This page intentionally left blank.

A version of this paper will appear in the Fifth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS V). October 1992.

Consistency Management for Virtually Indexed Caches

Bob Wheeler and Brian N. Bershad

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Abstract

A virtually indexed cache can improve performance by allowing cache lookup and address translation to occur in parallel, thus reducing processor cycle time. Unlike physically indexed caches, virtually indexed caches create consistency problems because a physical address may be represented in more than one cache line when it has been accessed through more than one virtual address. *Write-back* virtually indexed caches create additional inconsistencies because memory may become stale with respect to the cache.

In this paper we examine the problem of consistency management for a virtually indexed write-back cache. We assume that the hardware does not support intra-cache consistency. We present a model and software implementation strategy for maintaining consistency with virtually indexed caches.

We present measurements from an implementation of this model on the HP 9000 Series 700 in the context of the Mach operating system. Our measurements show that a virtually indexed cache can be managed with nearly the same cost as that required to manage a physically indexed one, even when used by a virtual memory system that encourages and exploits sharing.

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035, by the Open Software Foundation (OSF), and by a grant from the Hewlett-Packard Corporation. Bershad was partially supported by a National Science Foundation Presidential Young Investigator Award.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, OSF, HP, the NSF, or the U.S. government.

1 Introduction

Virtually indexed caches are becoming increasingly common as architects try to reduce processor cycle times [Kohn 89, Lee 89]. With a virtually indexed cache, the virtual address of a data item selects the cache line in which the item should reside. In contrast, with a physically indexed cache, the virtual address must first be translated into its corresponding physical address, and that address is used to select the cache line. Virtually indexed caches offer a speed improvement over physically indexed caches, since a cache lookup can occur in parallel with address translation. Comparable performance is possible with a physically indexed cache only by tying cache size and associativity to page size [Jouppi 88], by reducing the associativity of virtual to physical mappings [Chiueh & Katz 92], or by introducing a separate pipeline stage for address translation [DEC Alpha 92].

Despite their low-level performance advantages, virtually indexed caches have generally been considered less desirable for operating systems that support arbitrary memory sharing within and between programs. Since the selection of a cache line is based on a virtual address, the contents of a physical address that has been mapped at more than one virtual address may appear in more than one line in the cache at a time. This creates an internal cache consistency problem. The problem becomes more acute with a *write-back* virtually indexed cache because memory can become stale with respect to the cache.

In this paper we describe a solution to the problems of internal cache consistency for machines with virtually indexed write-back caches. Our solution is based on a simple model of consistency that captures the behavior of virtual memory, physical memory, and cache memory. The model defines the consistency of virtually indexed cache lines in terms of the operations that are performed on memory (CPU-read, CPU-write, DMA-read, and DMA-write) and the cache (purge, flush). The model lends itself to a straightforward software solution that relies on virtual memory hardware to deny access to potentially inconsistent data.

From the standpoint of performance, two aspects of the consistency model are important. First, the model delays cache control operations for as long as possible, ensuring consistency only when not doing so would cause the memory system (the cache plus primary memory) to transfer a stale value to either the CPU or a device. In contrast, many previous solutions have forced the memory system into a consistent state at the time that the inconsistency would be created, rather than when it would be detected. Second, our model naturally captures the effect of different virtual addresses that map to both the same physical address and to the same line within the virtually indexed cache. Such *aligned* addresses do not create consistency problems and therefore do not require any consistency management.

1.1 Motivation and goals

We have implemented the machine-dependent layer of the virtual memory system [Rashid et al. 87] in Mach 3.0 [Accetta et al. 86] for the HP 9000 Series 700 [Lee 89]. The HP 9000 Series 700 uses a high-performance RISC-based microprocessor (HP PA-RISC) with separate instruction and data caches that are direct mapped, virtually indexed, and physically tagged. The data cache is write-back. There is no hardware support for consistency when a physical address is represented at more than one cache line. I/O devices that rely on DMA do not snoop the cache. A translation lookaside buffer (TLB) translates virtual page frames into physical page frames in parallel with cache lookup. At the end of the cache lookup, the physical frame number is compared to the cache tag. If they match, the cache access hits. Otherwise the data is retrieved from memory and stored in the cache. If the previous data in the cache line is dirty, it is first written back to memory before being replaced. For the purposes of cache management, the cache exports two operations to the processor. These are *flush virtual address* and *purge virtual address*. Both remove the cache line containing the specified virtual address from the cache. The flush will write the line back to main memory if it is dirty.

