TABLE 6
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

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.

5.7 Optimizations

As suggested earlier, there are numerous optimizations that can be applied to this scheme, from lazy update of protection bits to careful placement of the handler code in physical memory so that it occupies a minimal number of cache blocks. In addition, related work can be drawn upon; for example, the software TLB explored by Bala et al. [3] is a PTE cache that holds the most recently referenced PTEs in a buffer in main memory, thus condensing useful data in a manner that can better utilize cache blocks and effectively fit more PTEs into the cache. This has the potential to lower access time to the page table on average. The Tempest group developed tricks in the compiler and operating system that reduced translation-related interrupt invocations by recognizing when the same mapping or the same data would be valid from one access to the next [34]. The privileged-mode-bit optimization by Henry reduces the need to flush the pipeline on taking a precise interrupt by adding a mode bit to every pipeline register, as opposed to using one global bit [17]. This would reduce the overhead of every L2 cache miss by several cycles. Last, one could handle the L2 cache in user mode directly, obviating the need to vector into the kernel at all—hardware and software support for such activity has been investigated by Thekkath and Levy (user-level interrupt handling) [39] and the Exokernel group (user-level virtual memory) [13].

6 Summary

For the design of a 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, achieving similar performance compared to TLB-based systems with very low-overhead designs. Therefore, software-managed address translation is a viable strategy for high-end computing today, achieving excellent performance with less hardware.

The benefits of a minimal hardware design are four-fold. 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, eliminating the TLB has the potential to reduce power consumption significantly [26]. Third, a reduction in hardware will leave room for more memory structures, perhaps helping to increase performance. Last, simpler hardware should be easier to design and debug, cutting down on development time.

Acknowledgments

The authors would like to thank the anonymous reviewers for helping to clarify points, improve descriptions of the mechanism, and point out related references.

Bruce Jacob is supported in part by the US National Science Foundation (NSF under contract EIA-9806645 and in part by an NSF CAREER Award, contract CCR-9983618. Trevor Mudge is partially supported by the US Defense Advanced Research Projects Agency under DARPA/ARO Contract Number DAAH04-94-G-0327.

References

[1] T.E. Anderson, H.M. Levy, B.N. Bershad, and E.D. Lazowska, "The Interaction of Architecture and Operating System Design," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '91),* pp. 108-120, Apr. 1991.

[2] A.W. Appel and K. Li, "Virtual Memory Primitives for User Programs," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '91),* pp. 96-107, Apr. 1991.

[3] K. Bala, M.F. Kaashoek, and W.E. Weihl, "Software Prefetching and Caching for Translation Lookaside Buffers," *Proc. First USENIX Symp. Operating Systems Design and Implementation (OSDI '94),* pp. 243-253, Nov. 1994.

[4] A. Chang and M.F. Mergen, "801 Storage: Architecture and Programming," *ACM Trans. Computer Systems,* vol. 6, no. 1, Feb. 1988.

[5] J.S. Chase, H.M. Levy, E.D. Lazowska, and M. Baker-Harvey, "Lightweight Shared Objects in a 64-Bit Operating System," Technical Report 92-03-09, Univ. of Washington, Mar. 1992.

[6] J.B. Chen, A. Borg, and N.P. Jouppi, "A Simulation Based Study of TLB Performance," *Proc. 19th Ann. Int'l Symp. Computer Architecture (ISCA '92),* May 1992.

[7] R. Cheng, "Virtual Address Cache in UNIX," *Proc. Summer 1987 USENIX Technical Conf.,* June 1987.

[8] D.R. Cheriton, G.A. Slavenburg, and P.D. Boyle, "Software-Controlled Caches in the VMP Multiprocessor," *Proc. 13th Ann. Int'l Symp. Computer Architecture (ISCA '86),* Jan. 1986.

[9] D.W. Clark and J.S. Emer, "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement," *ACM Trans. Computer Systems,* vol. 3, no. 1, pp. 31-62, Feb. 1985.

[10] H. Deitel, *Inside OS/2.* Reading, Mass.: Addison-Wesley, 1990.

[11] K. Diefendorff, "Transmeta Unveils Crusoe: Supersecret Startup Attacks Mobile Market with VLIW, Code Morphing," *Microprocessor Report,* vol. 14, no. 1, pp. 15-16, Jan. 2000.

