Supporting Reference and Dirty Bits in SPUR's Virtual Address Cache

David A. Wood
Randy H. Katz

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

ABSTRACT

Virtual address caches can provide faster access times than physical address caches, because translation is only required on cache misses. However, because we don't check the translation information on each cache access, maintaining reference and dirty bits is more difficult. In this paper we examine the trade-offs in supporting reference and dirty bits in a virtual address cache. We use measurements from a uniprocessor SPUR prototype to evaluate different alternatives. The prototype's built-in performance counters make it easy to determine the frequency of important events and to calculate performance metrics.

Our results indicate that dirty bits can be efficiently emulated with protection, and thus require no special hardware support. Although this can lead to *excess faults* when previously cached blocks are written, these account for only 19% of the total faults, on average. For reference bits, a *miss bit approximation*, which checks the references bits only on cache misses, leads to more page faults at smaller memory sizes. However, the additional overhead required to maintain true reference bits far exceeds the benefits of a lower fault rate.

1. Introduction

Virtual address caches generally provide faster access times than physical address caches because they eliminate address translation from the critical path. Given this fundamental performance advantage, we might expect to find virtual address caches in most high-performance systems. Unfortunately, several problems have kept virtual address caches from wide spread use. The most serious of these, virtual address *synonyms*, has received considerable attention [Knap85, Smit82]. The simplest solutions restrict the virtual address mapping and require only minimal hardware support. In SPUR, the operating system prevents synonyms by restricting processes that share memory to use the same global virtual address [Hill86]. The hardware supports a simple segment mapping from the process virtual space to the global virtual space. The Sun-3 architecture prevents synonyms by restricting the cache to be direct-mapped, and restricting virtual address synonyms (aliases) to be equal modulo the cache size [Sun85].

While these solutions make large virtual address caches more attractive, maintaining reference and dirty bits is difficult. Most systems use them as hints to help optimize page replacement policies. A page's reference bit is set on the first reference to the page, and its dirty bit is set on the first write to the page. Systems with physical address caches usually use a translation lookaside buffer (TLB) to translate virtual addresses to physical addresses. The TLB provides a convenient place to cache the reference and dirty bits, along with the translation information. Since the TLB must be accessed on each reference, checking the bits incurs no additional overhead.

Systems with virtual address caches generally do not have this luxury. Since the cache is accessed with virtual addresses, the TLB is only accessed on cache misses. Thus these systems require additional hardware support to maintain reference and dirty bits. For example, the Sun-3 hardware checks the dirty bit in the memory management unit (in place of a TLB) on the first write to a cache block.

There are several different approaches to maintaining reference and dirty bits in systems with virtual address caches, requiring different levels of hardware and software support. In this paper we examine the most promising schemes and use measurements from the SPUR prototype to evaluate them.

In the remainder of this section, we briefly summarize the relevant background of SPUR and introduce some terminology. In Section 2, we describe the methodology used to evaluate the design alternatives. In Section 3, we examine and evaluate different implementations of dirty bits. In Section 4, we discuss reference bits, and evaluate the alternatives. Finally, in Section 5 we summarize the results and present our conclusions.

1.1. Background

SPUR is a shared-memory multiprocessor workstation developed at U.C. Berkeley [Hill86]. Each workstation can contain up to 12 processor boards (the prototype used in these studies was a uniprocessor system), each with three custom chips: a CPU, a floating point unit (FPU), and a cache controller (CC). A 128 Kilobyte direct-mapped unified cache reduces the load each processor demands of the single shared bus. The cache controller implements the Berkeley Ownership coherency protocol [Katz85] to maintain a consistent image across all the caches.

The cache is accessed with virtual addresses, so cache hits proceed without translation. Cache misses require the virtual address to be translated into a physical address before accessing main memory. The cache controller implements in-cache address translation [Wood86], an algorithm unique to SPUR.

In-cache translation does not use a TLB; instead, page table entries (PTEs) compete with instructions and data for space in the unified cache. When a reference misses in the cache, the controller computes the virtual address of the corresponding PTE, using a simple shift-and-concatenate circuit. Then the controller looks for the PTE in the cache, essentially using it as a very large TLB. If the PTE is found, then translation is complete. If not, the cache controller looks for the second-level PTE, which contains the physical address of the first-level PTE. If the second-level PTE is not in the cache either, the cache controller gets it directly from memory; this is possible because second-level page tables are "wired down" at well known addresses. The SPUR cache controller also implements reference and dirty bits, but we defer this discussion until the later sections.

Throughout the rest of this paper we discuss the interactions of virtual memory with the cache. It is important to distinguish between the 4 Kbyte virtual memory *pages* and the 32 byte cache memory *blocks*. When we discuss dirty bits, we are concerned with the *page dirty bits*, used for virtual memory replacement, not the *block dirty bits*, used for cache replacement.

2. Methodology

We usually use trace-driven simulation to evaluate memory system designs, such as cache memory organizations and paging algorithms. Trace-driven simulation provides precise repeatability using an accurate representation of a real workload. It is more accurate than analytic models, but still allows evaluation of systems which do not exist. Unfortunately, trace-driven simulation is limited by the length of the traces. In this study, we want to look at the interactions of the cache memory and the paging algorithm. However, for a trace to include a significant number of paging events it must contain 100's of millions of references. Traces of this length are well beyond our abilities to obtain, store, and simulate.