As we examined the consistency problem for virtually indexed caches, we began to appreciate the many different kinds of inconsistencies that can occur. For example, a cache line can become stale with respect to other cache lines or memory due to writes by the CPU or a DMA device, or primary memory can become stale with respect to the cache due to the cache's write-back policy. When we considered strategies for resolving the inconsistencies, we frequently discovered inconsistencies that could be ignored, or that could be handled lazily. For example, when zeroing data known to be stale in the cache, there is no reason to first purge the data as it will be entirely overwritten before being read. On the other hand, when the CPU reads data provided by a DMA device, previously stale data in the cache must first be purged to ensure that the device's new data is not improperly shadowed. Concerned that we might be taking an *ad hoc* approach to consistency which would result in a less-than-optimal implementation, we sought first to define a consistency model that would enable us to manage a virtually indexed cache correctly and efficiently.

Our consistency model is therefore intended to meet two goals. First, we want to make it easier to reason about the cache control and virtual memory operations that are required to ensure consistency. This allows us to evaluate an implementation strategy in terms of correctness (does it ensure consistency?), necessity (does it provide more consistency than is actually needed?), and efficiency (can certain operations be combined in order to achieve the same level of consistency with lower cost?). Our model permits inconsistencies within the memory system as long as those inconsistencies never result in stale data being returned to either the CPU or a device. Moreover, our model encourages the operating system to select virtual addresses that naturally align within the cache so that consistency operations can be avoided. The use of these techniques has resulted in application performance improvements of up to 10%.

Our second goal is to better understand the virtual cache management strategies that have been implemented in other systems. When we first began our work, we surveyed the literature to understand the approaches taken by others [Cheng 87, Chao et al. 90]. We found it difficult to describe succinctly the differences between the systems because we had no common reference point. Our model provides that reference point, and allows us to compare our solution to previous ones.

1.2 The rest of this paper

In Section 2 we describe the consistency and performance issues that arise in the use of a virtually indexed cache. In Section 3 we present a consistency model for such caches. In Section 4 we describe an implementation of the model that relies on the virtual memory system. We also describe changes to the Mach operating system that eliminate unnecessary cache inconsistencies. In Section 5 we present measurements that show the relative impact of our model and implementation strategy. In Section 6 we discuss related work. Finally, in Section 7 we present our conclusions.

2 Virtually indexed cache consistency

In this section we discuss several issues that arise when implementing a memory management system for a machine with a virtually indexed write-back cache. These issues relate to aliases, new mappings, and DMA.

2.1 The virtual memory model

Mach provides a hierarchical virtual memory model in which each process runs in its own address space. Memory can be shared between processes, although there is no requirement that it be shared at the same address everywhere. An alternative model places all processes in a single, global virtual address space in which naming and protection are orthogonal. From an operating systems perspective, exporting the large address space model to applications opens up a large number of research issues [Chase et al. 92]. These issues are complicated by the fact that global address space machines are still the exception and not the rule. Consequently, we are retaining Mach's hierarchical model for our port to the HP 9000 Series 700.

For the purposes of virtually indexed cache management, only one distinction exists between the two addressing models. In the hierarchical model, processes can share memory at different virtual addresses, and therefore possibly different cache lines. In the global model, memory is shared at the same address in all processes. This eliminates consistency problems due to sharing (which is defined as aliasing in the next subsection), but does not solve the problems that arise during the creation of new mappings or DMA-based I/O (Sections 2.3 and 2.4).

2.2 The problem with aliases

An *alias* occurs whenever the same physical address is mapped at more than one virtual address. In a virtually indexed cache, where the virtual address is used to select the cache line, an alias can result in a data item being in more than one cache line at a time. It is the job of the cache management system to ensure that references to aliased data yield consistent values.

