A Look at Several Memory Management Units, TLB-Refill Mechanisms, and Page Table Organizations

Bruce L. Jacob
Dept. of Electrical & Computer Engineering
University of Maryland, College Park
blj@eng.umd.edu

Trevor N. Mudge
Dept. of Electrical Engineering & Computer Science
University of Michigan, Ann Arbor
tnm@eecs.umich.edu

ABSTRACT

Virtual memory is a staple in modern systems, though there is little agreement on how its functionality is to be implemented on either the hardware or software side of the interface. The myriad of design choices and incompatible hardware mechanisms suggests potential performance problems, especially since increasing numbers of systems (even embedded systems) are using memory management. A comparative study of the implementation choices in virtual memory should therefore aid system-level designers.

This paper compares several virtual memory designs, including combinations of hierarchical and inverted page tables on hardware-managed and software-managed translation lookaside buffers (TLBs). The simulations show that systems are fairly sensitive to TLB size; that interrupts already account for a large portion of memory-management overhead and can become a significant factor as processors execute more concurrent instructions; and that if one includes the cache misses inflicted on applications by the VM system, the total VM overhead is roughly twice what was thought (10-20% rather than 5-10%).

1 INTRODUCTION

Virtual memory (VM) is one of the few interfaces through which the architecture and operating system interact directly. It was developed to automate the movement of program code and data between main memory and secondary storage to give the appearance of a single large store. This greatly simplified the job of the programmer, particularly when program code and data exceeded the size of main memory. The basic idea proved readily adaptable to additional requirements including address space protection, the execution of a process as soon as a single page is in memory, and a user-friendly programming paradigm such as the virtual machine environment in which a process may assume that it owns all available hardware resources. Consequently, virtual memory has become widely used and most modern processors have hardware to support it.

However, there has been little agreement on how virtual memory's functionality is to be implemented on either the hardware or software side of the interface [15, 14]. One need only look at the memory-management units of today's processors to see the variation in hardware designs [14]; the software side of the interface sports numerous page table organizations [12, 18, 30], different hardware abstraction layers [23, 8], and significant variations in performance [22, 21, 4]. While this may pose little problem to applications designers, it is an important issue for systems designers—memory management is playing an increasingly significant role as systems engineers port popular system code to different platforms with varying degrees of memory-management support, as more embedded designers take advantage of low-overhead embedded operating systems that provide virtual memory, and as more designers choose object-oriented systems in which run-time garbage collection is pervasive. An understanding of VM's performance issues is therefore in order.

We present a trace-based simulation study of several different virtual memory designs, including combinations of hierarchical and inverted page tables on hardware-managed and software-managed translation lookaside buffers (TLBs). To our knowledge, this is the first study of its kind, simulating several different software systems (Ultrix, Mach, BSD, Windows NT, PA-RISC) against multiple hardware configurations (software-managed TLB, hardware-managed TLB, no TLB). The primary goal of the study is to determine and understand the fundamental differences between virtual memory implementations: those differences due to one's choice of memory management unit and page table organization. A secondary issue is to distinguish between the performance impact due to VM organization and the impact due to the implementation of that organization. For example, the virtual memory systems of different operating systems can have significantly different performance [22, 4]; is this due to implementation, or is it inherent in the system design?

Our simulation study produces several interesting results:

- The x86 memory-management organization (a hierarchical page table walked from the root to the leaves, with a hardware-managed TLB) outperforms other schemes, even with the handicap that every page table lookup needs two memory references. One reason is that the scheme does not use the precise interrupt mechanism and so avoids the overhead of taking an interrupt every TLB miss. Also, the scheme requires no I-cache and so avoids any memory overhead for fetching instructions.
- Inverted tables can impact the data caches less than hierarchical page tables, even when their page table entries (PTEs) are four times as large as those of hierarchical tables and therefore should impact the data caches four times as much. This is due to the densely-packed nature of PTEs in an inverted table, as compared to the relatively sparsely-packed PTEs of a hierarchical table.
- When one includes the overhead of cache misses inflicted on the application as a result of the VM system displacing user-level code and data, the overhead of the virtual memory system is roughly twice what was previously thought (10-20% rather than 5-10%). These numbers are normally not included in VM studies because, to make a comparison, one must execute the application without any virtual memory system. In addition, when one includes the overhead of handling VM-related interrupts, the total increases to three times what was previously thought: 10-30%.

From this study, we make three observations regarding memory management. First, interrupts already account for a large portion of mem-

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

This work appears in the *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-VIII), San Jose CA, Oct. 3-7, 1998.

ASPLOS-VIII 10/98 CA, USA

ory-management overhead, and they can become a significant factor in overall performance as processors execute larger numbers of concurrent instructions. We conclude that more attention should be paid to improving the handling of precise interrupts (e.g., [10, 31]). If interrupts were infrequent, they would be less of a concern. However, the general-purpose interrupt mechanism is being used increasingly often to support "normal" (or, at least, relatively frequent) processing events such as TLB misses [26, 22], cache misses [6, 13], and system-level functions from copy-on-write to garbage collection to distributed shared virtual memory [2]. Since the precise handling of an interrupt results in the flushing of potentially dozens of instructions from the pipeline and reorder buffer [34], interrupt overhead can become a significant factor as processors execute increasing numbers of concurrent instructions (thereby requiring increasingly large reorder buffers) and as we reduce other sources of performance overhead (e.g., cache misses, layers of system software, serial instruction execution).

Second, finite state machines are the best candidate for walking one's page table, even though they offer limited flexibility—a finite state machine does not impact the instruction cache, incurs no interrupt penalty, and can even operate in parallel with normal instruction execution. A likely future memory-management design would use a programmable finite state machine that walks the page table in a user-defined manner; this would offer the flexibility of alternate page table organizations and yet would incur no interrupt or I-cache overhead.

Third, the performance of software-managed caches is still reasonable [6], which is particularly interesting given the amount of recent activity in alternative mechanisms for determining cacheability of program data [32, 24, 16]. A software-managed cache allows the operating system to determine cacheability of instructions and data on a line-by-line basis; this decision can be made either statically at program compile time or dynamically at program run time. This would enable a designer to define a different cache-fill algorithm on a per-application basis; cache-fill could conceivably be part of the user-level executable itself—similar in concept to the Exokernel notion of *application-level virtual memory* [9].

2 BACKGROUND

To give the illusion of a large translation table held entirely in hardware, early systems provided a hardware state machine to refill the TLB. In the event of a TLB miss, the state machine would walk the page table, locate the mapping, insert it into the TLB, and restart the computation. This is known as a *hardware-managed TLB*. It is an efficient design choice as it disturbs the processor pipeline only slightly. When the state machine handles a TLB miss, the processor effectively freezes. Compared to taking an interrupt, the contents of the pipeline are unaffected, and the reorder buffer need not be flushed. The I-cache is not used and the D-cache is only used if the page table is in cacheable space. At the very worst, the execution of the state machine will replace a few lines in the D-cache. Many designs do not even freeze the pipeline; for instance, the Intel Pentium Pro allows instructions that are independent of the faulting instruction to continue processing while the TLB miss is serviced [33]. The primary disadvantage of the state machine is that the page table organization is effectively etched in stone; the operating system has no flexibility in choosing a design.

Responding to this disadvantage, recent memory-management units have used a *software-managed TLB*, in which there is no hardware state machine to handle TLB misses. In a software-managed TLB miss, the hardware interrupts the operating system and vectors to a software routine that walks the page table and refills the TLB. The page table can thus be defined entirely by the operating system, since hardware never directly manages the table. The flexibility of the software-managed mechanism does come at a performance cost. The TLB miss handler that walks the page table is an operating system primitive usually 10 to 100 instructions long; if the handler code is not in the instruction cache at the time of the TLB miss exception, the time to handle the miss can be much longer than in the hardware-walked scheme. In addition, the use of the precise interrupt mechanism adds to the cost by flushing the pipeline, removing a possibly large number of instructions from the reorder buffer. This can add hundreds of cycles to the overhead of walking the page table. Nonetheless, the flexibility of the software-managed scheme can outweigh the potentially higher per-miss cost of the design [22].

Typical studies put TLB handling at 5-10% of a normal system's run time [7, 22, 25]. However, there can be worst-case scenarios: studies have shown that significant overhead can be spent servicing TLB misses [3, 5, 22, 29]. In particular, Anderson [1] shows TLB miss handlers to be among the most commonly executed primitives, Huck and Hays [12] show that TLB miss handling can account for 40% of total run time, and Rosenblum [25] shows that TLB miss handling can account for 80% of the kernel's computation time.

Note that a TLB is not necessary if one uses virtual caches. For example, *softvm* [13] is a scheme in which the processor uses a virtual cache hierarchy and receives an interrupt on every level-2 cache miss. On a miss, the operating system performs the page table lookup and cache fill in software. This is similar to the *software-managed caches* of the VMP multiprocessor [6], which even performed cache-consistency management in software, invalidating stale copies of data in the caches of other processors. Virtual caches do have drawbacks, including the need to maintain ASIDs and protection information with the cache tags, and the synonym problem of virtual-address aliasing [35]. Usually, the OS must be aware of a virtual cache, whereas a physical cache is transparent to software. While there are solutions for these problems, virtual caches are often avoided because of them.

Support for precise interrupts traditionally requires that at the time the interrupt is handled, the state of the machine reflect what the state would have been had the machine executed instructions one at a time. This is the *precise state* and reflects a sequential model of execution. Several hardware mechanisms support this in pipelined and out-of-order execution engines (e.g. the reorder buffer, history buffer, future file [27], and register update unit [28]), and many processors achieve a precise state by waiting until the exception-causing instruction is at the head of the reorder buffer and then flushing the contents of the buffer [20, 34]. Henry addresses the issue by tagging each individual instruction with its execution permissions so that user-level and supervisor-level instructions can co-exist in the pipeline simultaneously [10]. However, one still must ensure that before the exception is handled, there are no partially-completed instructions that might yet cause exceptions out-of-order (which is why many implementations wait until the instruction in question is at the head of the reorder buffer). One possible solution is to speculatively handle the exception immediately, and back out only if there are problems.