Because we could not fully evaluate the alternatives before we built SPUR, we were forced to use back-of-the-envelope calculations and small scale simulations to make our design decisions. However, we included a set of performance counters on the cache controller chip to count important events. These on-chip counters give us the opportunity to re-evaluate our decisions with more complete information.

The SPUR cache controller [Wood87] contains 16 32-bit performance counters. A mode register selects one of 4 possible sets of events to be measured. Events include the number of instruction fetches, processor reads and writes, and the number of times each type of reference misses in the cache. The counters also measure the performance of the SPUR in-cache translation algorithm and the Berkeley Ownership coherency protocol.

The measurements reported in this paper were taken on a prototype SPUR system. Because of noise problems on this system's processor board, it runs at 1.5 times the design cycle time. In addition, the processor chip's instruction buffer was disabled, further slowing the performance by nearly a factor of 3. These factors combine to make the system run at approximately 1.5 MIPS. However, since the relative speed of the processor and I/O is a second order effect to our measurements, we believe our results scale to faster processors.

The Sprite operating system [Oust88] runs on the SPUR hardware. Sprite is UNIX compatible at the system call level, but is completely rewritten with an emphasis on network services and multiprocessors. Because Sprite was developed at Berkeley, as part of the larger SPUR project, it was easy to modify the kernel as described later.

Table 2.1: SPUR System Configuration

Processor Information	
Cache Size	128 Kbytes
Associativity	Direct Mapped
Block Size	32 bytes
Page Size	4 Kbytes
Instruction Buffer	Disabled
Processor cycle time	150ns
Backplane cycle time	125ns
Memory Information	
Time to first word	3 cycles
Time to next word	1 cycle

To evaluate the different alternatives, we needed synthetic workloads that could be repeated with different paging policies and memory sizes. We designed a workload (called *WORKLOAD1*) to reflect a moderately heavy load for a CAD tool developer. This script includes the compilation of several modules plus the link and debug of a 12000 line CAD tool (espresso). The same CAD tool runs in the background optimizing a large PLA. Other edit, compile, and miscellaneous commands manipulate files and directories. In addition, two performance monitor programs periodically report status of the virtual memory system and CPU performance[1]. For a second workload (called *SLC*), we used the SPUR Common Lisp [Zorn87] system and the SPUR lisp compiler compiling a set of benchmark programs. These two workloads are representative of the types of applications originally intended to run on SPUR.

3. Dirty Bit Alternatives

3.1. Dirty Bit Implementation Trade-offs

In this section, we examine the performance of several dirty bit implementations. Since maintaining dirty bits is a small component of total system performance, we are not interested in finding the optimal approach. Rather, our goal is to determine the simplest implementation that yields acceptable performance.

One of the lessons we have learned from RISC processor designs [Patt85] is to implement frequent cases in hardware and trap to software for the infrequent ones. We can apply this lesson to dirty bits. A page's dirty bit must be checked frequently, perhaps as often as every processor write, but only needs to be set infrequently, on the first write to each page. Thus while we must dedicate some hardware to check the bit, we can trap (or, in SPUR nomenclature, *fault*) to software to update the bit. Moving infrequent functions from hardware into software reduces the hardware complexity and may improve performance by reducing the cycle time.

Setting the dirty bit in software is very desirable in shared-memory multiprocessors. Because page table entries (PTEs) are shared between processors, updates must either be atomic or controlled by some higher level synchronization

[1] This workload lacks any window activity, a major deficiency for a workstation environment. Unfortunately, no window system currently runs on SPUR, so it is not possible to include this behavior.

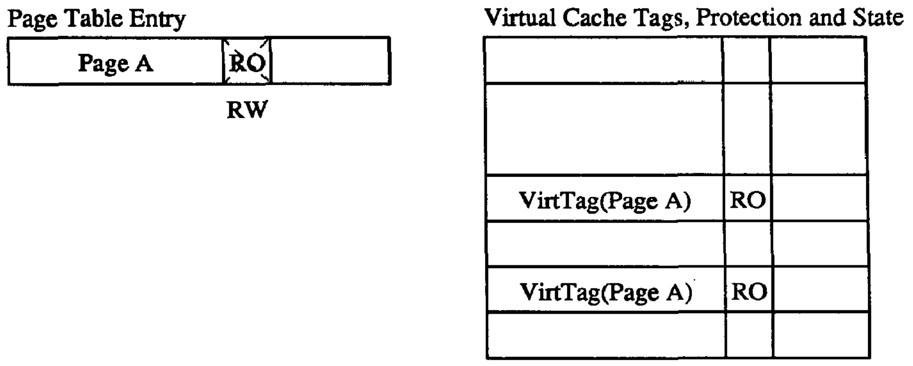


Figure 3.1: Example of Multiple Cache Blocks

In a virtual address cache, the protection is cached along with each block. In this example, two blocks from Page A were brought into the cache while the page protection was read-only. Changing the protection in the page table entry does not directly affect the protection of the two previously cached blocks. If these blocks are left unchanged, subsequent writes will result in protection faults.

mechanism, such as a semaphore. Performing the PTE updates in software can therefore substantially simplify the memory management unit design. Throughout the rest of the paper we assume this hardware/software partitioning.