Write-back caches exacerbate the alias problem. A write-back cache can improve processor performance by reducing the frequency of stores to main memory. Only the replacement of a dirty cache line requires a store to memory, enabling physical memory to become stale with respect to the cache. This can cause reads through one virtual address to return stale values if the data was written through another. Writes can also be lost if a physical address is dirty in more than one cache line because one or both dirty lines can be written back to physical memory in any order.

Some solutions

Some solutions to the alias problem, such as the global address space model mentioned earlier, disallow aliases altogether. Others preclude arbitrary aliasing by forcing shared data to reside at virtual addresses that *align* in the cache, or by requiring that shared data be non-cacheable [Cheng 87]. Two virtual addresses align if they both map to the same cache line. In a physically tagged cache, aligned aliases can be resolved without going to memory. In a virtually tagged cache, aligned aliases compete for the same cache line, although there is no consistency problem.

Other solutions allow arbitrary aliases, but rely on consistency protocols implemented in hardware [Wang et al. 89, Knapp & Baer 85] or software [Chao et al. 90]. Hardware protocols use a *reverse translation buffer* to determine if a physical address is aliased in more than one cache line, and allow only the most recent alias to be valid. Software protocols allow aliases to be read-only. On the first write to a read-only alias, a page-fault occurs, the page is mapped writable, and any other read mapping is broken. On the next read through a different virtual address, the write-mapping is broken and the alias becomes read-only again. The cache may require flushing or purging during transitions.

When do aliases occur?

Aliases occur for two general reasons. Either applications explicitly request that physical pages be multiply mapped, or the operating system uses multiple mappings to implement techniques such as copy-on-write to reduce the overhead of memory management [Young et al. 87]. In either case, only unaligned aliases pose a problem, as aligned aliases map to the same line in the cache.

Despite the ability to share memory at arbitrary addresses, we are aware of few applications that rely on this feature. Even applications that share in sophisticated ways [Appel & Li 91] can generally do so without specifying the address at which shared data must be mapped [Li 92]. The name of a piece of virtual memory is much less important than other attributes, such as its contents, pageability, protection, and reference information.

In contrast to applications, the operating system itself is a more aggressive client of virtual memory sharing primitives. As a result, it might be more inclined to require unaligned aliases. In practice, though, this turns out not to be case for the same reason as with applications. The virtual memory system can therefore generally select aligning virtual addresses for shared data. We discuss this characteristic further in Section 4.2.

Nevertheless, there will always be cases where it may be more convenient to place shared memory at specific virtual addresses (such as with shared persistent data structures). Consequently, the cache management system must deal with these aliases correctly.

2.3 The problem with new mappings

Aliases introduce a potential memory inconsistency whenever the number of virtual mappings to a given physical page is greater than one. An additional consistency problem is created whenever a physical page is newly mapped, that is, the number of mappings to a physical page changes from zero to one. Unlike the alias problem, the new mapping problem is not solved by the use of a single, global address space.

New mappings are a problem because the cache may contain data brought in through an old (but now non-existent) mapping. For example, suppose that virtual address V is mapped to physical page P, written, and then unmapped so that P has no corresponding virtual address. This could happen for any one of a number of reasons, such as the termination of an address space, or the remap of P from one virtual address to another. If V' is then mapped to P, yet V and V' do not align in the cache, a read or write through V' could access data from page P in physical memory that is older than the data from page P in the cache. Furthermore, write-back of dirty data that was once mapped at V could overwrite physical memory that had been written through V'.

A straightforward way to ensure consistency for new mappings is to clean the cache with a flush or a purge when a virtual mapping to a physical page is removed. The flush or purge ensures that the *next time* the physical page is accessed, none of its data will be in the cache. This approach, for example, is taken in [Cheng 87].

While correct, the approach is unnecessarily eager. It is possible that the next time the page is mapped, it would be mapped at a virtual address that aligns with the last assigned virtual address. By removing the data from the cache at unmap, subsequent accesses will involve slower fetches from main memory, thereby degrading performance [Chao et al. 90]. On the other hand, if the next mapping to the page is not aligned, eager removal reduces the likelihood that the data would be naturally replaced by other references. On the HP 9000 Series 700, for example, a purge or flush of a virtual address can be up to seven times slower when the data is in the cache as opposed to when it isn't.

