AN IN-CACHE ADDRESS TRANSLATION MECHANISM

David A. Wood, Susan J. Eggers, Garth Gibson, Mark D. Hill, Joan M. Pendleton, Scott A. Ritchie, George S. Taylor, Randy H. Katz, and David A. Patterson

Computer Science Division
Electrical Engineering and Computer Science Department
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT: In the design of SPUR, a high-performance multiprocessor workstation, the use of large caches and hardware-supported cache consistency suggests a new approach to virtual address translation. By performing translation in each processor's virtually-tagged cache, the need for separate translation lookaside buffers (TLBs) is eliminated. Eliminating the TLB substantially reduces the hardware cost and complexity of the translation mechanism and eliminates the translation consistency problem. Trace-driven simulations show that normal cache behavior is only minimally affected by caching page table entries, and that in many cases, using a separate device would actually reduce system performance.

Key Words and Phrases: Virtual Memory Management, Multiprocessor Architectures, Trace-Driven Simulation

1. Introduction

Paged virtual memory is used in most computer systems to extend the address space available to the programmer [Denn70]. Programs execute using virtual addresses, which are translated by the system into physical addresses at run-time. Virtual memory allows program size to be independent of the amount of available physical memory, and eliminates the problems of contiguous physical memory allocation.

The mapping from virtual addresses to physical addresses is maintained in a structure called a *page table*. A virtual address is used to index into the table and locate the corresponding *page table entry* (PTE); the entry is used to construct the physical address. This translation process is usually accelerated by special hardware called a *translation lookaside buffer* (TLB)[1]. A TLB is a small cache, typically 64 to 512 entries, of recently-referenced page table entries. Like all caches, the TLB reduces the *average* access time to a PTE, thus reducing the overhead of virtual address translation [Saty81].

In this paper we describe a new translation mechanism, called *in-cache address translation*, that uses the virtually-tagged data cache instead of a TLB to hold page table entries [Ritc85]. In-cache translation requires less hardware, since it eliminates the TLB, and has comparable performance to TLB-based mechanisms when combined with a large cache. Performance depends critically on the cache memory having low miss rates; fortunately, the increased density of RAM chips makes large caches ($\geq$64K bytes) feasible.

In-cache address translation is being implemented as part of the SPUR workstation project at U.C. Berkeley. The SPUR workstation [Hill85] is a high performance personal computer that has evolved from the previous RISC [Patt85] and SOAR [Unga84] research projects. SPUR is a single bus, shared-memory multiprocessor, containing 6 to 12 processors with private caches. The prototype of this machine will serve as a test-bed for parallel processing research.

[1] Also known as a *directory lookaside table* (DLAT) or *translation buffer* (TB).

Shared-memory multiprocessors with cache memories suffer from the well known problem of *cache coherency* [Tang76, Cens78]. Solutions to this problem guarantee that all processors see a consistent view of memory. For shared bus multiprocessors, this is often accomplished using extra hardware that monitors transactions on the bus, as is done in SPUR [Katz85]. Since a TLB is nothing more than a special purpose cache, if each processor has its own TLB[2], then a multiprocessor also suffers from a *TLB coherency problem*. Thus changes to a page table entry (e.g., making a page inaccessible so it can be flushed to the paging store) must be reflected in the TLBs of all processors. The data cache solution also works for TLBs, but requires significant additional hardware. Since in-cache translation does not use a TLB, it eliminates the TLB coherency problem.

Section 2 describes the basic mechanisms and algorithms needed for a uniprocessor system. Section 3 extends the scheme to shared-bus multiprocessors employing a hardware cache consistency algorithm. In section 4, we describe a performance evaluation using trace-driven simulation, and present the results. Finally, we discuss the implementation status of in-cache translation in SPUR and summarize our results.

2. Uniprocessor In-Cache Address Translation

This section describes the basic mechanisms and algorithms of in-cache translation. First, the motivation for the virtually-tagged cache is explained, followed by a discussion of its problems and their solutions. Then the page table organization and conceptual translation process is described. Next, the actual in-cache address translation process is described. Finally, the details of memory management are discussed for completeness.

2.1. Overview of Translation Scheme

The in-cache translation mechanism assumes that the cache is addressed using virtual addresses, i.e., the cache index and tag are derived from virtual, rather than physical, addresses. The advantage of this virtually-tagged cache is that address translation is required only on a cache miss, when the data must be fetched from main memory. In contrast, if physical addresses are used for cache access, the translation must occur on *every* cache reference. Many systems with physical address caches perform address translation in parallel with the cache access, deriving the index from the virtual address and the tag from the physical address. In general, this requires the cache index bits to be the same in both virtual and physical addresses. However, as cache sizes increase for a fixed page size, the complexity (e.g., associativity) must also increase for parallel translation to be possible[3]. For example, a 16K byte cache with 4K byte pages must

[2] Most current generation multiprocessors utilize a TLB per processor. Some recent research systems, e.g., MIPS-X [Stan85] and Dragon [McCr84], are investigating centralized and hybrid translation buffers.

[3] Other ways to satisfy the restriction are to increase the page size, or to restrict the mapping of virtual pages to physical pageframes such that some low order bits of the virtual page number and pageframe number are the same.