Since we are going to update the dirty bit using a software fault handler, it is natural to consider combining the dirty bit checking mechanism with the protection checking mechanism. After all, both functions cause a fault to software on the first write to a page. To emulate dirty bits with protection, the operating system initially marks writable pages as read-only, then when the first write causes a fault, it sets a software dirty bit and increases the protection level to read-write. This approach requires no special hardware support, and only minor changes to the operating system.

This approach, which we will refer to as the *FAULT* alternative, is very promising, and is used in several commercial machines, e.g., the MIPS R2000 [DeMo86]. However, there is a drawback to using it in systems with virtual address caches. In a virtual address cache like SPUR's, a copy of the protection information is stored with each cache block. Thus there may be several blocks from a particular page in the cache, each with its own copy of the protection level. Changing the protection in the page table entry does not affect blocks already brought into the cache, as illustrated in Figure 3.1. Thus, even though the first write to a page results in a fault, making the page writable, subsequent writes to other resident cache blocks will also fault. If these *excess faults* occur frequently, they could significantly degrade performance.

One approach to eliminating excess faults is to flush the page from the cache when the first fault occurs. This guarantees that no blocks from the page remain in the cache with the old protection level. This alternative, called *FLUSH*, incurs less overhead if the time to flush the page is less than the expected overhead due to excess faults. On the other hand, if flushing a page from the cache is inefficient, or excess faults are infrequent, then this alternative could be slower.

In the design of SPUR, we took another approach to reducing the performance penalty of previously cached blocks. Rather than emulate the dirty bit with protection, the PTE contains an explicit, hardware-defined dirty bit. And just as we cache the page's protection with each cache block, we also cache the page's dirty bit. Note that this bit is distinct from the *block dirty bit* that indicates that a particular cache block has been modified (this difference is illustrated in Figure 3.2). When a block is brought into the cache, both the protection and dirty bit are copied from the PTE into the cache. Thus if the page has already been modified, the cached page dirty bit will be a one.

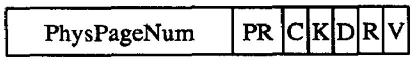
Each time a processor writes to a cache block, the hardware checks the cached copy of the page dirty bit. The first time a page is written, the cached copy indicates that the page is clean. To verify that this is the first write to a page, the hardware checks the PTE. If the PTE dirty bit indicates that the page is still clean, then this is the first write to the page and the hardware generates a "dirty bit fault" to a software handler which sets the bit. If a write finds that the cached copy of the dirty bit indicates the page is clean, but the PTE is marked dirty, then we have already faulted on another cache block and don't need to fault again. Instead, the hardware merely updates the cached copy of the page dirty bit, and the write proceeds normally. All subsequent writes to that block find the cached copy of the page dirty bit set, and proceed without delay. In the SPUR implementation, updating the cached copy of the page dirty bit is implemented by forcing a cache miss; this leads to the name *dirty bit miss*, which we use for the rest of the paper.

It is important to emphasize the similarities and differences between the SPUR scheme and emulating dirty bits with protection (the FAULT alternative). In both approaches, the "dirty bit information" (dirty bit or protection) is cached with each block on a cache miss, and the first write to a page results in a fault to a software routine that actually modifies the PTE. The key difference is that any subsequent writes to other cache blocks (from the same page) that were brought in while the page was still clean cause faults when emulating with protection, but only dirty bit misses in the SPUR scheme. Since a fault takes at least one order of magnitude longer than a dirty bit miss (as discussed in the next section), the SPUR scheme performs significantly better under certain workload conditions.

Note that while SPUR implements an explicit dirty bit, the same idea could be applied directly to the protection. Instead of immediately faulting to software when the cached copy of the protection indicates an access violation, the hardware first checks the PTE. If the cached copy is out of date, the hardware refreshes it (with a "protection bit miss") and permits the access to proceed. Since the performance of this scheme is identical to what we implemented in SPUR, we will not discuss it separately.

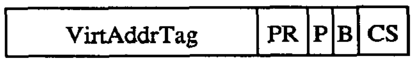
Finally, we also consider a fourth alternative, called WRITE, that is similar to the approach used in the Sun-3 architecture [Sun85]. In this scheme, the hardware checks the PTE dirty bit on the first write to a cache block. There are two cases to consider. First, when a write misses in the cache, the controller must examine the PTE to obtain the physical address, so checking the dirty bit incurs no additional penalty. In the second case, a write hits on a clean cache block. In this case, some additional overhead is needed to access the PTE and check the dirty bit. Unlike the Sun-3, we assume that the hardware generates a fault if the page is clean, and that software performs the update. This assumption makes the comparison unbiased, by using the mechanisms available in SPUR. Since this scheme always checks the PTE before faulting it never generates excess faults.

a) SPUR Page Table Entry Format



PR = Protection (2 bits)
C = Coherency
K = Cacheable
D = Page Dirty Bit
R = Page Referenced Bit
V = Page Valid Bit

b) SPUR Cache Tag Format



PR = Protection (2 bits)
P = Page Dirty Bit
B = Block Dirty Bit
CS = Coherency State (2 Bits)

Figure 3.2: SPUR Page Table and Cache Line Format

This figure illustrates the page table entry format and cache line (block frame) format in the SPUR prototype. Note that the cache line contains two dirty bits: a *block dirty bit*, that indicates that the cache block has been modified while in the cache, and a *page dirty bit*, that indicates that the page has been modified. When a block is brought into the cache, the page dirty bit and protection are copied from the PTE into the cache line. Since the PTE may change while a block is in the cache, the cached copies of the page dirty bit and protection may become inconsistent with respect to the PTE.