An alternative approach is to delay consistency management until the new mapping is created. It is not necessary to purge or flush the cache of data when a virtual address is unmapped by the operating system. Other structures, however, such as TLB and page table entries, must be invalidated to deny access to the data in the memory system.

As with aliases, consistency operations for new mappings are only required when the new mapping to a physical address does not align with the previous mapping to that physical address. If the previous and new addresses do not align, the previous address may need to be flushed and the new address may need to be purged. The flush through the previous mapping ensures that any writes to it will take effect in memory before any reads through the new mapping. The purge of the new mapping ensures that any reads through it will return fresh data from memory.

The consistency issues of new mappings are orthogonal to the operating system issues of initializing data in a newly mapped page. Security concerns dictate that a newly mapped physical page not contain data left over from a previous mapping, otherwise one process' data could be made available to another process through a remapping. Proper page preparation must occur with both physically and virtually indexed caches. We discuss two optimizations that are possible when preparing a newly mapped page in Section 4.

2.4 The problem with DMA-based I/O

DMA-based I/O is another source of cache inconsistency, and hence potential cache management overhead. The I/O problem is independent of the cache architecture (virtual or physical) or address space model (hierarchical or global).

DMA devices support two memory-oriented operations: DMA-write and DMA-read. DMA-write causes the device to transfer data into the memory system. DMA-read causes the device to read data from the memory system. Prior to issuing a DMA-write, the CPU must ensure that the physical addresses written by the device will not be overwritten by previously dirtied data still in the cache. Prior to accessing the data, the CPU must ensure that old data in the cache is not shadowing new data in memory. Prior to issuing a DMA-read, the CPU must ensure that data at the addresses being read by the device has been flushed to main memory.

2.5 The cost of cache management

In managing a virtually indexed cache, an operating system has two responsibilities. First, it must strive to reduce the frequency of unaligned aliases and new mappings by taking the cache index function into account when defining virtual to physical mappings. Second, when unaligned mappings do occur and consistency management is required, the operating system should provide it with as little overhead as possible. For example, because purges and flushes of virtual addresses that are not present in the cache should be less expensive than those that are, they should be delayed for as long as possible. A delayed cache management operation may even be avoided altogether if it is known to be obviated by a subsequent and otherwise necessary operation. For example, copying into or zeroing a cache line overwrites its contents entirely, making an earlier purge of stale data in that line unnecessary.

To demonstrate the importance of meeting these responsibilities, we have measured the performance of several benchmark programs on two versions of the Mach 3.0 kernel running on the HP 9000 Series 700 (Model 720). An operating system server running at user level provides Unix functionality [Golub et al. 90]. The benchmarks are *afs-bench, latex-bench,* and *kernel-build.* The first is a version of the Andrew File System benchmark [Satyanaranyanyan et al. 85] that runs a file-intensive shell script. The second formats a version of this paper using LaTeX. The third builds a version of the Mach kernel from about 200 source files.

The two versions of the kernel are identical with the exception of their cache management policies. In the first version of the system (labeled "old"), neither the kernel nor the user-level operating system server attempt to align virtual addresses. Both the kernel and the Unix server run under the mis-assumption that the cache is physically indexed, while low-level machine-dependent software guarantees consistency through a simple strategy for cache management. [1] On a write to an aliased physical page, all other mappings to that page are broken. On a read to an unmapped aliased page, any existing writable mapping is broken and the faulting address is marked read-only. Whenever a virtual to physical mapping is broken, the page is removed from the cache with a flush (if dirty) or a purge. In the second version of the system (labeled "new"), which uses the cache management strategies laid out in the next two sections, careful alignment and delayed consistency are used extensively.

Table 1 summarizes the performance of the benchmarks on the two systems in terms of program execution time and number of cache consistency operations. The numbers represent the average from the last two of three runs on an otherwise idle machine.[2] Despite the fact that none of the benchmarks directly stress Mach's virtual memory system (they are all Unix programs), their indirect reliance on the kernel and the Unix server can incur a significant amount of cache management overhead stemming from the cost of having to purge and flush pages from the cache. Overall, the new system's cache consistency policies improve execution times by between 5% and 10%. A more detailed performance analysis is presented in Section 5.