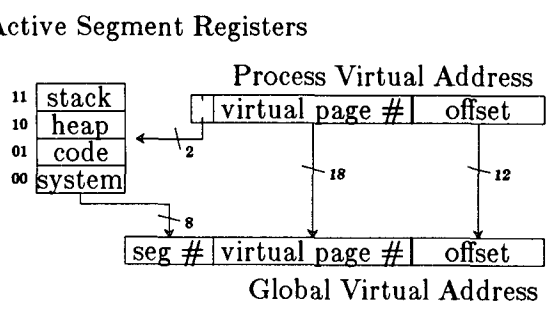

Figure 1 : Formation of a Global Virtual Address

The virtual memory allows for multiple large address spaces by providing one large global virtual address space. Each process's virtual address space is divided into four segments, typically stack, heap, code, and system. The global virtual address is formed in the cache controller by prepending the global segment number, from one of the four active segment registers, to the process virtual address.

be 4-way set-associative. For much larger caches, the associativity becomes unmanageable, and translation is usually done sequentially. In this case, translation must be fast to achieve reasonable effective memory access times. On the other hand, cache misses only occur on a small percentage of the total references, and therefore the translation mechanism used with a virtually-tagged cache need not be as fast to yield comparable performance.

Virtually-tagged caches suffer from the address *synonym* problem [Smit82]. If two virtual addresses are allowed to map to the same physical address, the same data could be in two separate cache entries; if modifications to one entry are not reflected by the other, inconsistent results can occur. We avoid this problem by providing a single *global* virtual address space, shared by all processes. When two processes share data, they must use the same global virtual address to access it.

In SPUR, the global virtual address space is 256 gigabytes (2^{38}), divided into 256 1-gigabyte segments. Each process sees a 4-gigabyte process virtual address space, composed of 4 1-gigabyte segments. Each process segment is independently mapped to one of the 256 global segments. As Figure 1 shows, the top two bits of the process virtual address select one of four *active segment registers*, which define the current mapping. Processes share data by mapping to the same global segment. If any portion of a segment is shared, then the whole segment is shared. The segment register can be accessed in parallel with the cache, since the segment number is not needed until the tag comparison. A similar scheme has been implemented in the CDC Cyber 180 [CDC84] and variations are discussed by Knapp in her dissertation [Knap85].

Each segment is divided into 256K 4K byte pages. Over 64 million page table entries (2^8 *segments* $\times 2^{18}$ *pages/segment* $= 2^{26}$ *PTEs*) are needed to map the entire global address space, for a total page table size of 256 megabytes. To avoid excessive memory requirements, the page tables are placed in virtual space, and therefore may be paged out to disk. Rather than residing in a separate *system virtual space*, as is done in the VAX-11 architecture [Digi81], the page tables may reside anywhere in the global virtual space. Since the page tables map the entire global virtual space, some portions of the page tables map themselves. These portions are referred to as second-level or *root* page tables, to denote their special significance. The root page tables must be kept resident at known addresses within physical memory, requiring a minimum of 256K bytes of physical memory to map the entire global virtual address space.

2.2. In-cache Translation Process

Conventional systems with virtual caches have used a separate TLB to cache page table entries [Smit82]. In our approach, page table entries can reside in the same cache as instructions and data, and are referenced with virtual addresses. Translation is accomplished by fetching PTEs from the cache instead of a TLB. Since translation is necessary only on cache misses, the impact of the additional delay on performance is minor.

To translate a virtual address, we need to fetch its PTE to determine the physical address. To do this, the in-cache mechanism must compute the PTE's global virtual address. By requiring all page tables to be contiguous in virtual space, the PTE can be found by using the (data) virtual page number as an index into the page table. By further requiring the page tables to be aligned on a 256M byte boundary, the address computation can be performed by a simple shift and concatenate, as illustrated in Figure 3.

In this paper, we assume that the processor issues a memory request and remains stalled until the request is completed. A separate cache controller performs all the address computation and

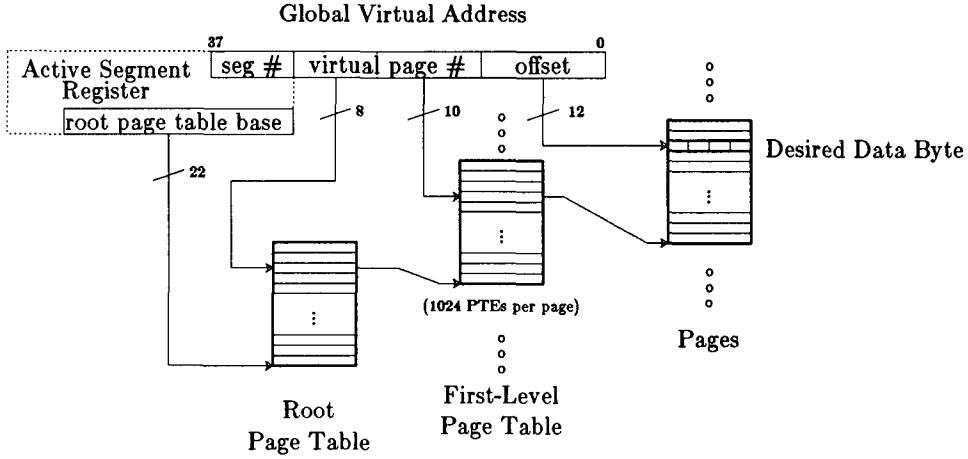

Figure 2 : Two-Level Page Tables for an Active Segment

Associated with the global segment number of an active segment is the base address of the root page table for that segment. The high-order eight bits of the virtual page number index into the root page table to find the physical address of the "first-level" page table page. The low-order ten bits of the virtual page number select the page table entry for the desired data page. The offset field then specifies the byte within the page.