3.2. Analysis

The goal of this analysis is to determine which alternative is superior under different workload conditions. Emulating dirty bits with protection makes the least demands upon the hardware, and is therefore the most attractive to implement. But do the excess faults present a performance problem? If so, is flushing the page an efficient alternative? Does the performance gain from SPUR's dirty bit miss mechanism justify the implementation complexity? Our inability to answer these questions to our satisfaction led us to choose a conservative alternative for SPUR (i.e., the dirty bit miss mechanism). Now that the SPUR prototype is operational, we can use the built-in performance counters to evaluate the trade-offs.

We evaluate the different approaches by comparing simple models of their overhead. We use measurements from SPUR to determine the frequency of different events, and, in some cases, measure their duration in the SPUR implementation. We also consider some alternatives to what was actually built in SPUR.

The performance overhead of the four alternatives

$$O(policy) = overhead\ of\ dirty\ bit\ policy$$

can be expressed in terms of the following parameters:

N_{ds}	Number of necessary dirty bit faults.
$N_{ef} = N_{dm}$	Number of previously cached blocks that cause *excess faults* or *dirty bit misses.*
N_{zfod}	Number of zero-filled page faults.
N_{w-hit}	Number of blocks brought into cache by a read that are later modified.
N_{w-miss}	Number of blocks brought into cache by a write miss.
t_{ds}	Time required to handle a dirty bit fault.
t_{dm}	Time required to handle a dirty bit miss.
t_{flush}	Time to flush a page from the cache. Assumes 10% of blocks from the page are in cache and are clean.
t_{dc}	Time to check the dirty bit when writing to a clean block in the cache.

In the first alternative (FAULT), the hardware faults when it is necessary to set the dirty bit (i.e., on the first write to a page) and also on previously cached blocks. In the SPUR implementation, the fault handler must switch to the kernel stack, read a cache controller status register to determine the type of fault, then decode the instruction to determine the address of the page table entry. Because of the high overhead incurred on a fault, roughly 1000 cycles[2], the actual time to update the PTE is a small fraction of the total time.

$$O(FAULT) = (N_{ds} + N_{ef})\, t_{ds}$$

Thus we assume there is no difference in the time to handle necessary and excess faults.

The second alternative (FLUSH) calls for flushing the page from the cache on a fault, preventing the excess faults from occurring.

$$O(FLUSH) = N_{ds}\,(t_{ds} + t_{flush})$$

[2] The current implementation of the fault handler has not been tuned. We believe that it can be improved, but doing so will not affect our conclusions.

SPUR's flush mechanism flushes a single cache block regardless of its virtual address tag. Thus flushing a page from the cache requires 128 flush operations, one for each block. Since the hardware does not check the address tag, blocks from other pages may be unnecessarily flushed, substantially increasing the bus traffic and the total overhead. Assuming one-fifth of the blocks must actually be written back to memory, the flush would cost nearly 2000 cycles.

However, a flush operation that checks the address tag could easily be implemented, and we assume this operation exists for a more generic comparison. Even with this operation, flushing a page from the cache will still take approximately 500 cycles (128 blocks to check, two instructions for loop overhead, 90% of blocks at 1 cycle per block, 10% must be flushed at 10 cycles per block). This is approximately half the overhead of an excess fault, not counting the time to reread blocks that are accessed again. Therefore, FAULT is superior to FLUSH if there are at least twice as many necessary faults as excess faults.

The SPUR alternative greatly reduces the overhead of writes to previously cached blocks. A dirty bit miss takes 25 cycles, on average, compared to 1000 cycles for an excess fault.

$$O(SPUR) = N_{ds}\,(t_{ds} + t_{dm}) + N_{dm}\,t_{dm}$$

If a large fraction of the blocks in a page are read before they are written, then this scheme will greatly reduce the overhead to maintain dirty bits.

The last alternative (WRITE) checks the PTE on the first write to a cache block.

$$O(WRITE) = N_{ds}\,t_{ds} + N_{w-hit}\,t_{dc}$$

We assume that translation is done using SPUR's in-cache translation algorithm, and that the probability the PTE is in the cache is the average across all PTE references. This assumption is pessimistic, but, as we shall see below, does not affect our conclusions. It takes 3 cycles to check the PTE if it is in the cache, plus a weighted miss penalty of about 2 cycles. Thus t_{dc} is approximately 5 cycles.

Finally, we also consider the minimal policy (MIN) for comparison.

$$O(MIN) = N_{ds}\,t_{ds}$$

This alternative includes only the overhead to update the dirty bit in software, and is optimal in the sense that it incurs no additional overhead to check the dirty bit or to handle excess faults.