We also ran an entirely contrived benchmark, which is not shown in the table, that exposes the benefit of aligning shared mappings. A single thread repeatedly wrote one physical address through two virtual addresses. When the virtual addresses were aligned, a loop of 1,000,000 writes completed in a fraction of a second. When unaligned, the loop took over 2 minutes.

Program	Elapsed Time (seconds)			Page Flushes ($\times 10^3$)		Page Purges ($\times 10^3$)	
	old	new	% gain	old	new	old	new
afs-bench	66.0	59.4	10%	77.41	7.754	62.09	22.61
latex-paper	5.8	5.5	5%	2.029	.143	1.52	.28
kernel-build	678.9	620.9	8.5%	602.9	57.88	418.41	171.62

Table 1: *Performance of several common benchmarks using two approaches to consistency management on two versions of the Mach 3.0 kernel (version MK67) and Unix server (version UX28).*

[1] This version derives its cache management policy from an earlier version of Mach worked on by one of this paper's authors while at the University of Utah.

[2] The first run, which reflects performance on a cold instruction, data, and file system buffer cache, took slightly longer due to increased file system read activity.

3 A model for virtually indexed cache management

The previous section enumerated different situations that arise in consistency management for virtually indexed write-back caches. There are cases where caches need to be flushed or purged when the inconsistency is created (write access to an alias, DMA). In other cases, the flush or purge can be delayed (remove a mapping). Finally, in other cases, the flush or purge can be avoided altogether (virtual addresses align). These situations also include a large number of cases that can be handled specially to improve performance.

The problems introduced by aliases, new mappings and DMA-based I/O can be distilled into the single problem of ensuring that the memory system never transfers an inconsistent value to either the CPU or a DMA device.

In this section we present a consistency model for a virtually indexed cache that allows us to solve this problem. The model expresses the consistency state of each cache line with respect to virtual and physical addresses. Transitions between consistency states occur as a result of operations applied to the memory system by the CPU and devices. We define the states and transitions in such a way as to prevent the detection of data in an inconsistent state, allowing us to delay and sometimes omit cache purge and flush operations. We also use this model to describe the behavior and cache management requirements for several other cache architectures.

3.1 A restatement of the problem

A correctly functioning memory system must never transfer stale data to either the CPU or a DMA-device. Because data is kept only in memory or the cache, stale data may be transferred for only two reasons:

1. Memory is stale with respect to the cache. The most recently written data is not in memory. In the case of a CPU access, the most recently written data may be in a cache line other than the one being accessed. In the case of DMA access, the most recently written data may be in the cache.

2. The cache is stale with respect to memory. The most recently written data is in memory, but the cache is returning stale data, either to the CPU (in the case of a CPU access) or to memory (in the case of a cache write-back).

We can solve both problems by keeping track of cache lines that are stale, and by ensuring that stale lines are never transferred out of the cache.

3.2 A solution to the problem

For any virtual address, a cache line can be in one of four states: empty, present, dirty, or stale (E,P,D or S).

An empty cache line does not contain the data at the virtual address that was used to select the cache line; an access through that virtual address results in a cache miss and a value being transferred from main memory. A present line is one that contains the correct data at the virtual address. A dirty line is like a present line, except that the line has been written by the CPU and may therefore be inconsistent with respect to memory or another cache line. A stale line is one for which the data at the cached physical address is inconsistent with a more recently written version either in memory or in another cache line.

Six events can change the consistency state of cache lines with respect to a virtual address. These are *CPU-read, CPU-write, DMA-read, DMA-write, Purge,* and *Flush.* The first four operations can create inconsistencies, while the last two can resolve them.

Figure 2 enumerates the state transitions that must occur during each operation in order to ensure consistency. The first column names the operations that can be applied to a target virtual address. The second column describes the transitions that must occur for the target cache line, depending on the target line's current state. The target line is the one selected by the cache index function for the target virtual address. The third column describes the transitions that must occur for all other cache lines which share the same mapping as the target virtual address but are not aligned. Transitions labeled with *purge* and *flush* indicate the cache consistency operation which is required to force the transition.