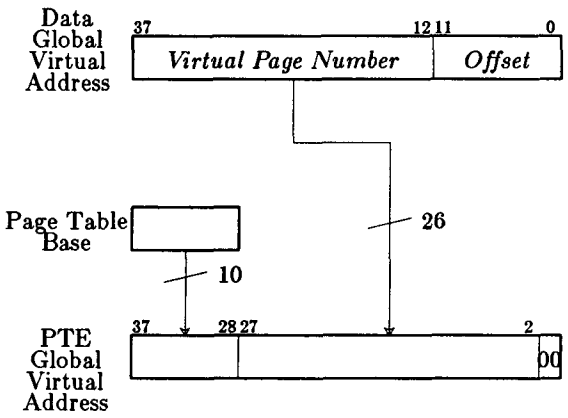

Figure 3 : Formation of Page Table Entry Global Virtual Address

The virtual address of the PTE is formed by using the virtual page number portion of the (data) global virtual address as an index into the page table. Because the page tables must be aligned on a 256M byte boundry, the address may be formed using concatenation rather than addition. Since PTEs are 4 bytes, the bottom two bits of the byte address must be 0.

logic required to service a miss, including address translation.

Figure 4 shows the four cases that can occur on a memory reference. In the most frequent case (case A), the cache hits and data is delivered in one cycle (assuming a single-cycle cache). In preparation for a miss, a shift-and-concatenate circuit in the cache controller forms the global virtual address of the page table entry during the reference. If translation is required, the cache controller uses this address and attempts to read the page table entry in the following cycle. Case B corresponds to a cache miss on the data and a hit on the PTE. This requires one additional cache reference, for the PTE, and one memory transfer to fetch the desired data block.

If the PTE is not in the cache (case C), a third cache reference is required for the root PTE, from which the physical address of the PTE may be formed. After the PTE is fetched from memory, it is loaded into the cache for use in future translations. In the worst case (case D), all three cache references fail, and the root PTE must also be fetched from memory and cached. The physical addresses of the root page table for each of the four active segments are kept in registers in the cache controller.

More than three memory operations may occur if a *write-back* of a dirty cache block is necessary. Because the cache is virtually-tagged, a recursive translation must be performed to obtain the physical address needed for the write-back. If the PTE for the replaced block must be fetched from memory, it should be placed in a separate register to prevent deadlock: deadlock could occur if a PTE tries to displace the block being written back to memory. In SPUR, this complexity is eliminated by keeping the physical address tag for each block in cache tag memory (as is also done in the Dragon [McCr84]). We trade off control complexity for the additional bits of cache tag memory.

On examining any page table entry, the desired page may be shown to be *invalid*, indicating that the page is not in memory but resides on disk. In this event, a trap to the page fault handler is taken, and the page fault is resolved in software.

2.3. Reference and Dirty Bits

In most systems, *reference* and *dirty* bits for each page are kept in the PTE to optimize the replacement and write-back of pages in memory. Traditionally, copies of these bits are kept in the TLB, checked on each cache reference, and are set when necessary. For in-cache translation, maintaining a true reference bit requires that the corresponding page table entry be examined for *every* reference to the cache. This overhead would be prohibitive; instead, we use an approximation to reference bits, which we refer to as *miss* bits. The PTE miss bit is set only when a reference to a cache block misses. In this event, the PTE must be brought into the cache anyway to carry out the address translation. After a miss bit is cleared by the paging daemon, it is not set again until the next cache miss for that page. Thus it is possible, although unlikely, for a page to be thrown out of memory despite being referenced. Since even true reference bits only allow the operating system to *approximate* a "Least Recently Used" policy, we feel that this additional approximation is not a serious shortcoming[4].

Dirty bits could be maintained in the same way, by setting the

[4] At least one very successful architecture, the VAX-11 [Digi81], does not provide reference bits at all.

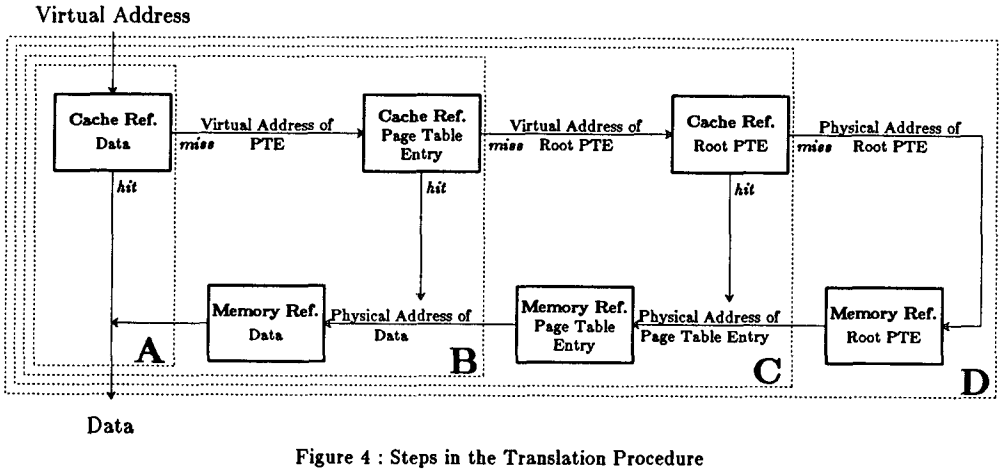

Figure 4 : Steps in the Translation Procedure

Four cases are possible depending on whether the cache contains the data, page table entry, and root PTE. **A**: The cache hits, and no translation is required. **B**: The first cache reference misses, but the cache contains the page table entry (a "TLB hit"). **C**: The second cache reference misses, but the cache contains the root PTE. **D**: The cache misses on all three attempts, and the root page table entry must be fetched from memory.