Table 3.1: Dirty Bit Implementation Alternatives

FAULT	Emulate dirty bits with protection. Writes to previously cached blocks cause excess faults.
FLUSH	Emulate dirty bits with protection. When a fault occurs, flush all blocks in that page from the cache, preventing excess faults.
SPUR	Store a copy of the dirty bit with each cache block. Check the PTE before faulting; if the cached copy is merely out of date, update it with a dirty bit miss.
WRITE	Check the PTE on the first write to each cache block.
MIN	Minimal policy. Includes only overhead intrinsic to all policies.

Table 3.2: Time Parameters

Parameter	Cycle Count	Description
t_{ds}	1000	Time for handler to set dirty bit
t_{flush}	500	Time to flush page from cache
t_{dm}	25	Time to update cached dirty bit
t_{dc}	5	Time to check PTE dirty bit

Table 3.1 summarizes the implementation alternatives. Table 3.2 summarizes the time parameters discussed above; we assume the faster flush operation for a more balanced comparison. Table 3.3 summarizes the event frequencies, measured on the SPUR prototype. It is immediately clear from this table that *excess faults occur very infrequently*. At 6 and 8 megabytes of main memory, both workloads cause less than 8% as many excess faults as necessary faults. At 5 megabytes, the fraction climbs to 16% for WORKLOAD1 and 10% for SLC. However, in all cases excess faults are infrequent, suggesting that pages that will be modified are modified quickly. The ratio of N_{w-hit} to N_{w-miss} supports this hypothesis: roughly one-fifth (from 16% to 24%) of the modified cache blocks are read before they are written. Based on this ratio, a simple probability model[3] predicts less than 20% as many excess faults as modified faults. The model provides some intuition why the number of excess faults is low.

However, further investigation uncovered another reason for the low percentage of excess faults. Like UNIX, the Sprite operating system initializes newly allocated stack and heap pages to zero. The kernel maps the initialized page into the process's address space with the dirty bit turned off. But since programs rarely want to read stack and heap pages before they are written (i.e., read a zero), the first operation to these pages is almost always a write, resulting in a dirty bit fault. If we exclude these zero-fill pages from from the necessary dirty bit faults, the fraction of excess faults increases, ranging from 15% to 34%. While still a small percentage, this range is closer to our model's prediction.

Table 3.4 summarizes the overhead for the implementation alternatives. Because they are not intrinsic, we exclude the zero-fill pages from the calculations[4] (i.e., the difference $N_{ds} - N_{zfod}$ is substituted for N_{ds} in the models). Because excess faults occur infrequently, we expect the FAULT policy to perform well. Flushing the page from the cache, the FLUSH policy, increases the overhead by 26% over FAULT on average. Only for WORKLOAD1 at 5 megabytes does FLUSH start to come close, dropping to only 12% worse. The SPUR scheme has the best performance, requiring only 3% more than the minimum (MIN). But since excess faults are rare, SPUR's overhead is only 16% less than FAULT's; this difference amounts to only a small fraction of total system performance. In addition, the SPUR scheme requires one additional state bit

[3] The model assumes a) a uniform distribution of read and write misses, b) infinitely large pages, and c) that necessary faults occur only on write misses. With these assumptions, the number of excess faults has a geometric distribution with parameter $p_w = \frac{N_{w-miss}}{N_{w-hit} + N_{w-miss}}$. Relaxing assumptions b) and c) only reduces the expected number of excess faults.

[4] Note that Sprite will always write a zero-filled page to swap the first time it is replaced, even if the program has not modified it.

Table 3.3: Event Frequencies

Workload	Memory Size (megabytes)	N_{ds}	N_{zfod}	$N_{ef} = N_{dm}$	N_{w-hit} (million)	N_{w-miss} (million)	$t_{elapsed}$ (seconds)
SLC	5	2349	905	237	1.27	7.38	948
	6	1838	905	143	0.839	5.11	502
	8	1661	905	120	0.612	3.68	341
WORKLOAD1	5	9860	5286	1534	6.15	34.0	3016
	6	7843	5181	456	4.92	20.4	2535
	8	7471	5182	364	4.10	17.3	2555

Table 3.4: Overhead of Dirty Bit Alternatives (Excluding Zero-Fills)

Workload	Memory Size (megabytes)	MIN	FAULT	FLUSH	SPUR	WRITE
		millions of cycles (relative to MIN)				
SLC	5	1.44 (1.00)	1.68 (1.16)	2.17 (1.50)	1.49 (1.03)	7.81 (5.41)
	6	0.933 (1.00)	1.08 (1.15)	1.40 (1.50)	0.960 (1.03)	5.13 (5.50)
	8	0.756 (1.00)	0.876 (1.16)	1.13 (1.50)	0.778 (1.03)	3.82 (5.05)
WORKLOAD1	5	4.57 (1.00)	6.11 (1.34)	6.86 (1.50)	4.73 (1.03)	35.3 (7.72)
	6	2.66 (1.00)	3.12 (1.17)	3.99 (1.50)	2.74 (1.03)	27.3 (10.2)
	8	2.29 (1.00)	2.65 (1.16)	3.43 (1.50)	2.36 (1.03)	22.8 (9.95)

per cache block[5], plus an additional 14 product terms in the controller's main PLA (193 vs 207, or 7%). We believe the minor performance improvement does not justify the increase in chip and board complexity.