Initially, at power up, all cache lines for all virtual addresses are in the empty state (the cache can be purged to ensure this). Any operation that can modify state must be caught and acted upon before the operation takes place, and the requisite state transitions must occur atomically. This guarantees, for example, that an empty line is not marked present and then read before dirty data in another similarly mapped line has been flushed to memory.

A CPU-read through a cache line in the empty state must cause that cache line to enter the present state ($\mathrm{E} \longrightarrow \mathrm{P}$), indicating the data's presence in the cache at the particular line. To ensure that the previously empty line returns the most recently written data, any similarly mapped but unaligned cache line in the dirty state must be flushed to main memory, leaving the flushed line empty ($\mathrm{D} \stackrel{flush}{\longrightarrow} \mathrm{E}$). A CPU-read of a stale line requires that the line first be purged.

A CPU-write forces an empty, present, or dirty line to enter the dirty state ($[\mathrm{E},\mathrm{P},\mathrm{D}] \longrightarrow \mathrm{D}$). Other similarly mapped but unaligned cache lines not in the empty state must enter the stale state ($[\mathrm{P},\mathrm{S}] \longrightarrow \mathrm{S}$) or the empty state ($\mathrm{D} \stackrel{flush}{\longrightarrow} \mathrm{E}$). As with a CPU-read, a CPU-write to a stale line requires purging.

DMA-read and DMA-write are similar to CPU-read and CPU-write, respectively. Their similarity is revealed by the set of equivalent transitions for similarly mapped but unaligned cache lines for CPU-read and DMA-read, and CPU-write and DMA-write. DMA does not go through the cache, so all cache lines that contain the physical address referenced by the DMA operation share the same transitions. In contrast, CPU-read and CPU-write of a virtual address affect the target cache line (and state) differently than the cache lines (and states) for similarly mapped but unaligned virtual addresses. One difference between CPU-write and DMA-write is that a DMA-write under a dirty cache line only requires that the line be purged rather than flushed, since the DMA-write will cause the data in memory to be overwritten.

The states themselves are pessimistic with respect to consistency and the operation of a cache. Because we do not consider the cache replacement policy in the transitions, it is possible to have a cache line in the present state in terms of the model, yet not physically present in an actual cache (although the converse is not possible). Such pessimism does not influence correctness because a flush or purge of a physically non-present line has no effect on the system.

Operation on target address	Target cache line	All other similarly mapped but unaligned cache lines
CPU-read	$\mathrm{E} \longrightarrow \mathrm{P}$	$\mathrm{E} \longrightarrow \mathrm{E}$
	$\mathrm{P} \longrightarrow \mathrm{P}$	$\mathrm{P} \longrightarrow \mathrm{P}$
	$\mathrm{D} \longrightarrow \mathrm{D}$	$\mathrm{D} \stackrel{flush}{\longrightarrow} \mathrm{E}$
	$\mathrm{S} \stackrel{purge}{\longrightarrow} \mathrm{E} \longrightarrow \mathrm{P}$	$\mathrm{S} \longrightarrow \mathrm{S}$
CPU-write	$\mathrm{E} \longrightarrow \mathrm{D}$	$\mathrm{E} \longrightarrow \mathrm{E}$
	$\mathrm{P} \longrightarrow \mathrm{D}$	$\mathrm{P} \longrightarrow \mathrm{S}$
	$\mathrm{D} \longrightarrow \mathrm{D}$	$\mathrm{D} \stackrel{flush}{\longrightarrow} \mathrm{E}$
	$\mathrm{S} \stackrel{purge}{\longrightarrow} \mathrm{E} \longrightarrow \mathrm{D}$	$\mathrm{S} \longrightarrow \mathrm{S}$
DMA-read	$\mathrm{E} \longrightarrow \mathrm{E}$	$\mathrm{E} \longrightarrow \mathrm{E}$
	$\mathrm{P} \longrightarrow \mathrm{P}$	$\mathrm{P} \longrightarrow \mathrm{P}$
	$\mathrm{D} \stackrel{flush}{\longrightarrow} \mathrm{E}$	$\mathrm{D} \stackrel{flush}{\longrightarrow} \mathrm{E}$
	$\mathrm{S} \longrightarrow \mathrm{S}$	$\mathrm{S} \longrightarrow \mathrm{S}$
DMA-write	$\mathrm{E} \longrightarrow \mathrm{E}$	$\mathrm{E} \longrightarrow \mathrm{E}$
	$\mathrm{P} \longrightarrow \mathrm{S}$	$\mathrm{P} \longrightarrow \mathrm{S}$
	$\mathrm{D} \stackrel{purge}{\longrightarrow} \mathrm{E}$	$\mathrm{D} \stackrel{purge}{\longrightarrow} \mathrm{E}$
	$\mathrm{S} \longrightarrow \mathrm{S}$	$\mathrm{S} \longrightarrow \mathrm{S}$
Purge	$\mathrm{E} \longrightarrow \mathrm{E}$	$\mathrm{E} \longrightarrow \mathrm{E}$
	$\mathrm{P} \longrightarrow \mathrm{E}$	$\mathrm{P} \longrightarrow \mathrm{P}$
	$\mathrm{D} \longrightarrow \mathrm{E}$	$\mathrm{D} \longrightarrow \mathrm{D}$
	$\mathrm{S} \longrightarrow \mathrm{E}$	$\mathrm{S} \longrightarrow \mathrm{S}$
Flush	$\mathrm{E} \longrightarrow \mathrm{E}$	$\mathrm{E} \longrightarrow \mathrm{E}$
	$\mathrm{P} \longrightarrow \mathrm{E}$	$\mathrm{P} \longrightarrow \mathrm{P}$
	$\mathrm{D} \longrightarrow \mathrm{E}$	$\mathrm{D} \longrightarrow \mathrm{D}$
	$\mathrm{S} \longrightarrow \mathrm{E}$	$\mathrm{S} \longrightarrow \mathrm{S}$