dirty bit whenever a writable block is brought into the cache. This has the disadvantage that the dirty bit may be set even though the page has not been modified, and therefore unmodified pages may be written to disk unnecessarily. Another solution is to examine the PTE on every processor write; however, this results in a significant degradation in performance. Instead, every cache entry contains a *PageDirty* bit. The PTE dirty bit is copied into the PageDirty bit when a block is brought into the cache. When the processor writes to a block whose PageDirty bit is set, i.e., the page is known to be dirty, then the write proceeds normally. If this bit is not set, then the PTE dirty bit *may not* have been set yet. The cache controller must first reference the PTE (potentially causing a cache miss), check the dirty bit, and set it if not already set. Then the referenced cache block's PageDirty bit is set (since the page is now known to be dirty), and the write is allowed to proceed.

The PageDirty bits of other blocks that were read from the page before it was written are not updated when the PTE dirty bit was set. Thus, when one of these blocks is modified, the PageDirty bit is still zero, and the cache controller must check the PTE dirty bit.

In the SPUR prototype, when the processor writes to a block whose PageDirty bit is not set, it handles the reference like a cache miss. When the PTE is referenced, the PTE dirty bit is checked. If it is set (a dirty bit miss), then the miss handling completes and the write is allowed to proceed. If the dirty bit is not set, a trap is generated (a dirty bit trap) and the PTE is modified under software control. By handling the PageDirty bit in this manner, the control logic is greatly simplified, and, as we will show later, these dirty bit misses occur infrequently.

When the page is recycled (see below) or written back to disk, the page's dirty bit is cleared. Care must be taken to first flush all that page's blocks from the cache; this guarantees that subsequent writes to that page won't find a block remaining in the cache with the PageDirty bit incorrectly set. Since this must be done infrequently, the performance impact of the flushes is slight.

2.4. Segment and Pageframe Recycling

The SPUR workstation supports 256 global segments. This should be sufficient for the number of concurrent processes on a personal workstation. Processes have finite lifetimes, however, so segments will need to be reassigned from terminated processes to newly active ones (*recycled*). Because of the virtually-tagged cache, the system must guarantee that no blocks from the deactivated segment's pages are resident in the cache before it is reassigned. Otherwise, a process accessing the reassigned segment might reference old data.

A similar problem arises when we want to reuse a physical pageframe. As before, we want to change the virtual to physical mapping, but this change is not immediately reflected in the cache since it is accessed by virtual addresses. If we simply remap the pageframe to a different virtual page, then there may be blocks from the original virtual page remaining in the cache. Since, in SPUR, the physical address tag is cached with each entry, blocks mapped to the old virtual address will still be written-back to the pageframe[6]. When the old block is written-back, it could over-write new data. To prevent this loss of data, it is necessary to guarantee that a page's dirty blocks are removed from the cache before re-using the page's pageframe.

Both of these problems can be solved by flushing the cache. If a number of pages and segments are recycled together, then the overhead is not significant. Other algorithms exist which may have less overhead in some cases.

[6] Note this problem does not exist if we do *not* cache the physical tags. However, we would incur the performance loss and complexity of translating on write-back.

3. Extensions for Multiprocessor In-Cache Translation

Traditional multiprocessors use TLBs to perform address translation. As mentioned previously, multiple TLBs experience a translation coherency problem analogous to the cache coherency problem. Existing systems have solved this problem using a combination of hardware and software. At one extreme, a special instruction can be provided to invalidate remote TLBs. At the other, the software must verify that the TLB entry is valid before using it. Some new multiprocessors with virtually-tagged caches have a single TLB shared by all processors [Stan85, McCr84]. This solution eliminates the coherency problem, but may result in a performance bottleneck at the TLB.

In-cache translation is easily extended to multiprocessors that use hardware to enforce cache coherency [Katz85]. Page tables are only cached in the processor's data cache, and not in a separate translation buffer. Therefore, updates to cached PTEs are done in a consistent manner, as guaranteed by the regular cache coherency mechanism.

The basic translation process is unchanged for multiprocessors, but recycling pages and segments is slightly more complicated. When recycling a page (or segment), there must be no blocks from that page (or segment) in *any* processor's cache. Each processor can be instructed to flush its cache by sending it a message or interrupt. The completion of the operation can be synchronized using a counter in shared memory; each processor atomically incrementing it upon completion of its flush.

4. Evaluation

The previous sections have described the operation of the in-cache translation mechanism. While this translation scheme eliminates the TLB and the TLB coherency problem, it must also have acceptable performance to be useful. One reasonable definition of "acceptable", examined below, is that it should perform at least as well as existing translation mechanisms.