3 EXPERIMENTAL METHODOLOGY

We are interested in the performance issues associated with different page table organizations, page table lookup schemes, and TLB architectures. We are less interested in knowing what the actual performance is; we are more interested in understanding the performance behavior. We performed trace-driven simulations of five memory-management organizations on three TLB configurations, using the SPEC '95 integer suite for our benchmarks. To avoid obscuring performance differences, we simulated split, direct-mapped caches at both the L1 and L2 levels (set associative or unified caches, while giving better performance, would add too many variables for us to interpret behavior). For the same reason, we simulated blocking caches at both levels and looked at individual benchmark results rather than averages. The following sections discuss the simulated hardware/software systems and the statistics taken by our simulator.

Table 1: Simulation details

Characteristic	Range of values simulated
Benchmarks	SPEC '95 integer suite
Cache organizations	Caches are split, direct-mapped, virtually-addressed All caches are blocking, write-allocate, write-through
L1 cache size	1, 2, 4, 8, 16, 32, 64, 128KB (per side)
L2 cache size	512KB, 1MB, 2MB (per side)
Cache linesizes	16, 32, 64, 128 bytes
TLB organizations	TLBs are fully associative with random replacement (similar to MIPS). Some simulations (those that are most MIPS-like: MACH and ULTRIX) reserve 16 slots for "protected" entries containing root-level PTEs; other simulations (INTEL, PA-RISC) do not.
TLB size	128-entry I-TLB/128-entry D-TLB
Page size	4 KB
Cost of interrupt	10, 50, 200 cycles
Architecture/ operating system combinations	**ULTRIX**: Ultrix (BSD-like) on MIPS **MACH**: Mach on MIPS **INTEL**: BSD/Windows NT on Intel x86 **PA-RISC**: HP-UX hashed page table on PA-RISC **NOTLB**: Software-managed caches and no TLB **BASE**: Baseline cache performance without VM

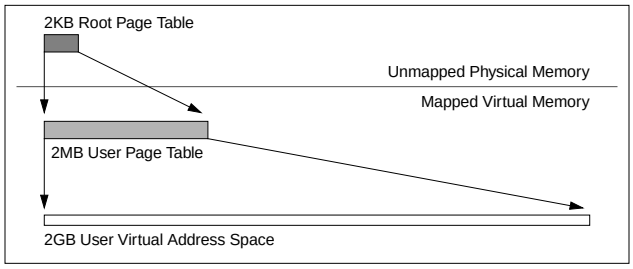

Figure 1: The Ultrix/MIPS page table organization. The Ultrix page table on MIPS is a simple two-tiered table. The user address space is the bottom 2GB of the hardware's address space; the top 2GB belongs to the kernel. A 2KB table wired down in physical memory maps each user page table.

3.1 Scope of study

The range of variables simulated is listed in Table 1; this paper evaluates a space equal to their effective cross-product. We simulate operating system activity related to searching the page tables and managing the TLB (if present). In particular, we measure the OS's effect on the cache hierarchy, including I-cache misses when executing handlers and D-cache misses when loading PTEs. The following pseudocode illustrates the fundamental simulator algorithm:

```
while (i = get_next_instruction()) {
     if (itlb_miss( i->pc )) {
         walk_page_table( i->pc );
         insert_itlb( i->pc );
     }
     icache_lookup( i->pc );
     if (LOAD_OR_STORE( i )) {
         if (dtlb_miss( i->daddr )) {
             walk_page_table( i->daddr );
             insert_dtlb( i->daddr );
         }
         dcache_lookup( i->daddr );
     }
}
```

Depending on the page table organization being simulated, the function `walk_page_table()` may or may not access the I-cache and instruction & data TLBs; it will always access the D-cache.

The systems simulated include the DEC Ultrix virtual memory system as implemented on a software-managed TLB such as the MIPS, Mach's virtual memory system as implemented on MIPS, the BSD virtual memory system as implemented on the Intel IA-32 architecture (which is similar to the Windows NT virtual memory system on IA-32, except that we do not implement shared memory and hence do not use Windows NT's *prototype PTE* mechanism [8]), the PA-RISC virtual memory system as described by Huck and Hays [12], and a system with software-managed caches and no TLB, as in VMP or *softvm* [6, 13]. The following sections describe each of the MMU and page table organizations simulated.

ULTRIX. The Ultrix page table as implemented on MIPS is a two-tiered table walked bottom-up [22], illustrated in Figure 1. The 2GB user address space is mapped by a 2MB linear table in virtual space, which is in turn mapped by a 2KB array of PTEs. It requires at most two memory references to find the appropriate mapping information.

The caches in the simulation are all virtual. The TLB (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) is used to provide protection information; if the TLB misses on a reference, the page table is walked before the cache lookup can proceed. The TLB miss handler is comprised of two code segments: one to handle user-level misses, one to handle kernel-level misses. The first code segment is called when an application load, store, or instruction-fetch causes a TLB miss; the second handles the case when a PTE reference in the first handler causes a TLB miss. The handlers are located in unmapped space, so executing them cannot cause I-TLB misses. We choose handler lengths based on real-world code, e.g., the MIPS TLB-miss handler (sans several NOPs):

```
mfc0 k0,tlbcxt    # move context register to k0
mfc0 k1,epc       # move PC of bad instr. to k1
lw k0,0(k0)       # load mapping PTE
mtc0 k0,entry_lo  # move PTE into EntryLo
tlbwr             # write PTE into TLB
j k1              # jump to bad PC (to retry)
rfe               # RESTORE FROM EXCEPTION
```

The user-level handler is ten instructions long; the kernel-level handler is twenty. The start of the handler code is page-aligned. The following is pseudocode for the ULTRIX *walk_page_table* function:

```
tlbmiss_handler( UPT_HANDLER_BASE, 10 );
if (dtlb_miss( UPT_BASE + uptidx( addr ) )) {
     tlbmiss_handler( RPT_HANDLER_BASE, 20 );
     dcache_lookup( RPT_BASE + rptidx( addr ) );
}
dcache_lookup( UPT_BASE + uptidx( addr ) );
```

The variable *addr* is passed in as an argument. The *tlbmiss_handler* function probes the I-caches for a number of instructions beginning at a base address (simulates execution of handler). The *uptidx* and *rptidx* functions calculate indices into the user and root page tables, respectively, which amounts to extracting and right-shifting a bitfield. The code simulates the effect that walking the page table has on the data TLB, I-cache, and D-cache. The other VM simulations are analogous.

MACH. The Mach page table as implemented on the MIPS processor is a three-tiered table walked bottom-up [22, 3], illustrated in Figure 2. The 2MB user page tables are located in kernel space, the entire 4GB kernel space is mapped by a 4MB kernel structure, which is in turn mapped by a 4KB kernel structure. It requires at most three memory references to find the appropriate mapping information.

The Mach TLB-miss handler on actual MIPS hardware uses two main interrupt paths. There is a dedicated interrupt vector for user-

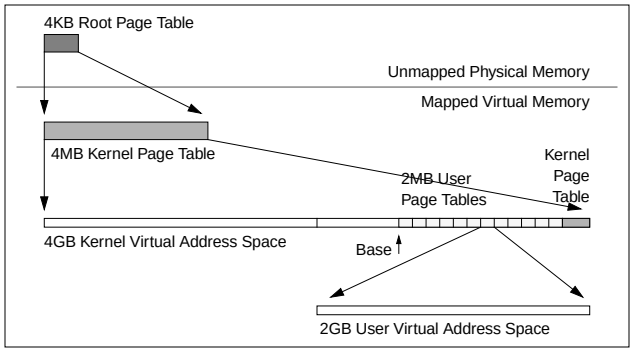

Figure 2: The Mach/MIPS page table organization. Mach as implemented on MIPS has a three-tiered page table. A user-level address space is mapped by a 2MB table in kernel space at an offset aligned on a 2MB boundary and related to the process ID of the user-level application: the virtual base address of the table is essentially *Base + (processID * 2MB)*. The top 4MB of the kernel's virtual address space is a page table that maps the 4GB kernel space. This kernel table is in turn mapped by a root table in physical memory.

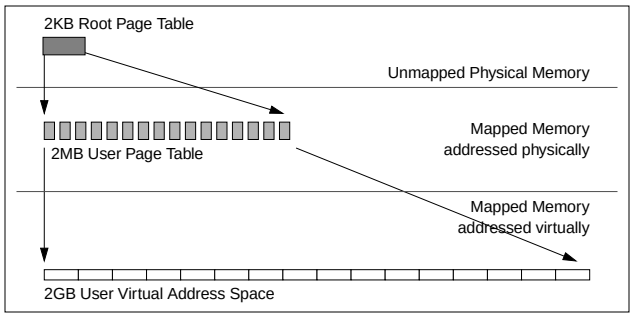

Figure 3: The BSD/Intel page table organization. The Intel page table is similar to the MIPS and NOTLB page tables; it is a two-tiered hierarchical table. However, unlike the other two, it is walked in a top-down fashion. Therefore, the user page table is a set of page-sized tables (4KB PTE pages) that are not necessarily contiguous in either physical space or virtual space (they do not need to be contiguous in virtual space because the table is never treated as a unit; it is never indexed by the VPN). These 4KB PTE pages map 4MB segments in the user's virtual address space. The 4MB segments that make up the user's address space are contiguous in virtual space.