Not surprisingly, the WRITE policy performed worst of all by a large margin. Roughly one fifth of all cache blocks are read before they are written. Therefore, even though checking the PTE is relatively fast, the frequency it must be done makes the total overhead higher than the other schemes. Even if the time to check the PTE dirty bit is reduced to only 1 cycle, this alternative still has the worst performance. Since this policy has higher overhead despite special hardware support, it is clearly inferior to the FAULT policy.

To summarize the results of this section, it is clear that dirty bits can be efficiently emulated using protection even when using a large virtual address cache. Based on our synthetic benchmarks and measured events on the SPUR hardware, the frequency of excess faults ranges from 16% to 34% the frequency of necessary faults. Additional hardware support can improve dirty bit performance by at most 34%, which amounts to much less than 1% of overall system performance. Simply tuning the fault handler would probably achieve a larger improvement. This is good news to hardware designers because it further simplifies the logic they must support. Had these results been available during the design of SPUR, we could have eliminated the dirty bit support, reducing the number of product terms in the sequencer PLA by 7%.

[5] The generalized version, using the protection field, eliminates the need for an extra bit.

3.3. Benefits of Dirty Bits

In the first half of this section, we looked at the cost and performance impact of different implementations of dirty bits. In the remainder of the section, we take a step back and look at what we gain from implementing dirty bits. Although we have shown that dirty bits require no special hardware support, they do add some complexity to the operating system.

Dirty bits improve system performance by eliminating unnecessary page-outs; clean pages need not be written back to secondary storage when displaced from main memory. Dirty bits only help for pages which can be modified; since most systems disallow direct updates of code, dirty bits provide no information for code pages. During times of heavy paging, pages do not stay in memory long and thus are unlikely to be modified. Under these conditions dirty bits can greatly reduce the page-out traffic. However, memory prices have dropped by a factor of 100 over the last 10 years [Myer86] prompting a rapid increase in main memory sizes[6]. Workstations are now commonly sold with at least 8 megabytes of memory. With the relatively low price of memory, most users will increase their memory size rather than sustain consistently high paging rates. Many computing environments also provide compute servers in addition to workstations; jobs which page heavily on a workstation are usually moved to these larger machines.

In conjunction with the increase in memory size, many workstations have gone to large page sizes: the Sun-3's pages are 8 kbytes, the MIPS R2000's and SPUR's are 4K bytes.

[6] We believe the recent DRAM shortage is only a short-term aberration resulting from protectionist trade policies.

Table 3.5: Page-Out Results from Sprite Development Systems

Hostname	Memory Size	Uptime (hours)	Number of Page-Ins	Potentially Modified Pages	Not Modified Pages	Percent (%) Not Modified	Percent (%) Additional Paging I/O
mace	8 MB	70	15203	2681	488	18%	2.8%
sloth	8 MB	37	10566	2146	129	6%	1.0%
mace	8 MB	46	48722	5198	814	16%	1.4%
sage	12MB	45	5246	544	14	3%	0.2%
fenugreek	12MB	36	8556	1154	58	5%	0.6%
murder	16 MB	119	23302	12944	895	7%	2.5%

Both factors, large memories and large pages, suggest that most modifiable pages will be modified while in memory. Thus dirty bits may be of marginal utility, and perhaps could be excluded from future operating systems. To examine this hypothesis, we looked at the page-out performance of our Sun-3's running Sprite. The Sprite developers use these systems to enhance and maintain the Sprite operating system, as well as other tasks such as reading mail, and writing papers and dissertations. The workload on these machines is similar to many other software development environments.

All the systems have at least 8 megabytes of memory, so the paging rates are relatively low. There is also a certain amount of self-scheduling; users tend to run programs with very large memory demands on the systems with more physical memory.

Table 3.5 displays the measurements from the development machines. The second to last column holds the main result: with 8 megabytes of memory at least 80% of all modifiable pages are modified. With 12 megabytes or more, the fraction is at least 90%. Also, as shown in the last column, the total number of additional pages that would be written out without dirty bits only increases the total number of paging I/Os by at most 3%. Since the paging rate is already low, a 3% increase will have a negligible impact on the total system performance.

These results support our hypothesis, indicating that dirty bits provide only negligible performance improvement for this class of workload. While these results may not apply to all applications, it is clear that for an important class of workloads dirty bits provide little benefit, which will decline further with increasing memory size.

4. Reference Bits

4.1. Reference Bit Policies

In this section, we examine the trade-offs in reference bit implementations. Reference bits are used to maintain a pseudo-LRU ordering of resident pages. A *page daemon* periodically clears the reference bits and reclaims unreferenced pages.

As with dirty bits, the approach generally taken in systems with TLBs is not directly applicable to systems with virtual address caches. In traditional implementations, the reference bits are cached in the TLB and checked on each processor reference. However, in a system like SPUR, it is impractical to check the bits this frequently. Instead, SPUR only checks the reference bit on cache misses; since the hardware must access the PTE on a cache miss, there is no additional penalty to check the reference bit. As with dirty bits, SPUR generates a fault to a software handler when the bit must be set. While SPUR supports an explicit reference bit, we could just as easily emulate them with the valid bit, as done in BSD Unix [Baba81].