4.1. Methodology

We used the DineroII cache simulator [Hill83] to evaluate the performance of the in-cache translation mechanism and other translation buffers. This simulator uses address traces as input and reports miss rates and bus traffic for specified cache parameters. We modified the simulator to simulate the caching of page table entries. All simulations were for a uniprocessor and do not take into account additional misses that might result from maintaining cache coherency. However, since pagetables are updated relatively infrequently, we do not expect cache coherency to have a major impact on performance.

Table 1 lists the five address traces that were used to drive the simulations, and the amount of virtual memory referenced by each trace. The first four were gathered on a VAX running UNIX with an address and instruction tracer [Henr84]. LISZT is the Franz LISP compiler compiling a portion of itself. VAXIMA is an algebraic manipulation system, written in LISP, performing a series of integrations, matrix operations, and solving differential equations. CS20K and CS100K are traces composed of two separate sections of the VAXIMA trace, and are designed to simulate context switching. They are identical except for the switching interval, which is 20,000 and 100,000 references, respectively. Since, in SPUR, independent (non-sharing) processes run in different segments, the two traces were given different segment numbers to simulate this behavior. The final trace, MVS, is a series of calls to the MVS operating system and was traced on an Amdahl 470 [Smit85]. The traces gathered on the VAX include only those references made while in user mode. The MVS trace, on the other hand, includes only system references.

The available computer resources forced us to limit the simulations to one million addresses per trace, even though this length represents under one second of execution. As a sensitivity analysis, traces of five million references were run for selected cases, and miss

Address Traces Used			
Trace	Description	Memory Referenced (Mbytes)	(pages)
LISZT	Franz LISP self-compilation	0.6Mb	145
VAXIMA	Algebraic expert system (a derivative of MACSYMA)	1.7Mb	413
CS20K	Two VAXIMA streams interleaved every 20K references (Multi-user context switch rate)	2.5Mb	609
CS100K	Two VAXIMA streams interleaved every 100K references (single-user context switch rate)	2.5Mb	609
MVS	Multiple calls to the MVS operating system	3.7Mb	284

Table 1 : Address Traces Used

These five traces were used in the analysis of the in-cache translation scheme. The first four were generated on a VAX running UNIX. The last trace was recorded on an Amdahl 470 running the MVS operating system. The amount of virtual memory referenced is shown by the number of 4K byte pages that were touched (4K bytes is the page size used in SPUR).

rates did not differ substantially. All digits of the results are significant for the particular traces that were simulated. However, as workload behavior varies wildly [Smit85], care should be taken in interpreting these results.

The in-cache translation scheme was simulated assuming a 128K byte, direct-mapped (associativity=1) cache, with 32 byte blocks. The memory page size was 4K bytes. These parameters are being used in the prototype of the SPUR multiprocessor workstation.

While trace-driven simulation has been shown to provide optimistic results [Clar85], it is nonetheless useful for making relative comparisons.

4.2. Performance of In-Cache Translation

There are two opposing views of the SPUR cache: a cache being corrupted by page table entries, and a translation buffer being polluted by instructions and data. Table 2 shows the increase in the cache miss rate because both functions are being performed in the cache. This total additional miss rate is computed by dividing the misses added when PTEs are cached by the total number of references made to the cache *by the processor*.

There is an important distinction to be made: *processor* references to the cache are for instructions and data, while the *cache controller* references the cache for page table entries during the translation process. The 5th and 6th columns of Table 2 separate the total additional miss rate according to this distinction. In the column labeled "Collisions", the processor is experiencing additional misses on instructions and data because normal cache contents are being displaced by PTEs. The "PTE Misses" column, on the other hand, reflects the additional misses incurred when the cache is being referenced during the translation process. The combination of "Collisions" and "PTE misses" is the measure of the translation mechanism *performance*: the "TLB" miss rate for SPUR. This figure of merit includes the misses on both page table and root page table references. As we shall see, there are few root page table references.

Effective, or average, memory access time is another important performance metric. As was shown in Figure 4, a memory reference can cause 4 different cases to occur. The average access time is computed by summing the products of the frequency of each case times the cycles required by that case. We assume that cache accesses complete in 1 cycle, and memory accesses complete in 13 (5 cycles latency and 8 cycles transfer time); these parameters are based on the implementation of the SPUR prototype.

Table 3 displays the percent of memory references handled by each of the four cases. Between 97% and 99% of the time, the reference hits in the cache and is handled in one cycle (A). The cache controller takes over on a miss, and in the next cycle references the cache with the virtual address of the page table entry. From 91% to 97% of these references, or from 0.5% to 2.5% of all references (B), are cache hits. The desired instruction or data can then be fetched from memory in one bus transaction.

For all the traces except MVS, only 2 to 4 references out of 10,000 result in a PTE miss and force a memory access for the page table entry (C). Only about 2 in 100,000 references cause a "double-miss," and require a memory fetch of the root page table entry as well (D). These second level lookups represent only 3.3% of PTE misses on average. These results agree with those of Clark and Emer [Clar85], that between 3.1 to 4.8% of PTE misses are for second level PTEs, and support the use of a two-level page table scheme.

4.3. Comparison to a Separate TLB

To properly evaluate in-cache translation, we must compare its performance to some standard. One reasonable standard is the performance of existing translation mechanisms. The most common translation mechanism is a TLB combined with a physically-addressed cache. Table 4 shows how in-cache translation compares to the translation buffers of several commercial computers.