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 code that adds an enormous cost to interrupts that cannot be handled by the dedicated vector. Measurements taken by Bala show that the non-user-level TLB miss paths can be several hundred cycles long [3]. However, to put our simulated VM systems on equal footing, we include an additional interrupt vector for kernel-level misses. Doing so should reduce the cost of many TLB misses by an order of magnitude [22]. Our simulated user-level TLB-miss handler is 10 instructions long, our kernel-level miss handler is 20 instructions long, and to differentiate the MACH simulation, we make the cost of accessing the root-level table extremely high. Root-level misses take a long path of 500 instructions and perform a number of additional loads to simulate the effect of the administrative code. The handlers are located in unmapped space (executing them cannot cause I-TLB misses). As with the handler code of the other simulated systems, the beginning of each section of handler code is aligned on a page boundary.

INTEL. An Intel x86-based page table is a two-tiered hierarchical table, but unlike the MIPS-style page table, an Intel-style page table is walked from the root level of the table down. Therefore on every TLB miss the hardware makes exactly two memory references to find the mapping information; one indexes the root table, the other indexes the user-level table. The organization is illustrated in Figure 3. The advantage of the Intel design is that the system does not take an interrupt on a TLB miss (it uses a hardware-managed TLB), and the contents of the instruction cache are unaffected. However, the contents of the data cache *are* affected; we assume in these simulations that the page tables are cacheable. The simulated TLB-miss handler takes seven cycles to execute, plus any stalls due to references to the page table that miss the data cache; executing it cannot affect the I-caches or cause any I-TLB misses. The table is walked in a top-down fashion and uses physical addresses; therefore referencing entries in the table cannot cause D-TLB misses. The number of cycles (7) is chosen to represent the minimum amount of sequential work to be done:

```
cycle 1: shift+mask faulting virtual address
cycle 2: add to base address stored in register
cycle 3: load PTE at resulting physical address
cycle 4: shift+mask faulting virtual address
cycle 5: add to base address just loaded
cycle 6: load PTE at resulting physical address
cycle 7: insert mapping information into TLB,
         return to instruction stream
```

The cache organization is identical to the other simulations. The root-level PTEs are not cached in the TLB, thus the TLBs are not partitioned as in the ULTRIX and MACH simulations. All 128 entries in each TLB are available for user-level PTEs in the INTEL simulation.

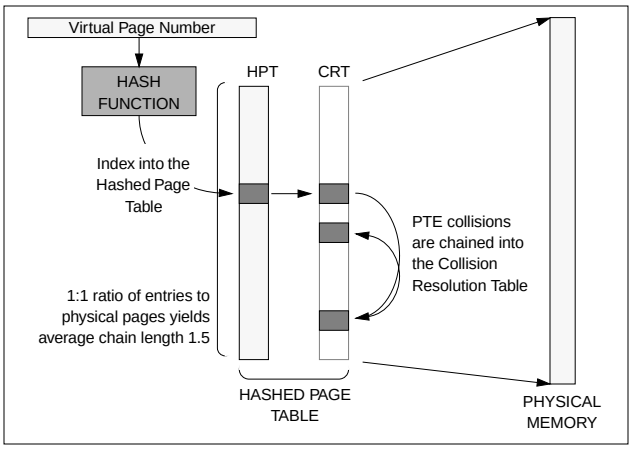

Figure 4: The PA-RISC page table organization. The PA-RISC hashed page table is similar in spirit to the classical inverted page table, but it dispenses with the hash anchor table, thereby eliminating one memory reference from the lookup algorithm. Since there is not necessarily a 1:1 correspondence between entries in the table and page frames in the system, the PFN must be stored in the page table entry, thereby increasing its size. While the collision-resolution table is optional, we include it in our simulation.

PA-RISC. PA-RISC uses a variant of the inverted page table that is more efficient in the number of memory references required to locate mapping information, but which requires that the page table entries be larger [12]: the PTEs are 16 bytes long, as compared to the 4-byte PTEs in the other VM simulations. The page table organization is illustrated in Figure 4. On a TLB miss, the operating system hashes the faulting virtual address to find the head of a collision-chain. Note that the walking of the table could be done in hardware as easily as in software. A page table lookup can require many memory references to find the mapping information, as there is no guarantee on the number of virtual addresses that produce the same hash value.

Simulating the PA-RISC is more difficult than simulating the other architectures. Since the size of the page table is dependent on the size of physical memory, we must make some choices about the organization of physical memory. We define our simulated physical memory to be 8MB, which is small for the average workstation but

Table 2: Components of MCPI

Tag	Cost per
L1i-miss	20 cycles
L1d-miss	20 cycles
L2i-miss	500 cycles
L2d-miss	500 cycles

Table 3: Components of VMCPI

Tag	Cost per	Description
uhandler	variable	A TLB miss (or an L2 cache miss in the case of a NOTLB 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 NOTLB 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 the miss handler, code misses the L1 instruction cache; reference goes to L2 instruction cache
handler-MEM	500 cycles	During execution of the miss handler, code misses the L2 instruction cache; reference goes to main memory

Table 4: 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
INTEL	7 cycles, 2 PTE loads	n.a.	n.a.
PA-RISC	20 instrs, variable # PTE loads	n.a.	n.a.
NOTLB	10 instrs, 1 PTE load	n.a.	20 instrs, 1 PTE load

larger than the physical memory requirements of any one benchmark, and so this should behave similarly to a large physical memory. An 8MB physical memory has 2,048 4KB pages; we choose a 2:1 ratio to get 4,096 entries in the page table, which should result in an average collision-chain length of 1.25 entries [17]. GCC, for example, produced an average collision-chain length of just over 1.3. We place no restriction on the size of the collision resolution table.

We do not simulate the operating system’s page placement policy, since the cache hierarchy is entirely virtually-addressed and the placement of a PTE within the hashed page table is dependent on the virtual page number, not the page frame number. We use the same hashing function as described by Huck & Hays: “a single XOR of the upper virtual address bits and the lower virtual page number bits.” We also use Huck & Hays’ 16-byte PTE. The hierarchical page tables use 4-byte PTEs (a PTE for a hierarchical page table scales with the size of the physical address), so a PTE load in the PA-RISC simulation impacts the data cache four times as much as in other simulations.

There is one TLB-miss handler, and the handler cannot cause a data TLB miss: as with the Intel page table, the handler uses physical but cacheable addresses to access the page table. The handler is twenty instructions long, located in unmapped space, so executing it cannot cause misses in the I-TLB. No distinction is made between user-level PTEs and kernel-level PTEs, therefore the simulated TLBs are like those in the INTEL simulations: they are not partitioned—all 128 entries in each of the TLBs are available for user-level PTEs.

NOTLB. NOTLB uses a two-tiered “disjunct” page table similar to a Ultrix/MIPS page table; it is described in detail in [13] and is illustrated in Figure 5. It is based on a segmented address space; the segments that make up the 2GB space are disjunct segments in a flat global space and the page groups that make up the user page table are also disjunct regions in the flat space. As with the Ultrix table, it requires at most two memory references to find mapping information.

The cache-miss handler is comprised of two code segments located in unmapped space (executing them cannot cause cache-miss exceptions). The first code segment is called when an application

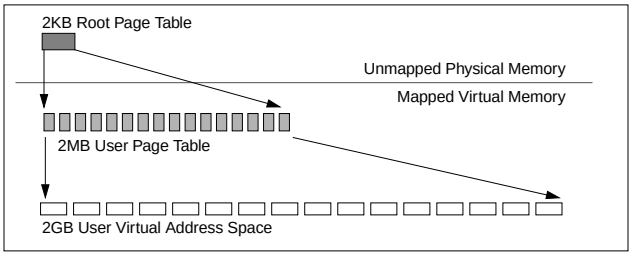

Figure 5: The disjunct page table organization. The NOTLB page table is comprised of page groups in a global segmented address space that are not necessarily contiguous—just as the segments that make up the application’s address space are not necessarily contiguous. The page table is traversed bottom-up like an Ultrix/MIPS page table.

load, store, or instruction-fetch misses the L2 virtual cache; the second handles the case when a PTE reference in the first handler misses the L2 cache. The first is ten instructions long, the second is twenty. Since the ULTRIX and NOTLB page tables are similar and the cost of walking the tables is identical, the differences between the measurements should be entirely due to the presence/absence of a TLB.