[12] Digital Equipment Corp., *DECchip 21064 and DECchip 21064A Alpha AXP Microprocessors Hardware Reference Manual.* Maynard, Mass.: Digital Equipment Corp., 1994.

[13] D.R. Engler, S.K. Gupta, and M.F. Kaashoek, "AVM: Application-Level Virtual Memory," *Proc. Fifth Workshop Hot Topics in Operating Systems (HotOS-V),* May 1995.

[14] Etch, *Memory System Research at the University of Washington.* Univ. of Washington, http://etch.cs.washington.edu/, 1998.

[15] J.R. Goodman, "Coherency for Multiprocessor Virtual Address Caches," *Proc. Second Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '87),* pp. 72-81, Oct. 1987.

[16] L. Gwennap, "Intel's P6 Uses Decoupled Superscalar Design," *Microprocessor Report,* vol. 9, no. 2, Feb. 1995.

[17] D.S. Henry, "Adding Fast Interrupts to Superscalar Processors," Technical Report Memo-366, MIT Computation Structures Group, Dec. 1994.

[18] J. Huck and J. Hays, "Architectural Support for Translation Table Management in Large Address Space Machines," *Proc. 20th Ann. Int'l Symp. Computer Architecture (ISCA '93),* pp. 39-50, May 1993.

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

[20] B.L. Jacob and T.N. Mudge, "Specification of the PUMA Memory Management Design," Technical Report CSE-TR-314-96, Univ. of Michigan, Aug. 1996.

[21] B.L. Jacob and T.N. Mudge, "Software-Managed Address Translation," *Proc. Third Int'l Symp. High Performance Computer Architecture (HPCA '97),* pp. 156-167, Feb. 1997.
[22] B.L. Jacob and T.N. Mudge, "A Look at Several Memory-Management Units, TLB-Refill Mechanisms, and Page Table Organizations," *Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '98),* pp. 295-306, Oct. 1998.
[23] B.L. Jacob and T.N. Mudge, "Virtual Memory in Contemporary Microprocessors," *IEEE Micro,* vol. 18, no. 4, pp. 60-75, July/Aug. 1998.
[24] B.L. Jacob and T.N. Mudge, "Virtual Memory: Issues of Implementation," *Computer,* vol. 31, no. 6, pp. 33-43, June 1998.
[25] B.L. Jacob, "Software-Oriented Memory-Management Design," PhD thesis, Univ. of Michigan, July 1997.
[26] T. Juan, T. Lang, and J.J. Navarro, "Reducing TLB Power Requirements," *Proc. 1997 IEEE Int'l Symp. Low Power Electronics and Design (ISLPED '97),* pp. 196-201, Aug. 1997.
[27] G. Kane and J. Heinrich, *MIPS RISC Architecture.* Englewood Cliffs, N.J.: Prentice Hall, 1992.
[28] J. Liedtke and K. Elphinstone, "Guarded Page Tables on MIPS R4600," *ACM Operating Systems Review,* vol. 30, no. 1, pp. 4-15, Jan. 1996.
[29] *The PowerPC Architecture: A Specification for a New Family of RISC Processors,* C. May, E. Silha, R. Simpson, and H. Warren, eds. San Francisco: Morgan Kaufmann, 1994.
[30] D. Nagle, R. Uhlig, T. Stanley, S. Sechrest, T. Mudge, and R. Brown, "Design Tradeoffs for Software-Managed TLBs," *Proc. 20th Ann. Int'l Symp. Computer Architecture (ISCA '93),* May 1993.
[31] 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 Trans. Computers,* vol. 37, no. 8, pp. 896-908, Aug. 1988.
[32] S.A. Ritchie, "TLB for Free: In-Cache Address Translation for a Multiprocessor Workstation," Technical Report UCB/CSD 85/233, Univ. of California, May 1985.
[33] M. Rosenblum, E. Bugnion, S.A. Herrod, E. Witchel, and A. Gupta, "The Impact of Architectural Trends on Operating System Performance," *Proc. 15th ACM Symp. Operating Systems Principles (SOSP '95),* Dec. 1995.
[34] I. Schoinas, B. Falsafi, A.R. Lebeck, S.K. Reinhardt, J.R. Larus, and D.A. Wood, "Fine-Grain Access Control for Distributed Shared Memory," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '94),* pp. 297-306, Oct. 1994.
[35] M.L. Scott, T.J. LeBlanc, and B.D. Marsh, "Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System," *Proc. 1988 Int'l Conf. Parallel Processing,* Aug. 1988.
[36] M. Talluri and M.D. Hill, "Surpassing the TLB Performance of Superpages with Less Operating System Support," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '94),* pp. 171-182, Oct. 1994.
[37] M. Talluri, M.D. Hill, and Y.A. Khalidi, "A New Page Table for 64-Bit Address Spaces," *Proc. 15th ACM Symp. Operating Systems Principles (SOSP '95),* Dec. 1995.
[38] M. Talluri, S. Kong, M.D. Hill, and D.A. Patterson, "Tradeoffs in Supporting Two Page Sizes," *Proc. 19th Ann. Int'l Symp. Computer Architecture (ISCA '92),* pp. 415-424, May 1992.
[39] C.A. Thekkath and H.M. Levy, "Hardware and Software Support for Efficient Exception Handling," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '94),* pp. 110-119, Oct. 1994.
[40] S.-Y. Tzou and D.P. Anderson, "The Performance of Message-Passing Using Restricted Virtual Memory Remapping," *Software—Practice and Experience,* vol. 21, no. 3, pp. 251-267, Mar. 1991.
[41] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham, "Efficient Software-Based Fault Isolation," *Proc. 14th ACM Symp. Operating Systems Principles (SOSP '93),* pp. 203-216, Dec. 1993.
[42] W.-H. Wang, J.-L. Baer, and H.M. Levy, "Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy," *Proc. 16th Ann. Int'l Symp. Computer Architecture (ISCA '89),* pp. 140-148, June 1989.
[43] S. Weiss and J.E. Smith, *POWER and PowerPC.* San Francisco: Morgan Kaufmann, 1994.
[44] B. Wheeler and B.N. Bershad, "Consistency Management for Virtually Indexed Caches," *Proc. Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '92),* pp. 124-136, Oct. 1992.
[45] D.A. Wood, "The Design and Evaluation of In-Cache Address Translation," PhD thesis, Univ. California at Berkeley, Mar. 1990.
[46] 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," *Proc. 13th Ann. Int'l Symp. Computer Architecture (ISCA '86),* Jan. 1986.