Checking the reference bit on cache misses reduces the overhead, but it does not provide exactly the same functionality. When the page daemon, or allocate procedure, clears the

Table 4.1: Reference Bit Results

Workload	Memory Size (megabytes)	Policy	Page-Ins		Elapsed Time (seconds)	
SLC	5	MISS	4647	(100%)	948	(100%)
		REF	4738	(102%)	1020	(108%)
		NOREF	8230	(177%)	1341	(141%)
	6	MISS	1833	(100%)	502	(100%)
		REF	1866	(102%)	534	(106%)
		NOREF	3465	(189%)	703	(140%)
	8	MISS	1056	(100%)	341	(100%)
		REF	1062	(101%)	342	(101%)
		NOREF	1512	(143%)	382	(112%)
WORKLOAD1	5	MISS	11959	(100%)	3016	(100%)
		REF	11119	(93%)	3153	(105%)
		NOREF	16045	(134%)	3214	(107%)
	6	MISS	3556	(100%)	2535	(100%)
		REF	3617	(102%)	2677	(106%)
		NOREF	5073	(143%)	2555	(101%)
	8	MISS	1837	(100%)	2555	(100%)
		REF	1790	(97%)	2701	(106%)
		NOREF	1926	(105%)	2505	(98%)

reference bit, it does not affect blocks from that page that are already in the cache. Thus the processor can continue to reference those blocks without setting the reference bit. Under this policy, which we call the *MISS bit approximation*, or simply the *MISS* policy, the page daemon may incorrectly replace pages that have actually been recently referenced, but have not recently caused a cache miss.

We can eliminate these incorrect replacements if the page daemon flushes the page from the cache when it clears its reference bit. This guarantees that the next reference to the page will cause a cache miss, setting the reference bit. Under this policy, called *true reference bits* or simply the *REF* policy, the reference bits are accurately maintained and should result in fewer page faults than the MISS policy. However, the REF policy does not come for free. Flushing a page from the cache is an expensive operation, relative to the cost of clearing the reference bit. This is especially true in a multiprocessor, which must flush the page from all the caches. Not only does the flush take a long time, but it disrupts the cache, forcing additional cache misses to refetch some of the blocks.

For small caches, the MISS policy is probably a good approximation to true reference bits. When the cache miss rate is high then the average residency of a block is short, and the reference bit for active pages will be set fairly soon after it is cleared. But as caches increase in size, we expect the approximation to become worse. Consider a cache of infinite capacity. Once a block is brought into the cache it never leaves without being explicitly flushed. Thus the MISS policy never sets the reference bit once the entire page is resident in the cache. At this extreme, the MISS bit approximation provides no benefit; eliminating reference bits all together, the *NOREF* policy, would be superior since it eliminates the overhead of checking, setting and clearing them.

The NOREF policy should also be superior with large memory sizes. It has been observed that large systems spend lots of time searching for unreferenced pages [McKu85]. With large enough main memories, the overhead to maintain pseudo-LRU ordering may exceed the overhead of additional faults incurred by not maintaining reference bits.

We consider a very simple NOREF policy, primarily to minimize changes to the Sprite virtual memory system [Nels86]. Under this policy, the basic replacement algorithm is unchanged, however the machine dependent routine that reads the hardware reference bit always returns false. Conversely, the routine that clears the hardware reference bit has no effect, leaving the hardware bit always set (thus preventing reference faults). While we consider this policy to be reasonable, we believe there may be better replacement algorithms that do not support reference bits. Nonetheless, if we get acceptable results under this policy, then it supports the proposition that we can eliminate reference bits.

4.2. Reference Bit Evaluation

To evaluate these three policies, we modified Sprite to use each of the them, under control of run-time flags. We ran our synthetic workloads on the SPUR prototype, with 5, 6, and 8 megabytes of memory. We ran five repetitions of each data point, using a randomized experiment design to minimize bias. The main results are summarized in Table 4.1.

Running WORKLOAD1 at 8 megabytes of main memory generates only a small amount of paging activity, so the overhead of maintaining reference information exceeds the benefits. The REF policy requires 3% fewer page-ins than the MISS policy, but takes 6% longer to execute due to the flush overhead. The NOREF policy generates 5% more page-ins than MISS, but because it spends no time maintaining reference bits runs 2% faster.

As we would expect, the reference bits provide more benefit as memory becomes more scarce. At both 5 and 6 megabytes, the NOREF policy generates significantly more page-in traffic, 134% and 143% respectively. Note, however, that at 6 megabytes, the performance degradation is still only 1%. Finally, at 5 megabytes paging becomes heavy and the REF policy results in 7% fewer page-ins than MISS. Nonetheless, MISS still requires 5% less time to execute because of its lower overhead.

The SLC workload displays much more uniform behavior. The MISS policy always results in the fewest page-ins and the shortest elapsed time. For this workload, the NOREF policy never comes closer than 12% slower; for the two smaller memory sizes, it is 40% slower. The REF policy has comparable performance to MISS at 8 megabytes, when there is little paging. But at smaller memory sizes, despite selective cache flushing on reference bit clears, it still generates more page-ins than MISS; consequently, REF has a longer execution time than MISS.

In summary, implementing true reference bits may reduce the number of page-ins at low memory sizes, but the cpu overhead required always exceeds the benefit of the lower fault rate. The MISS bit approximation has the best overall performance, generating significantly fewer page-ins than the NOREF policy but without the high overhead of true reference bits. For WORKLOAD1, however, eliminating reference bits altogether (NOREF) has the best performance at 8 megabytes, and only slightly worse performance at 6 megabytes. Clearly these results do not apply to all systems, but certainly for this workload reference bits provide at best a small increase in performance. As memory sizes increase, this benefit will tend to decrease and may eventually become a hindrance.