The in-cache method displays lower miss rates for almost all cases. Of the TLBs shown, only the large set-associative buffers achieve better performance. By allowing a large, variable number of cache entries to hold PTEs, the in-cache scheme is able to adapt to the dynamics of program behavior. As seen in the last column of

Increase in Cache Miss Rate						Average Number of PTEs in SPUR Cache
Trace	Miss Rate (%) Pure Cache	Miss Rate (%) w/Translation	Additional Cache Miss Rate (%) Total(% increase)	Collisions	PTE Misses	
LISZT	0.584	0.610	0.026(4.4%)	0.009	0.016	211
VAXIMA	1.844	1.875	0.030(1.6%)	0.004	0.026	598
CS100K	2.214	2.260	0.046(2.1%)	0.005	0.041	928
CS20K	2.445	2.495	0.050(2.0%)	0.007	0.042	933
MVS	1.677	1.981	0.304(18.1%)	0.142	0.162	329

Table 2 : Additional Cache Misses Due To In-Cache Translation

This table shows the increase in cache misses when translation is performed in-cache. For example, the miss rate for VAXIMA is 1.844% when PTEs are not cached and 1.875% when PTEs are cached, a relative increase of 1.6%. The additional 0.03% is composed of two elements: extra misses when the processor references instruction and data blocks that have collided with PTEs (0.004%), and the misses when the cache controller references a page table entry (0.026%). The last column shows the average number of PTEs resident in the cache; less than 3% of the cache entries are used to store PTEs.

Types of Memory Accesses					
Trace	Percentage of Total References				Average Access Time (cycles)
	Cache Hits A (1$)	PTE Hits B (2$,1M)	RPTE Hits C (3$,2M)	RPTE Misses D (3$,3M)	
LISZT	99.4065	0.5775	0.0158	0.0002	1.0854
VAXIMA	98.1517	1.8236	0.0231	0.0016	1.2624
CS100K	97.7806	2.1805	0.0373	0.0016	1.3164
CS20K	97.5478	2.4115	0.0390	0.0017	1.3492
MVS	98.1804	1.6600	0.1576	0.0020	1.2773

Table 3 : Breakdown of Memory References

This table shows the percentage of total references to memory that fall into the four categories of Figure 4. The average number of cycles per reference is given for each trace. A "$" represents a cache reference requiring 1 cycle, and an "M" indicates a memory transaction requiring 13 cycles. Note that the cache hit ratios presented for case A include the effect of PTEs colliding with instructions and data. Thus they correspond to the "pure" cache miss rates *plus* the collision miss rates of Table 2.

Table 2, the average number of PTEs in the cache increases for more poorly behaved traces. Clearly, the performance of in-cache translation is at least comparable to popular high-performance machines. Therefore, we conclude that in-cache translation has acceptable performance.

Much of in-cache's performance obviously results from the large virtually-tagged cache, since translation is only required on cache misses. As an alternative standard of comparison, we can examine the performance of a TLB placed *after* the SPUR cache (i.e., using a TLB for address translation rather than in-cache translation). This comparison factors out the effects of the virtually-tagged cache.

The results for this comparison were generated by simulating the performance of the 128K byte, direct-mapped cache for each trace, and recording only those addresses that missed. These addresses were then used as the input to each of the simulated translation buffers.

Table 5 shows the commercial translation buffers examined in Table 4, but this time translating only on cache misses. Only VAXIMA and MVS are shown for brevity, but the results are comparable for the other traces. The "Cache Miss" columns have identical entries because the SPUR 128K byte cache was simulated in each case. The TLB miss rate is still normalized to processor references, rather than TLB references, to facilitate comparisons. The cycles required for the average reference are calculated as in Table 3; a TLB miss is assumed to take as much time as a cache miss (13 cycles). This assumption is somewhat pessimistic, as TLB misses often take much longer than cache misses. The relative performance, effective cycle times of TLBs over in-cache translation, is displayed in the last column of each section.

In-cache translation has lower TLB miss rates than many of the buffers; however, in one of the cases the effective cache access time is worse. This occurs because a TLB may be accessed in parallel with the cache access, reducing the cost of a miss by one cycle. Nonetheless, the average access time of in-cache is within 5% of any translation buffer.

When computing effective cycle times for TLBs, we have assumed that a full-block transfer from memory is required to satisfy a TLB miss. This is not strictly necessary: a single word transfer would be sufficient. Thus, our results are somewhat optimistic when compared to systems that make this optimization. On the other hand, this is partially balanced by the assumption that TLB misses are handled as quickly as cache misses. Despite these limitations, we believe the results are accurate enough to make valid comparisons.

To further increase performance, some machines [Gust82] employ a TLB in *addition* to caching page tables. While this reduces the average time to service a TLB miss, the caching of PTEs increases the cache miss rate, due to collisions between PTEs and data. However, in-cache translation achieves an effective access time within 5% of the ideal for all traces (ideal assumes no cost for translation). It is hard to justify the significant additional cost of a TLB to achieve *at most* 5%.