3.2 Statistics gathered

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 (partitioning of the TLB or adding a hardware state machine to walk the page table will not significantly impact cycle time). We present the measurements classed as memory-system overhead (MCPI) and virtual-memory overhead (VMCPI). VMCPI is the number of additional 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. MCPI represents the basic cost of the memory

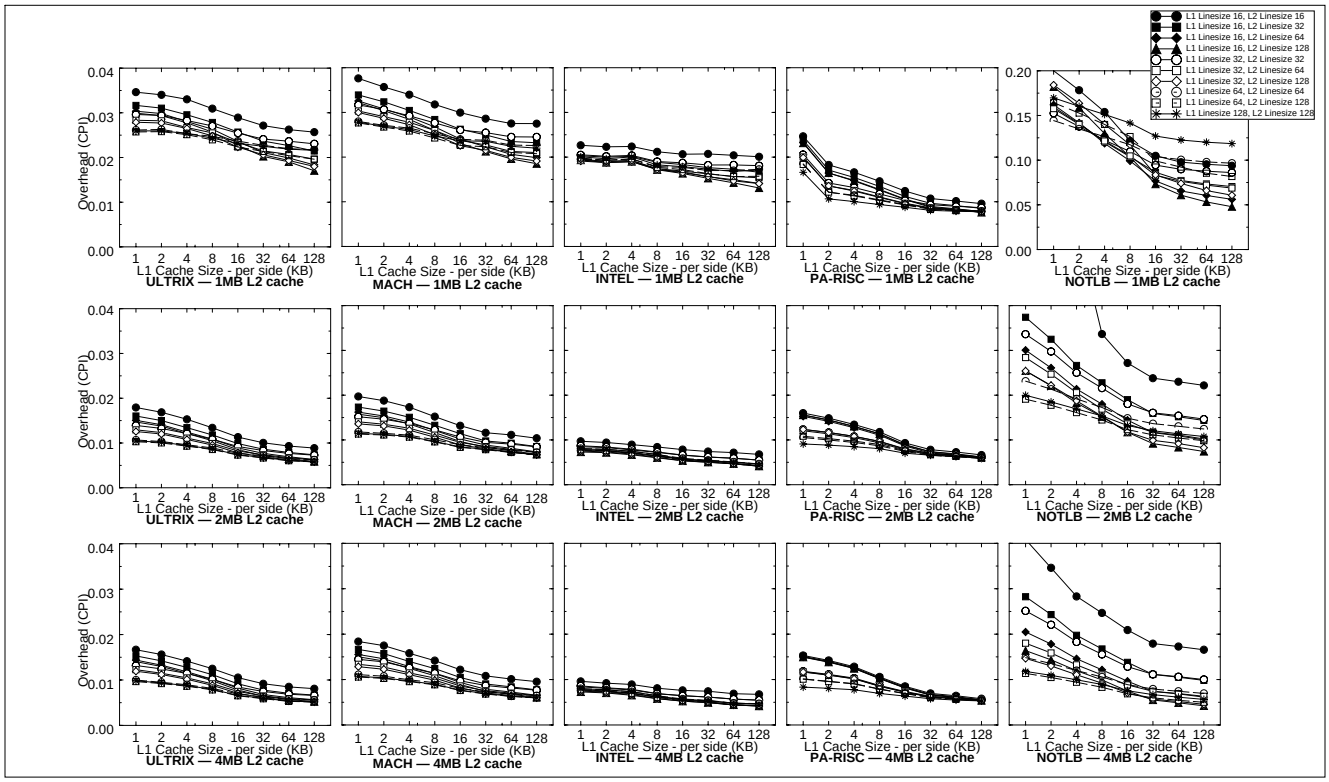

Figure 6: VMCPI vs. L1 and L2 cache size and linesize — GCC. These are the VMCPI totals for each of the VM simulations. This overhead represents only the cost of walking the page table and refilling the TLB (or, in the case of the NOTLB simulation, filling a cache block). Each data point represents one run of the simulator; each curve represents a different L1/L2 linesize configuration. Note that the scale differs for the NOTLB graph with 1MB L2 cache size.

system and includes only user-level references—but it does include those misses incurred when application instructions and/or data are displaced by the miss handlers. MCPI and VMCPI are further subdivided into the categories described in Tables 2 and 3. Note that not all of the categories apply to all simulations; for instance, the NOTLB and ULTRIX simulations have no kernel-level miss handlers (*khandler*, *kpte-L2*, and *kpte-MEM* events will not happen), and the INTEL simulation cannot cause instruction cache misses during execution of the handler (*handler-L2* and *handler-MEM* events will not happen).

Each VM simulation handles cache or TLB misses differently, since each uses a different MMU and page table organization. When simulating TLB-miss handlers, the contents of the I-caches are overwritten with the handler code (if the TLB is software-managed), and PTE loads overwrite the D-caches. The costs of the events that can occur in each of the simulations are summarized in Table 4. To put the organizations on 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. 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. All systems should see the same number of page initializations, corresponding to the first time each page is touched. Our measurements do not include this cost, as it would be the same for every simulation. The 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. The only part of the OS simulated is the TLB-refill mechanism.

Due to space constraints, in this paper we focus only on the benchmarks that have the worst virtual memory performance: **gcc** and **vortex**, and one that provides interesting counterexamples: **ijpeg**. We chose data sets so that each program would run to completion in less than 200 million instructions; this makes the total running time feasible since the number of simulations is extremely large.

4 EXPERIMENTAL RESULTS

We first look at VMCPI overhead alone, then show its relation to the cost of interrupts, the application's cache performance, and the baseline overhead of executing the application without virtual memory.

4.1 VMCPI as a function of cache organization

We begin by presenting the VMCPI overheads as a function of L1 and L2 cache sizes and linesizes. Figure 6 gives the results for GCC, and Figure 7 gives the results for VORTEX. Several things are clear:

- The overheads are in the right ballpark to represent a 5-10% overhead for a 1 CPI machine, even without considering address space and page table initialization, paging, I/O, etc.
- 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.
- The software-managed cache (NOTLB) tends to do about as well as the other schemes, once the L2 cache is large enough (2MB+), and once a suitable linesize is chosen (L2 linesize ≥ 64 bytes).
- The VM system without a TLB (NOTLB) is much 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.

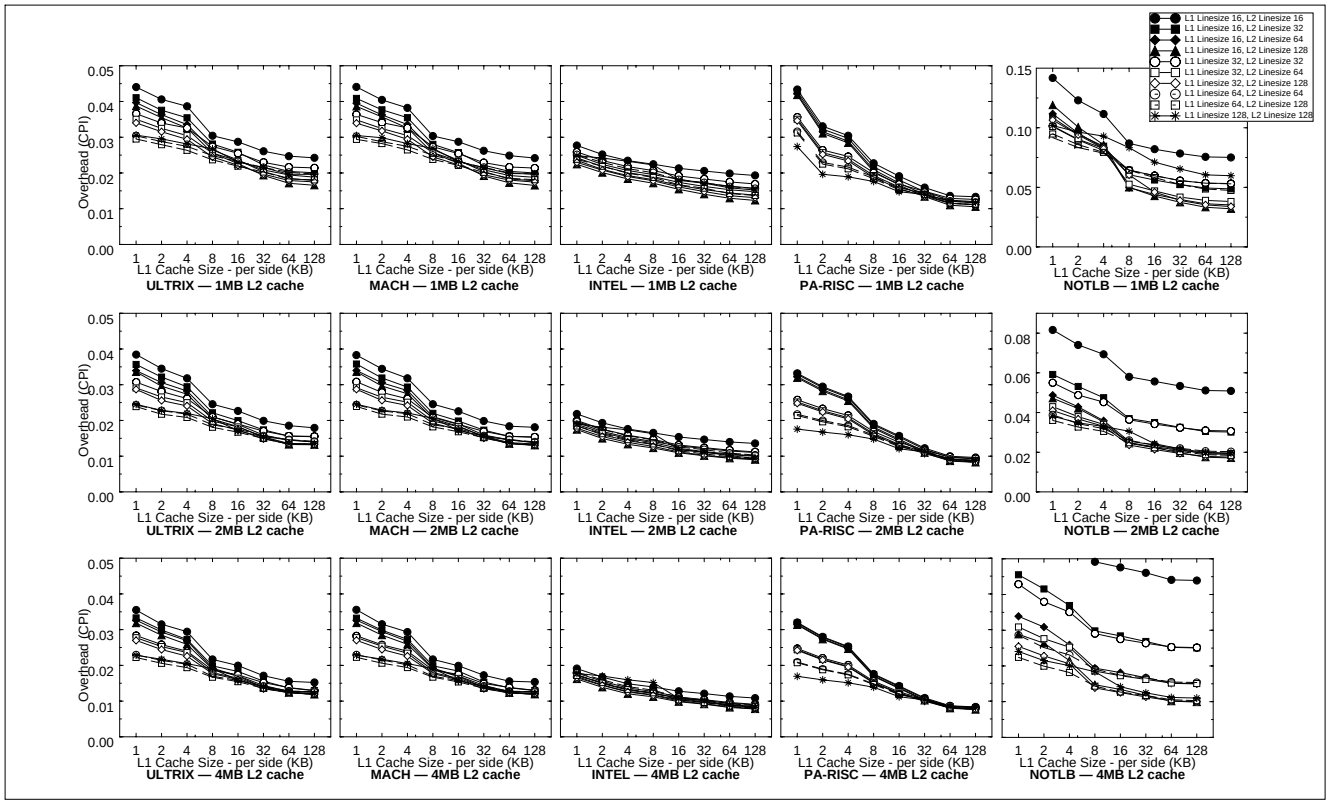

Figure 7: VMCPI vs. L1 and L2 cache size and linesize — VORTEX. These are the VMCPI totals for each of the VM simulations. This overhead represents only the cost of walking the page table and refilling the TLB (or, in the case of the NOTLB simulation, filling a cache block). Each data point represents one run of the simulator; each curve represents a different L1/L2 linesize configuration. Note that the scale differs for the NOTLB graphs with 1MB and 2MB L2 cache sizes.

- For larger L1 cache sizes, the PA-RISC organization is relatively immune to the choice of linesize, whereas the other schemes can still be affected by a factor of two with the choice of linesize.
- For smaller L1 cache sizes, the PA-RISC organization continues to benefit from increased linesize, after the other simulations show the familiar signature of diminishing returns. For all other simulations, performance remains constant or worsens when moving from 64-byte linesizes to 128-byte linesizes (holding L1 cache size constant).
- The curves for the TLB-based schemes are roughly grouped by L1 linesize (most evident in the PA-RISC graphs): for small L1 caches, linesize is more important than cache size. In contrast, the NOTLB curves are roughly grouped by L2 linesize: L2 linesize is more important than L1 cache size.

These simple graphs lend a good first insight into the VM systems. Since the differences in performance between MACH and ULTRIX are slight, we conclude that even fairly complicated page tables can have low overheads provided that the common-case portions of the structure are efficient. The PA-RISC simulation seems to contend with the cache system less than the other schemes, especially for larger caches—this could be due to the page table organization, which is what sets the PA-RISC simulation apart from the others. Later results show this to be the case.