Table 2: *Cache line state transitions. These transitions must occur to ensure that the memory system never returns inconsistent data to either the CPU or a device.*

Correctness of the state transitions

We can return to the problem restatement and show that the state transitions ensure a correct solution. The transition rules guarantee that neither:

1. *memory is stale with respect to the cache, or*
2. *the cache is stale with respect to memory*

can result in stale data being transferred by the memory system to the CPU or a device.

We avoid the first problem because a cache line cannot leave the empty state until memory is consistent with the most recent update. For updates caused by a CPU-write, consistency is enforced by the flush of the dirty line. For updates caused by a DMA-write, consistency is implied by the fact that the DMA device performed the most recent write to memory, and that all equivalently mapped cache lines are either empty or stale.

We avoid the second problem because stale lines are never transferred by the cache to either the CPU or memory. In the first case, a stale line must first be purged before it can be read or written. In the second case, only dirty data can ever be written back by the cache, and the transitions for CPU-write and DMA-write guarantee that data corresponding to a physical address is dirty in at most one cache line (one for CPU-write, zero for DMA-write).

3.3 Application to other architectures

Although we are primarily concerned with direct-mapped, virtually indexed, write-back caches on a uniprocessor, the consistency model can be applied to other memory system architectures.

- *Write-through caches.* In a write-through cache, memory is never stale with respect to the cache. Consequently, the dirty state can be replaced with the present state, and all redundant transitions can be eliminated. There is also no need for the flush operation.

- *Physically indexed caches.* With a physically indexed cache, all similarly mapped virtual addresses naturally align in the cache, so the third column in Table 2 becomes irrelevant. Only DMA-write and DMA-read create consistency problems, and those are handled with the transitions in the second column. As with a virtually indexed cache, write-back and write-through physically indexed caches are distinguished only by the existence of a dirty state.

- *DMA can access the cache.* In a system in which DMA can access the cache, CPU-read and DMA-read fold into a single *read* operation, and CPU-write and DMA-write fold into a single *write* operation. The transitions on *read* and *write* are the same as for CPU-read and CPU-write in Table 2.

- *Set-associative caches.* For a set-associative cache, the consistency rules remain the same since consistency within a set is ensured by hardware. That is, the physical tags associated with each entry are guaranteed to be unique within a set.