Bruce Jacob received the AB degree cum laude in mathematics from Harvard University in 1988 and the MS and PhD degrees in computer science and engineering from the University of Michigan, Ann Arbor, in 1995 and 1997, respectively. At the University of Michigan, he was part of a design team building high-performance, high-clock-rate microprocessors. He has also worked as a software engineer for two successful startup companies: Boston Technology and Priority Call Management. At Boston Technology, he worked as a distributed systems developer and, at Priority Call Management, he was the initial system architect and chief engineer. He is currently on the faculty of the University of Maryland, College Park, where he is an assistant professor of electrical and computer engineering. His present research covers memory-system design and includes DRAM architectures, virtual memory systems, and memory managmeent hardware and software for real-time and embedded systems. He is a recipient of a US National Science Foundation CAREER award for his work on DRAM systems. He is a member of the IEEE, the ACM, and Sigma Xi.

Trevor Mudge received the BSc degree in cybernetics from the University of Reading, England, in 1969 and the MS and PhD degrees in computer science from the University of Illinois, Urbana, in 1973 and 1977, respectively. Since 1977, he has been on the faculty of the University of Michigan, Ann Arbor. He is presently a professor of electrical engineering and computer science and the director of the Advanced Computer Architecture Laboratory—a group of eight faculty and 70 graduate research assistants. He is the author of more than 150 papers on computer architecture, programming languages, VLSI design, and computer vision, and he holds a patent in computer-aided design of VLSI circuits. He has also chaired about 20 theses in these research areas. His research interests include computer architecture, computer-aided design, and compilers. In addition to his position as a faculty member, he is a consultant for several computer companies. He is also associate editor for the *IEEE Transaction on Computers* and *ACM Computing Surveys*. Dr. Mudge is a fellow of the IEEE, a member of the ACM, the IEE, and the British Computer Society.