4.2 VMCPI break-downs

To better understand the behavior, we present break-downs for specific linesize organizations. For space and sensory-overload considerations, we limit this to one organization: 64/128-byte L1/L2 linesizes. For most VM simulations, this choice is consistently at or near the top in performance. The graphs of the worst-performing organizations have similar shapes. Figure 8 shows the VMCPI break-downs for GCC; Figure 9 shows results for VORTEX. We note the following:

- The *uhandlers* component is a conservative measurement. For example, hardware-walked page tables could overlap this cost with normal instruction execution (as in the Pentium Pro).
- The PA-RISC simulation fits well into small L2 cache sizes; small *upte-MEM* values indicate that PTEs for the inverted table are found at the L2 level when they tend to miss at the L2 level for other simulations. For GCC, PA-RISC has a harder time than the other simulations keeping its PTEs in the L1 cache (the *upte-L2* values do not taper down for larger L1 cache sizes), but for VORTEX, the opposite is true—the *upte-L2* values taper down more quickly than for the other simulations.
- For the TLB-based schemes, the *uhandlers* cost is constant over all cache organizations, whereas it decreases with increasing L2 cache sizes for the NOTLB simulations. This is because the frequency of executing the *uhandlers* is dependent entirely on the TLB miss rates for the TLB-based schemes, whereas it is dependent on the L2 cache miss rates for NOTLB. For all schemes, the *uhandlers* cost (the base overhead of the handler in terms of the number of instructions cycles required to execute it, given perfect caches) becomes dominant as cache sizes increase.
- The INTEL measurements show a noticeable overhead of having to go to the L2 cache and physical memory for the root-level PTEs (*rpte-L2* and *rpte-MEM*). This occurs so infrequently in the other simulations that it tends not to register in their bar charts. Unlike the other organizations, the INTEL page table is walked in a top-down manner, so the root level is accessed on every TLB

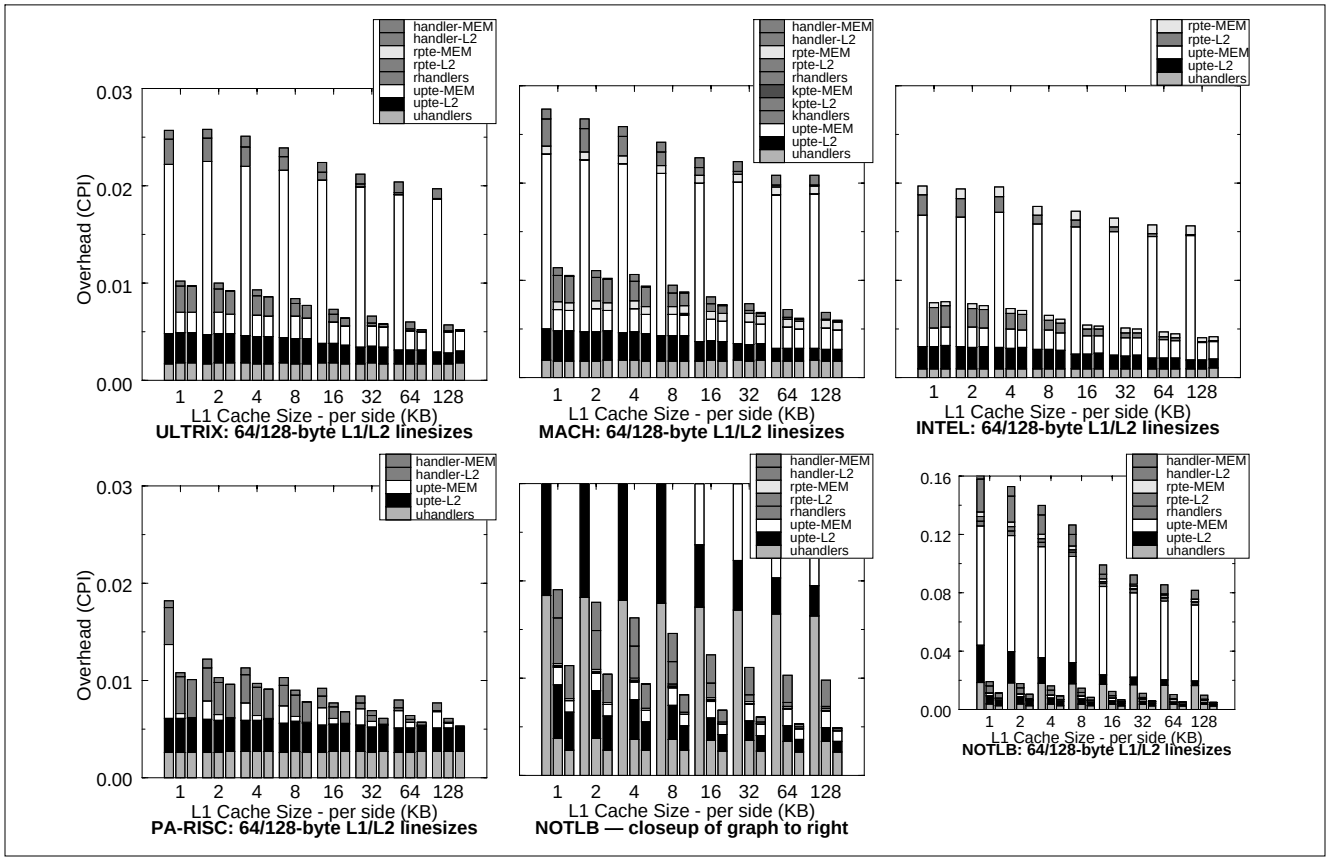

Figure 8: VMCPI break-downs — GCC. These are the VMCPI break-downs for the best-performing choices of linesize: 64/128 bytes in the L1/L2 caches. The x-axis represents different L1 cache sizes. For each point, we show three stacked bar charts, corresponding to 1, 2, and 4MB L2 cache sizes.

miss. The graphs show that for small L1 caches, one will miss the L1 cache equally often when referencing the root-level and user-level PTEs (*rpte-L2* and *upte-L2* are roughly the same size): if one PTE reference is likely to miss the cache, the other is likely to miss the cache as well. However, as the Level-1 cache increases, the *rpte-L2* overhead grows small very quickly; this is expected, since a single root-level PTE maps many user-level PTEs.

- The primary difference between MACH and ULTRIX is in *rpte-MEM*, which, along with *rpte-L2* and *rhandlers*, is where we account for the simulated "administrative" memory activity in the MACH simulation. This supports the conclusion that excessive VM overheads seen in other studies (e.g. [22]) are largely due to implementation and are not inherent in the page table design.

We now can reason about the behavior of PA-RISC. The hierarchical page table in ULTRIX, MACH, INTEL and NOTLB simulations is likely to spread itself across a large portion of the cache, if not across the entire cache. In our simulations, the virtual address spaces are 2GB, which means the page table spans 2MB—roughly the size of the L2 cache. Since space in the table is allocated an entire page at a time, there can be many empty PTEs adjacent in the table; these are likely to waste space in caches with long lines. In contrast, the PA-RISC inverted page table clusters the PTEs together more densely. Space in the table is allocated one PTE at a time; therefore the table is likely to make better use of cache space and will cover a smaller portion of the cache. This is not surprising—inverted tables were invented to save space. This effect is most likely to be seen in applications that tend to exhibit low degrees of spatial locality, which is certainly true for VORTEX: it is a database application with data accesses that have poor spatial locality. The inverted page table is also less likely to hit hotspots in the caches than the hierarchical page table, though this is easily solved with set associativity.

The GCC measurements show that the inverted table can do worse in the L1 caches than the hierarchical tables, evidenced by constant values for *upte-L2* for all L1 cache sizes. This is because each PTE in the inverted table is four times the size of the PTEs for the other schemes—for L1 caches, this is a significant effect. The VORTEX results show different behavior: the inverted table fits better in both L1 and L2 caches than the hierarchical table. The frequency of missing the L1 cache on PTE loads decreases by roughly 80% as the L1 cache size increases (*upte-L2*), as opposed to the decreases of 50% for the other hardware-oriented schemes, and the frequency of missing the L2 cache on PTE loads is much lower than any other VM-simulation (*upte-MEM*). However, this discrepancy says less about the inverted page table than it does about the two benchmarks.

In summary, the schemes with the lowest overhead are Intel's hardware-managed TLB, which requires little overhead to execute the handler, and the inverted page table of PA-RISC, which fits into the data caches better than hierarchical tables. The best solution would be to merge these two and use a hardware-managed TLB with an inverted page table. Note that this is exactly what has been done in the PowerPC and PA-7200 architectures [19, 12]. We also see that software-managed caches have widely varying performance that is highly dependent on cache organization, though with the right caches the scheme can perform as well as TLB-based designs.