5. Summary and Conclusions

In this paper, we have examined alternative implementations of reference and dirty bits for virtual address caches. Virtual address caches provide faster access times than physical address caches, but by eliminating the TLB they make maintaining reference and dirty bits more difficult.

We have shown that simpler is better; dirty bits may be efficiently emulated with protection. The excess faults that occur when multiple blocks from a clean page are brought into the cache and then modified account for only 19% of all dirty bit faults, on average. Thus hardware alternatives like SPUR's dirty bit miss mechanism and the Sun-3's first-write mechanism are not justified. No special hardware is necessary to efficiently support dirty bits.

Approximating reference bits by checking only on cache misses can result in more page faults at smaller memory sizes. However, the overhead of maintaining true reference bits, by flushing a page when clearing the reference bit, far exceeds the benefit of a lower fault rate.

We also examined the possibility of eliminating reference and dirty bits entirely. In measurements on machines used for software development, more than 80% of all writable pages are dirty when replaced; eliminating dirty bits would have increased the total paging activity by at most 3%. Similarly, for some workloads maintaining reference bits, even with the miss bit approximation, may become a liability at larger memory sizes. At 8 megabytes, not a large memory for today's workstations, the overhead to maintain reference bits exceeds the benefits for one of the two workloads presented. These results strongly suggest that the benefits of reference and dirty bits decline as memory size increases, and may eventually degrade rather than improve performance. We are conducting further studies to evaluate the performance of larger applications and larger memory sizes.

6. Acknowledgements

We are very grateful to Mendel Rosenblum who sacrificed progress on his own research to debug the Sprite virtual memory system; without his generous efforts the SPUR measurements would have been impossible. Mike Nelson also provided earlier assistance with Sprite. Alan Smith suggested the probabilistic model in Section 3. Jane Doughty, Garth Gibson and Doug Johnson provided useful comments on earlier drafts of this paper. This work was supported by SPUR/DARPA contract No. N00039-85-C-0269, NSF MIP-8352227, the California MICRO program (together with Texas Instruments, National Semiconductor, Xerox, Honeywell, and Philips/Signetics), and the IBM Predoctoral Fellowship program.

7. References

[Baba81] Babaoglu, O., "Virtual Storage Management in the Absence of Reference Bits", U.C. Berkeley, Electronic Research Laboratory, Memo. No. UCB/Electronics Research Lab. M81/92, November 1981. Ph.D. Dissertation.

[DeMo86] DeMoney, M., J. Moore and J. Mashey, "Operating System Support on a RISC", *Proceedings 1986 IEEE Compcon*, pp. 138-143, March 1986.

[Hill86] Hill, M. D., S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. Ousterhout and D. A. Patterson, "Design Decisions in SPUR", *IEEE Computer*, Vol. 19, No. 11, November 1986.

[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. , pp. 276-283, June 1985

[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.

[McKu85] McKusick, M. K., M. Karels and S. Leffler, "Performance Improvements and Functional Enhancements in 4.3BSD", U.C. Berkeley Computer Science Division Technical Report No. UCB/Computer Science Dpt. 85/245, June 1985.

[Myer86] Myers, G. J., A. Y. C. Yu and D. L. House, "Microprocessor Technology Trends", *Proceedings of the IEEE*, Vol. 74, No. 12 , December 1986.

[Nels86] Nelson, M., "Virtual Memory for the Sprite Operating System", UC Berkeley, Computer Science Division, Technical Report No. UCB/Computer Science Dpt. 86/301, June 1986. M.S. Thesis.

[Oust88] Ousterhout, J. K., A. R. Cherenson, F. Douglis, M. N. Nelson and B. B. Welch, "The Sprite Network Operating System", *IEEE Computer*, Vol. 21, No. 2 , February 1988, pp. 23-36.

[Patt85] Patterson, D. A., "Reduced Instruction Set Computers", *Communications of the ACM*, Vol. 28, No. 1 , January, 1985, pp. 8-21.

[Smit82] Smith, A. J., "Cache Memories", *Computing Surveys*, Vol. 14, No. 3 , Sept. 1982, pp. 473-530.

[Sun85] Sun Microsystems, Inc, Sun-3 Architecture Manual (July 1985).

[Wood86] Wood, D. A., S. J. Eggers, G. A. Gibson, M. D. Hill, J. M. Pendelton, S. A. Ritchie, G. S. Taylor, R. H. Katz and D. A. Patterson, "An In-Cache Address Translation Mechanism", *Proc. Thirteenth International Symposium on Computer Architecture*, Tokyo, Japan , pp. 358-365, June 1986.

[Wood87] Wood, D. A., S. Eggers and G. Gibson, "SPUR Memory System Architecture", Technical Report UCB/Computer Science Dpt. 87/394, University of California, Berkeley , December 1987.

[Zorn87] Zorn, B., P. Hilfinger, K. Ho and J. Larus, "SPUR Lisp: Design and Implementation", Technical Report UCB/Computer Science Dpt. 87/373, U.C. Berkeley, September 1987.