Summary of Commercial TLB Performance								
Machine	TLB Size (entries)	Set Size (entries)	Page Size (bytes)	Miss Rate (Percent)				
				LISZT	VAXIMA	CS100K	CS20K	MVS
VAX-11/730	128	1	512	3.588	4.070	4.332	4.444	3.948
VAX-11/780	128	2	512	1.782	2.306	2.344	2.460	2.664
VAX 8600	512	1	512	0.639	1.249	1.545	1.710	1.240
VAX-11/750	512	2	512	0.324	0.619	0.676	0.856	0.548
IBM 370 3033	128	2	4096	0.097	0.305	0.450	0.550	**0.145**
Amdahl 470V/6	256	2	4096	**0.023**	0.112	0.174	0.223	**0.063**
Amdahl 470V/8	512	2	4096	**0.014**	0.047	0.086	0.101	**0.030**
SPUR In-Cache	n/a	n/a	4K	0.026	0.030	0.046	0.050	0.304

Table 4 : Commercial TLB Performance

Simulations of the VAX TLBs are for one half only (only half the buffer is available to user programs while the other half is reserved for system-space translations). The IBM and Amdahl performance were simulated using a hashed index based on an Exclusive OR of the address bits. By allowing a large, variable number of cache entries to hold PTEs, the in-cache method achieves lower miss rates than almost all these buffers (exceptions in bold font). The figures for in-cache translation are from Table 2.

Commercial TLB Performance with SPUR Virtual Cache								
	VAXIMA				MVS			
Separate TLB	Cache Miss (%)	TLB Miss (%)	Avg. Time (cycles)	Relative Perf. (/SPUR)	Cache Miss (%)	TLB Miss (%)	Avg. Time (cycles)	Relative Perf. (/SPUR)
VAX-11/730	1.844	0.687	1.329	1.051	1.677	0.680	1.306	1.023
VAX-11/780	1.844	0.588	1.316	1.041	1.677	0.531	1.287	1.008
VAX 8600	1.844	0.559	1.312	1.038	1.677	0.619	1.299	1.017
VAX-11/750	1.844	0.415	1.294	1.023	1.677	0.341	**1.262**	**0.988**
IBM 370 3033	1.844	0.157	**1.260**	**0.997**	1.677	**0.092**	**1.230**	**0.963**
Amdahl 470V/6	1.844	0.080	**1.250**	**0.989**	1.677	**0.048**	**1.224**	**0.958**
Amdahl 470V/8	1.844	0.047	**1.246**	**0.986**	1.677	**0.030**	**1.222**	**0.957**
SPUR In-Cache	1.844	0.030	1.262	1.000	1.677	0.304	1.277	1.000

Table 5 : Performance of Commercial TLBs After Cache Miss

These are the same commercial translation buffers as in Table 4. Here, however, they are placed after the SPUR virtually-tagged cache to show performance when translating only on cache misses. As before, only half the entries for the VAX buffers were simulated, the IBM and Amdahl TLBs use a hashed index, and the page size is 512 bytes for VAX buffers and 4K bytes for the others. The TLB miss rate and average access time provide different information because the cost of a miss differs between a TLB and in-cache translation. A TLB may be accessed in parallel with the cache reference, while the PTE lookup for in-cache is strictly sequential.

4.4. Additional Analyses

To simplify the hardware, the SPUR workstation traps to software to execute infrequent operations, such as setting miss (reference) bits and dirty bits. A miss bit trap occurs at most once each time the miss bit is cleared. With large main memories (each SPUR workstation will have 20-40 megabytes), paging should occur infrequently. Since the miss bits need only be reset during memory starvation (to determine likely pages to swap out), the overhead of these traps should be small.

Dirty bit traps occur each time a page is written for the first time; all translation schemes incur similar overhead, since they must update the PTE dirty bit when the page is modified. Dirty bit misses, which occur on accesses to blocks that are brought into the cache before the page is first modified, are unique to in-cache translation and present a potential performance loss. The simulator was extended to estimate the frequency of dirty bit misses; the results of this study are presented in Table 6. Despite the cold-start effect, only 48 extra misses occurred in one million references, in the worst case. That is less than one dirty bit miss per 20000 references. For a longer trace the frequency decreases, since the extra misses due to cold-start are amortized over more references. These results indicate that pages are modified quickly; thus, dirty bit misses are not a performance problem.

One reason to have a physical address cache and TLB is to reduce the size of the cache tag memory. This reduction occurs because virtual addresses are typically larger than physical addresses, e.g., in SPUR, the virtual address is 38 bits and the physical address is 32 bits. In addition, SPUR caches both virtual and physical tags, which greatly increases the necessary bits. If we were to use the additional bits to build a TLB, it would have over 1600 entries (4K blocks * 21 bit virtual tag = 86016 bits, TLB entry = virtual tag + PTE = 21 + 32 = 53 bits, 86016 / 53 = 1622 TLB entries). If only the virtual tag were cached, then the possible TLB size drops to under 500 entries. This is still a large TLB. However, the absolute number of bits is not necessarily a meaningful metric: the number of IC packages is often much more important. For example, in the SPUR prototype the virtual tags occupy only six 4K x 4 bit static RAM chips, and require only 3 more glue chips than physical tags. It would be difficult, if not impossible, to build a TLB with so few chips without using custom logic. Thus, in-cache translation is not necessarily less space efficient than TLBs.

5. Status and Conclusions

As cache sizes increase, virtually-tagged caches provide high performance with less hardware complexity than physical caches. Since translation is performed only on cache misses, it can be slower than for physical caches without significant impact on performance. In this paper we describe a new translation mechanism that eliminates the traditional TLB, using the virtually-tagged cache to hold PTEs.