We can use these results to interpolate for the costs of other VM organizations, such as an inverted page table with a hardware-managed TLB, a MIPS-style page table with a hardware-managed TLB, or a system with no TLB but a hardware-walked page table (as in SPUR [11, 36]). A hardware-managed TLB with an inverted page

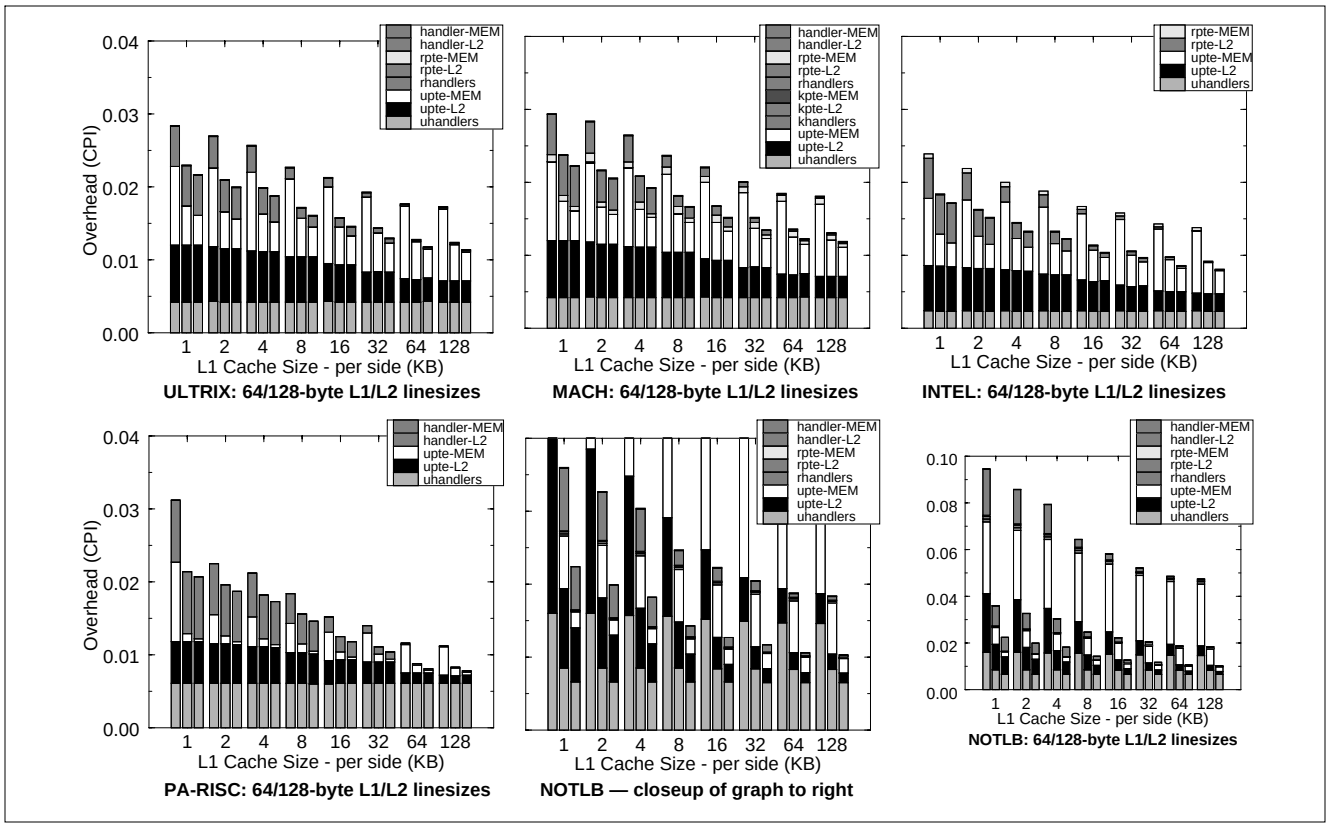

Figure 9: VMCPI break-downs — VORTEX. These are the VMCPI break-downs for the best-performing choices of linesize: 64/128 bytes in the L1/L2 caches. The x-axis represents different L1 cache sizes. For each point, we show three stacked bar charts, corresponding to 1, 2, and 4MB L2 cache sizes.

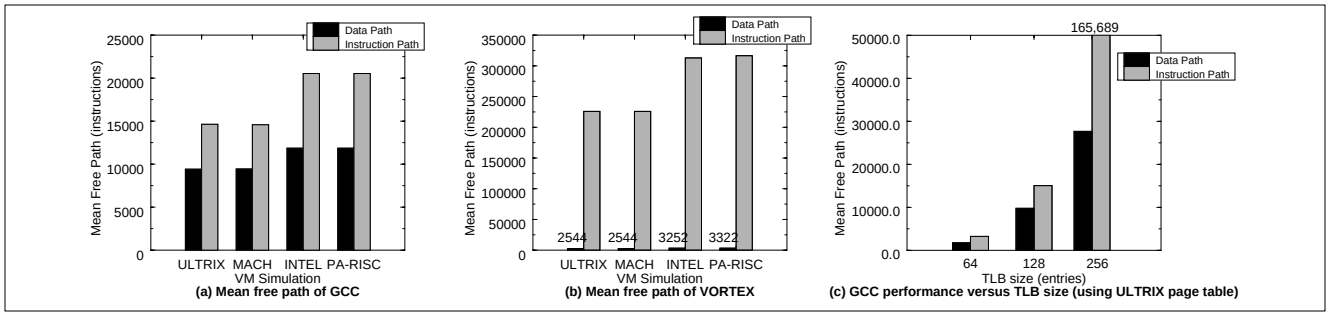

Figure 10: Mean free path, TLB performance. Mean free path is the average number of user-level instructions processed between TLB misses.

table would have similar overhead to the PA-RISC graphs, if one were to delete the top two bars (*handler-L2* and *handler-MEM*—those caused by invocation of the software miss handler) and scale the bottom bar to a fraction of its present size (*uhandlers*—the overhead of executing instructions, which could be reduced if done in hardware).

4.3 Mean free path between VM interrupts

We note that there is some variation in the bottommost grey bars (*uhandlers* overheads) of the TLB-based systems. They should be in the proportions 10, 10, 7, 20 for ULTRIX, MACH, INTEL, PA-RISC, respectively: these are the ratios of the costs of their handlers, and the handlers should execute with the same frequency, since this frequency is dependent only on the interaction of the address stream and the TLB. However, the *uhandlers* overheads are actually in the proportions 10, 10, 5, 15. This prompted us to look at the TLB performance numbers for each simulation. We give performance as the mean free path, which is similar to the metric used by Chen, et al [5] and corresponds to TLB hit rate. The numbers are presented in Figure 10, and we note that they are consistent with Chen's.

Figures 10(a) and 10(b) illustrate the differences; the TLB performance is obviously not identical for the four TLB-based simulations, though common sense says it should be. Moreover, there seems to be little that can account for a 30% difference in TLB hit rate. However, Figure 10(c) shows the differences when the size of the TLB is varied by a factor of two. Whereas changing the cache sizes by several orders of magnitude tends to result in factor-of-two differences in VM performance, factor-of-two changes in TLB size can result in order-of-magnitude differences in VM performance.

This explains the discrepancy between the four VM simulations. The two with lower mean free paths (MACH and ULTRIX) simulate partitioned TLBs. They reserve 16 of the 128 TLB entries for root- and kernel-level PTEs; 112 TLB entries are available for user-level PTEs. The INTEL and PA-RISC simulations, on the other hand, use all 128 TLB entries for user-level PTEs. Therefore, a 15% change in the TLB size results in a 30% change in TLB hit rate.

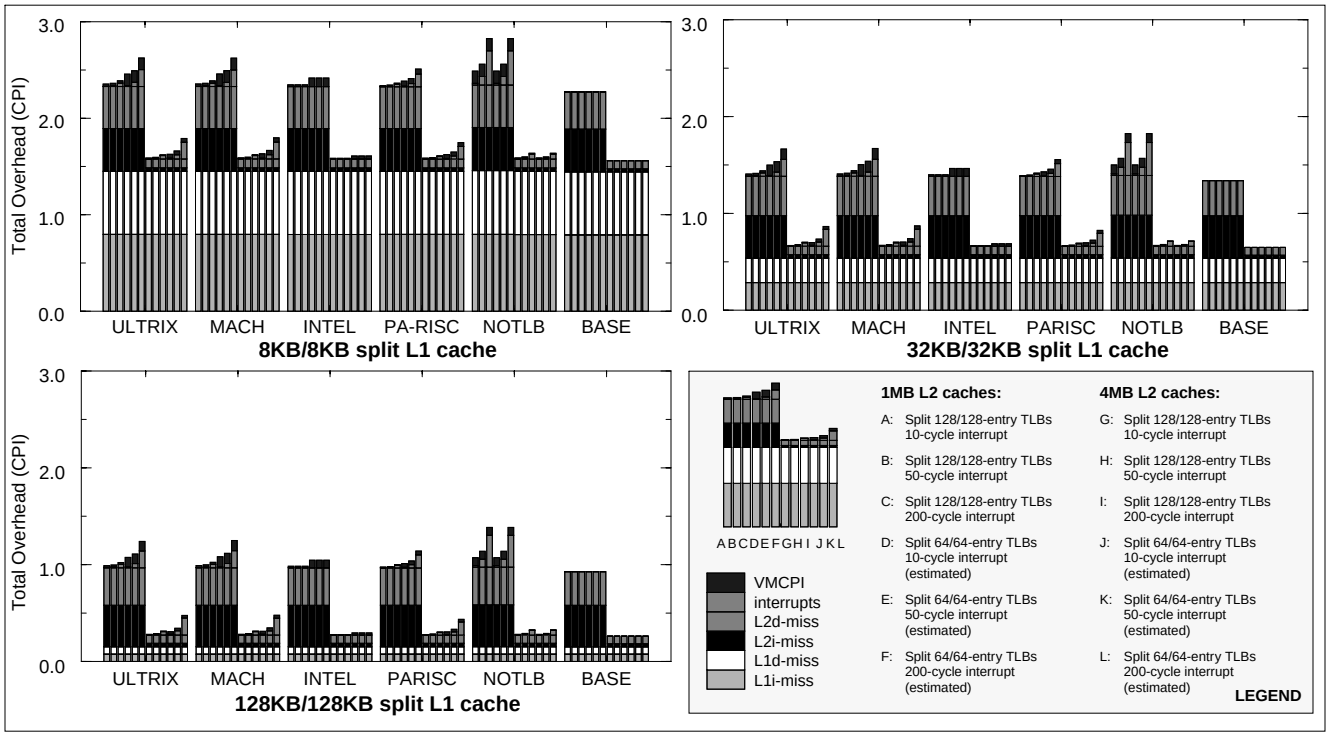

Figure 11: Total overhead — GCC. *Total overhead* includes application cache misses, interrupt costs, and virtual memory overhead; it is additive to base instruction-execution costs. All graphs represent 64/128-byte L1/L2 linesizes. For each VM simulation we show the performance of twelve configurations, covering 1MB L2 caches, 4MB L2 caches, 128/128-entry TLBs, 64/64-entry TLBs, and three different interrupt costs. Note that 64-entry TLB performance is estimated.

Note this does *not* imply that partitioning a TLB is a bad idea. Partitioning is important when there are two classes of PTEs in one's page table design: one that has a low overhead for replacement, and another that has a high overhead for replacement. For performance, it is necessary to keep the high-cost PTEs in the TLB; partitioning is an effective solution [22]. However, this study *does* show that a page table organization requiring TLB-partitioning can perform worse than a page table organization requiring none, because the scheme requiring no partitioning makes better use of the available TLB entries.

4.4 Total overhead: MCPI + VMCPI + interrupts

In this section, we show the relation of virtual memory overhead to overall performance, including VMCPI, MCPI, and the cost of taking interrupts. We will refer to the sum of these three overheads as *total overhead*, even though it does not include I/O, paging, or even instruction execution costs. This is the overhead that would be added to the base cost of instruction execution.

We present three interrupt models: 10-, 50-, and 200-cycle. The first corresponds to a simple in-order pipeline where all state is held in pipeline registers and the cost of taking an interrupt is roughly the cost of draining the pipe. This assumes that general-purpose registers need not be saved, as is often the case when handling memory-management interrupts in the MIPS processor. The second model corresponds to a contemporary out-of-order processor with a few dozen reorder-buffer entries that must be flushed. The last model corresponds to a processor that can execute an enormous number of concurrent instructions. Note that there is a trade-off here that we are over-simplifying for the sake of brevity: as the issue width grows, so does the cost of taking an interrupt, but so does the machine's IPC. An N-entry reorder buffer will not have a fixed N-cycle interrupt cost. We also approximate the performance of 64-entry TLBs, which are more commonly found in today's microprocessors. These numbers are extrapolated based on our measurements for mean free path.

Figures 11, 12, and 13 show the results for GCC, VORTEX, and IJPEG, respectively. Each figure presents graphs for three different L1 cache sizes: 16KB, 64KB, and 256KB, where each cache is split into I- and D-caches with identical cache line configurations. Each graph within the figure gives numbers for the five VM simulations as well as the BASE case (running the application alone on the caches without any VM overhead). We note the following:

- Comparing the BASE case to the other simulations shows that the addition of virtual memory code (handlers) and data (page tables) increases application L1 & L2 cache misses by 5% (for GCC) to 20% (for VORTEX). This effect is not normally mentioned in studies measuring VM overhead because it requires one to run the application without the VM system to make a comparison. The result is that this doubles the net effect of the VM system on total performance. This is almost entirely due to contention with the page table (most increases are seen in *L1d-miss* and *L2d-miss* and not in *L1i-miss* or *L2i-miss*): the VM code affects an application's I-cache behavior less than the application conflicts with itself. For GCC, the overhead is seen in the L2 cache, not the L1 cache, meaning the additional overhead imposed by the VM system is less significant than GCC simply not fitting in the L1 cache.
- The cost of interrupts is large and (for TLB-based systems) unaffected by cache size—as cache overhead decreases, the interrupt cost becomes more prominent. Interrupt overhead is as high as VMCPI, and can account for 10% of the total overhead. For the 200-cycle interrupt costs (i.e., for future systems with larger interrupt overheads), this number is even higher.
- For the software-managed cache (NOTLB), interrupt cost decreases significantly with increased L2 cache sizes: for large caches, interrupts represent a small fraction of total overhead.
- The INTEL scheme does better than all other hierarchical page tables, largely due to its lack of interrupt overhead.

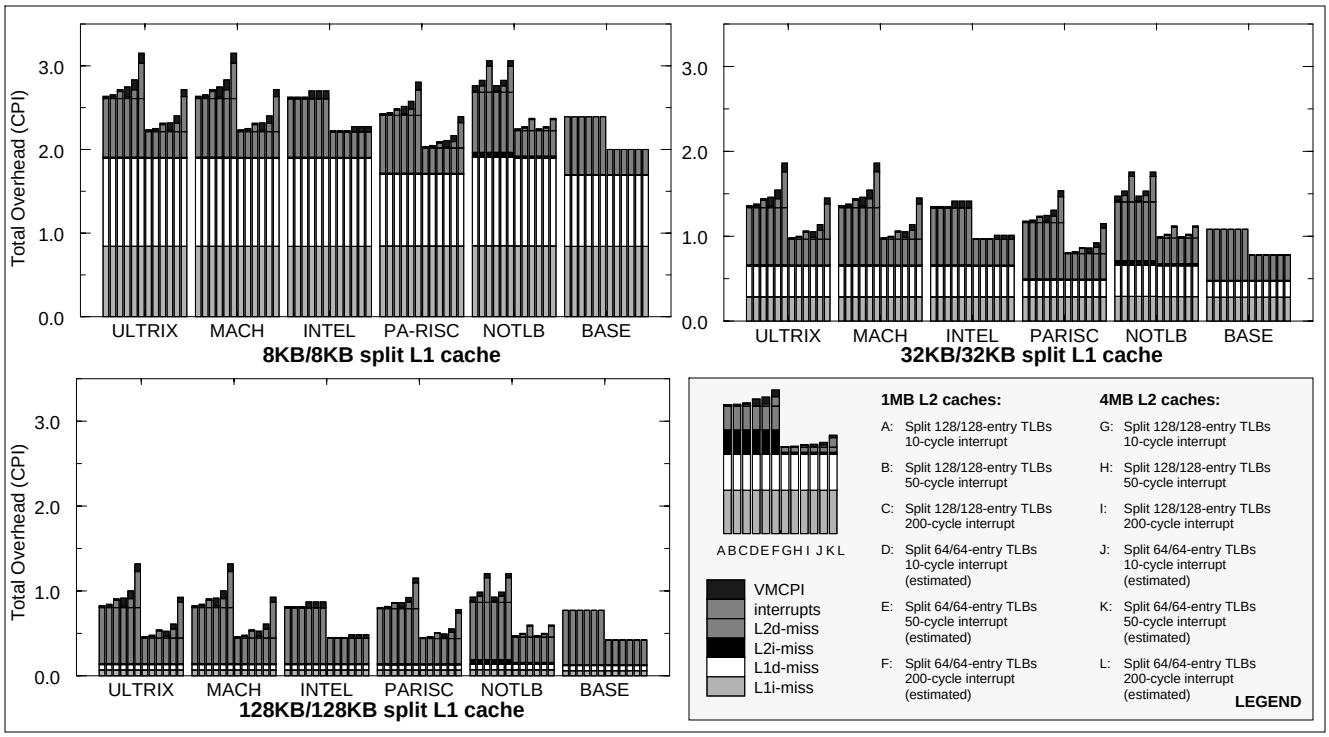

Figure 12: Total overhead — VORTEX. *Total overhead* includes application cache misses, interrupt costs, and virtual memory overhead; it is additive to base instruction-execution costs. All graphs represent 64/128-byte L1/L2 linesizes. For each VM simulation we show the performance of twelve configurations, covering 1MB L2 caches, 4MB L2 caches, 128/128-entry TLBs, 64/64-entry TLBs, and three different interrupt costs. Note that 64-entry TLB performance is estimated.

- IJPEG presents some interesting counterexamples: its results show that PA-RISC does worse than the other schemes; that there is a significant difference between the MACH and ULTRIX simulations; and that VMCPI is often larger than the interrupt overhead. These behaviors are largely due to the low level of overhead—the total is well below 1 CPI, so effects like compulsory misses are actually a large portion of the overhead.
- Cache sizes are extremely important: increasing L1 or L2 cache sizes by a factor of four can decrease total overhead by 50%.

One conclusion is that large TLBs are necessary. However, they do not solve the whole problem because copy-on-write (used extensively in operating systems such as Mach and Windows NT [23, 8]) can be disastrous—every copy-on-write causes a TLB protection violation. If used as frequently as in Mach (every 40,000 instructions [13]), this could increase each of the interrupt-overhead numbers by 50%.

We see that the total cost of memory management includes the execution of the virtual memory system (adding roughly 5-10% to total execution time), the cost of taking interrupts to handle the memory-management events (another 5-10%), and the increased number of cache misses seen by the application (another 5-10%). This adds up to a significant cost that becomes more noticeable as caches and reorder buffers increase in size.

5 CONCLUSIONS

This paper presents a trace-based study of five virtual memory designs, including combinations of hierarchical and inverted page tables on hardware-managed and software-managed TLBs (as well as a system with no TLBs). The primary goal of the study is to understand the performance differences that are due to one's choice of memory-management unit and page table organization.

We find a number of interesting results. Against common intuition, the x86 memory management unit performs better than all other mechanisms, despite the fact that it makes two memory references per TLB miss. An inverted page table tends to impact the data caches significantly less than a hierarchical page table, even when its entries are four times the size of the hierarchical table's entries. Memory management systems increase application cache misses by 5-10%, which roughly doubles the total impact of the VM system. Precise interrupts are a source of overhead that is not hidden by increased cache sizes or increased TLB sizes beyond 256 entries (the point at which copy-on-write effects would dominate, if used frequently). Software-managed caches are a viable design, provided that the caches and cache linesizes are large enough. The choice of page table and MMU is not likely to confer an enormous performance advantage, therefore the large performance differences in TLB handling reported in previous studies are likely due to implementation.