The simulation results show that a large cache is very effective in caching page table entries. The effective cache access time is comparable to large set-associative buffers, and is, in fact, within 5% of optimal. With this result, we have shown that a high performance memory system can be designed without a TLB. Eliminating the TLB reduces the hardware cost and complexity of a uniprocessor design. More importantly, the in-cache mechanism provides translation coherency in shared memory multiprocessors, without additional hardware complexity or software restrictions.

Dirty Bit Performance for In-Cache		
Trace	Dirty Bit Traps	Dirty Bit Misses
LISZT	48	2
VAXIMA	132	33
CS100K	164	25
MVS	147	48

Table 6 : In-Cache Dirty Bit Performance

This table shows the number of dirty bit traps and dirty bit misses generated by the traces used in the previous simulations. All translation schemes must take the necessary traps, i.e., the PTE dirty bit must be set when the page is first modified.

The in-cache translation algorithm is being implemented as part of the SPUR multiprocessor workstation. The cache controller, which executes much of the algorithm, is implemented as a custom CMOS chip, in a 2μm n-well process. The chip is targeted for fabrication in the summer of 1986. The cache data and tags are stored in vendor supplied static RAMs: the data portion uses 45ns 16K x 4 bit parts, and the tag portion uses 25ns 4K x 4 bit parts.

6. Acknowledgements

We wish to acknowledge John Ousterhout for his many suggestions that helped lead to this final design. Jean-Loup Baer, Russ Brown, David Culler, Jane Doughty, Hugh Lauer, Corinna Lee, Lishing Liu, Richard Sites, and Alan Smith read earlier drafts of this paper and provided excellent comments. Major funding for this research was from DARPA under contract order 482427-25840 by NAVALEX. Additional support was provided by Texas Instruments and the California MICRO program.

7. References

[CDC84] CDC,. *Hardware Reference Manual No. 60462090, CDC Cyber 180 Computer System Model 990, Virtual State.* Control Data Corporation, St. Paul, Minnesota, 1984.

[Cens78] Censier, L. M. and P. Feautrier. "A New Solution to Coherence Problems in Multicache Systems." *IEEE Transactions on Computers* **27**, 12 (December 1978), 1112-1118.

[Clar85] Clark, D. W. and J. S. Emer. "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement." *ACM Transactions on Computer Systems* **3**, 1 (February, 1985).

[Denn70] Denning, P. J. "Virtual Memory." *Computing Surveys* **2**, 3 (September, 1970).

[Digi81] Digital Equipment Corporation,. *VAX Architecture Handbook.* Maynard, Massachusetts 01754, 1981.

[Gust82] Gustafson, R. N and F. J. Shapiro. "IBM 3081 Processor Unit: Design Considerations and Design Process." *IBM J. of Research and Development* **26**, 1 (January, 1982), 12-21.

[Henr84] Henry, R. R. *Address and Instruction Tracing for the VAX Architecture.* Unpublished Report, U.C. Berkeley, November, 1984.

[Hill83] Hill, M. D. *Evaluation of On-Chip Cache Memories.* Master's Report, Computer Science Division, EECS Dept., U.C. Berkeley, December 1983.

[Hill85] Hill, M. D. *et al. SPUR: A VLSI Multiprocessor Workstation.* Submitted for publication in *Computer*, November 1985.

[Katz85] Katz, R. H., S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. "Implementing a Cache Consistency Protocol." *Proc. 12th International Symposium on Computer Architecture*, Boston, Mass., June 1985, pages 276-283.

[Knap85] Knapp, V. *Virtually Addressed Caches for Multiprogramming and Multiprocessing Environments.* U. of Washington, Dept. of Computer Science, Technical Report No. 85-06-02, June, 1985.

[McCr84] McCreight, E. M. "The Dragon Computer System: An Early Overview." *NATO Advanced Study Institute on Mircoarchitecture of VLSI Computers*, Urbino, Italy, July, 1984.

[Patt85] Patterson, D. A. "Reduced Instruction Set Computers." *Communications of the ACM* **28**, 1 (January, 1985), 8-21.

[Ritc85] Ritchie, S. A. *TLB For Free: In-Cache Address Translation For A Multiprocessor Workstation.* UC Berkeley, Computer Science Division, Technical Report No. UCB/CSD 85/233, May 1985.

[Saty81] Satyanarayanan, M. and D. Bhandarkar. "Design Trade-offs in VAX-11 Translation Buffer Organization." *IEEE Computer* **14**, 12 (Dec.1981), 103-11.

[Smit82] Smith, A. J. "Cache Memories." *Computing Surveys* **14**, 3 (Sept. 1982), 473-530.

[Smit85] Smith, A. J. "Cache Evaluation and the Impact of Workload Choice." *Proc. 12th International Symposium on Computer Architecture*, Boston, Mass., June 1985, pages 64-73.

[Stan85] Stanford University,. "MIPS-X: A High Performance Computer." *Computer Systems Laboratory Technical Progress Report*, March, 1985, pages 7-12.

[Tang76] Tang, C. K. "Cache System Design in the Tightly Coupled Mutliprocessor System." *Proceedings of NCC*, 1976, pages 749-753.

[Unga84] Ungar, D., R. Blau, P. Foley, D. Samples, and D. Patterson. "Architecture of SOAR: Smalltalk on a RISC." *Proc. Eleventh International Symposium on Computer Architecture*, June 1984, pages 188-197.