Future systems are likely to make much better use of superpages at the application level [29], thereby reducing all overheads related to virtual-memory, including interrupts (note that this study only considers 4KB page sizes). Future processors will also execute increasing numbers of concurrent instructions, thereby needing larger issue windows and reorder buffers. If the implementation of VM-related interrupts is not changed significantly, the individual cost of interrupts will increase in these future processors, competing with the tendency of superpages to bring the frequency of VM-related interrupts down. A simple solution is a hardware-managed TLB with a hardware-walked inverted table; it is also worth considering a compromise between performance and flexibility in the form of a software-configurable state machine that walks the page table in a software-defined manner.

In general, the performance differences between the systems studied here are not large enough to prefer one over another, which supports our previous suggestions that the industry standardize on some VM interface [14, 15]. However, if interrupts do remain a problem, it is interesting that the two schemes that are least dependent on the cost of interrupts as caches grow larger are the purely hardware-managed scheme, INTEL, and the purely software-managed scheme, NOTLB.

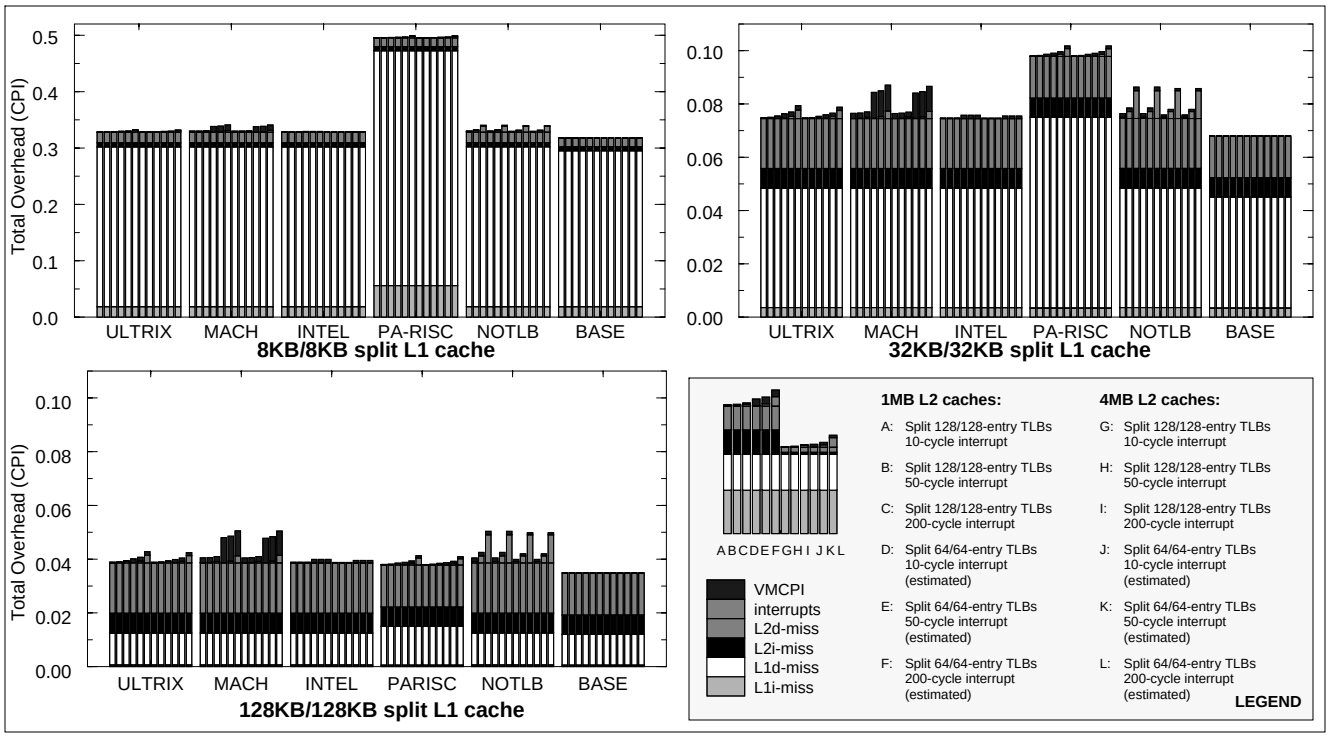

Figure 13: Total overhead — IJPEG. *Total overhead* includes application cache misses, interrupt costs, and virtual memory overhead; it is additive to base instruction-execution costs. All graphs represent 64/128-byte L1/L2 linesizes. For each VM simulation we show the performance of twelve configurations, covering 1MB L2 caches, 4MB L2 caches, 128/128-entry TLBs, 64/64-entry TLBs, and three different interrupt costs. Note that 64-entry TLB performance is estimated.

REFERENCES

[1] T. E. Anderson, et al. "The interaction of architecture and operating system design." In *Proc. ASPLOS-4*, April 1991, pp. 108–120.

[2] A. W. Appel and K. Li. "Virtual memory primitives for user programs." In *Proc. ASPLOS-4*, April 1991, pp. 96–107.

[3] K. Bala, M. F. Kaashoek, and W. E. Weihl. "Software prefetching and caching for translation lookaside buffers." In *Proc. OSDI-1*, Nov. 1994.

[4] J. B. Chen, et al. "The measured performance of personal computer operating systems." In *Proc. SOSP-15*, December 1995, pp. 299–313.

[5] J. B. Chen, A. Borg, and N. P. Jouppi. "A simulation based study of TLB performance." In *Proc. ISCA-19*, May 1992.

[6] D. R. Cheriton, G. A. Slavenburg, and P. D. Boyle. "Software-controlled caches in the VMP multiprocessor." In *Proc. ISCA-13*, January 1986.

[7] D. W. Clark and J. S. Emer. "Performance of the VAX-11/780 translation buffer." *ACM Trans. Comp. Sys*, vol. 3, no. 1, February 1985.

[8] H. Custer. *Inside Windows NT*. Microsoft Press, Redmond WA, 1993.

[9] D. R. Engler, S. K. Gupta, and M. F. Kaashoek. "AVM: Application-level virtual memory." In *Proc. HotOS-V*, May 1995.

[10] D. S. Henry. "Adding fast interrupts to superscalar processors." Tech. Rep. Memo-366, MIT Computation Structures Group, December 1994.

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

[12] J. Huck and J. Hays. "Architectural support for translation table management in large address space machines." In *ISCA-20*, May 1993.

[13] B. L. Jacob and T. N. Mudge. "Software-managed address translation." In *Proc. HPCA-3*, February 1997, pp. 156–167.

[14] B. L. Jacob and T. N. Mudge. "Virtual memory in contemporary microprocessors." *IEEE Micro*, vol. 18, no. 4, July/August 1998.

[15] B. L. Jacob and T. N. Mudge. "Virtual memory: Issues of implementation." *IEEE Computer*, vol. 31, no. 6, pp. 33–43, June 1998.

[16] T. L. Johnson and W.-M. W. Hwu. "Run-time adaptive cache hierarchy management via reference analysis." In *Proc. ISCA-24*, June 1997.

[17] D. E. Knuth. *The Art of Computer Programming–Volume 3 (Sorting and Searching)*. Addison-Wesley, 1973.

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

[19] C. May, et al, Eds. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, 1994.

[20] M. Moudgill and S. Vassiliadis. "Precise interrupts." *IEEE Micro*, vol. 16, no. 1, pp. 58–67, February 1996.

[21] D. Nagle, et al. "Optimal allocation of on-chip memory for multiple-API operating systems." In *Proc. ISCA-21*, April 1994.

[22] D. Nagle, et al. "Design tradeoffs for software-managed TLBs." In *Proc. ISCA-20*, May 1993.

[23] R. Rashid, et al. "Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures." *IEEE Transactions on Computers*, vol. 37, no. 8, pp. 896–908, August 1988.

[24] J. A. Rivers and E. S. Davidson. "Reducing conflicts in direct-mapped caches with a temporality-based design." In *Proc. ICPP-96*, Aug. 1996.

[25] M. Rosenblum, et al. "The impact of architectural trends on operating system performance." In *Proc. SOSP-15*, December 1995.

[26] J. E. Smith, G. E. Dermer, and M. A. Goldsmith. *Computer System Employing Virtual Memory*. US Patent Office, no. 4,774,659, Sep. 1988.

[27] J. E. Smith and A. R. Pleszkun. "Implementing precise interrupts in pipelined processors." *IEEE Trans. Computers*, vol. 37, no. 5, May 1988.

[28] G. S. Sohi and S. Vajapeyam. "Instruction issue logic for high-performance, interruptable pipelined processors." In *ISCA-14*, June '87.

[29] M. Talluri and M. D. Hill. "Surpassing the TLB performance of superpages with less operating system support." In *ASPLOS-6*, Oct. '94.

[30] M. Talluri, M. D. Hill, and Y. A. Khalidi. "A new page table for 64-bit address spaces." In *Proc. SOSP-15*, December 1995.

[31] C. A. Thekkath and H. M. Levy. "Hardware and software support for efficient exception handling." In *Proc. ASPLOS-6*, October 1994.

[32] G. Tyson, et al. "A modified approach to data cache management." In *Proc. MICRO-28*, November 1995, pp. 93–103.

[33] M. Upton. *Personal communication*. 1997.

[34] W. Walker and H. G. Cragon. "Interrupt processing in concurrent processors." *IEEE Computer*, vol. 28, no. 6, June 1995.

[35] B. Wheeler and B. N. Bershad. "Consistency management for virtually indexed caches." In *Proc. ASPLOS-5*, October 1992, pp. 124–136.

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