- *Cache-coherent multiprocessors.* The caches in a cache-coherent multiprocessor can be viewed as a distributed set-associative cache. Equivalent cache lines from each processor constitute an element of a set, while hardware ensures inter-cache (intra-set) consistency. As with set-associative caches, no changes to the transition rules are required.

4 Implementing the model

We now describe an implementation strategy for the cache consistency model described in the previous section. Our strategy requires that the hardware have the following characteristics:

- The first address within any virtual page aligns in the cache with the first address of any other virtual page if and only if all addresses within those two virtual pages align.

- Reads and writes to individual virtual memory pages can be caught by the operating system kernel.

The first requirement allows us to maintain consistency state on a "cache page," rather than a cache line, basis. A cache page is the set of cache lines onto which the cache index function maps all virtual addresses within a virtual page. A cache page is the same size as a virtual page, and a virtually indexed cache contains n cache pages, where n is the cache size divided by the page size.

All aligned virtual pages map into the same cache page. With cache pages, all cache lines within a cache page are defined to have the same consistency state. This enables the use of standard virtual memory hardware to implement the state transitions, and reduces the amount of cache state information from O(number of cache lines $\times$ number of virtual addresses in use) to O(number of cache pages $\times$ number of physical pages). The reduction in state is because of the larger coverage of a cache page, and because cache state information is required only for pages that are physically resident. A virtual memory system already denies access to nonresident pages, so cache consistency for these pages is not an issue.

The second requirement guarantees that we can detect cache page state changes, and that we can prevent stale cache pages from being accessed by the CPU.

These requirements are met by the HP 9000 Series 700. While it is conceivable that one could build a virtually indexed cache for which the first assumption does not hold, we are presently aware of no such cache, nor of any compelling reason to build such a cache. Any system with a memory management unit should be able to satisfy the second assumption. Consequently, no special hardware is required to implement the algorithm. Note that if the hardware supports multiple page sizes, then the operating system must take additional care to ensure that a cache page is mapped only by similarly sized virtual pages.

4.1 The algorithm

The cache control algorithm should be invoked during any operation that could change the consistency state of cache pages. Virtual memory protections are set to detect state transitions during CPU-reads and CPU-writes.[3] Operating system software should invoke the algorithm before scheduling DMA operations.

Pseudo-code for the algorithm is shown in the Figure 1. As input, the algorithm takes a virtual address, an operation type, and two booleans which indicate whether stale and dirty cache data will ever be read. The algorithm modifies cache state information and ensures that stale data is never mapped. The code in Figure 1 has been adapted directly from that running in the machine-dependent module of Mach's virtual memory system for the HP 9000 Series 700. Atomicity on a uniprocessor is guaranteed by running the code sequence with interrupts disabled. As presented, the code is not safe for use on a multiprocessor, although it could be made so with appropriate data structure locking.

Data structures

The algorithm relies on several data structures. Each physical page p in the system is represented by a data structure, $P[p]$, that contains a list of virtual mappings for the page, $P[p].mappings$, and the cache page state for that page. The cache page state consists of two bit vectors, $P[p].mapped$ and $P[p].stale$, and a single dirty bit, $P[p].cache\_dirty$.

Each bit in the vectors corresponds to a particular cache page. The bit vector $P[p].mapped$ indicates which cache pages have been mapped by the CPU, and therefore which cache pages may contain data from a given physical page p. The bit vector $P[p].stale$ indicates which cache pages may contain stale data from a given physical page p. The $P[p].cache\_dirty$ bit for a physical page p indicates that the physical page could be dirty within a cache page. That dirty cache page c is given by the entry for which $P[p].mapped[c]$ is true. The $P[p].cache\_dirty$ bit is cleared when the dirty cache page is removed from the cache with either a flush or a purge.

[3] The algorithm assumes that illegal page accesses, such as a write to a text page, have been filtered out in earlier stages of the kernel's fault handler.

```
CacheControl(virtual_address target_va, operation op,
             boolean will_overwrite, boolean need_data)
begin
phys_page  p = va_to_physical_page(target_va);
cache_page c = va_to_cache_page(target_va);

if (P[p].cache_dirty) then
    cache_page w = find_mapped_cache_page(P[p]);
    /* Clean cache if dirty page is not target  */
    if (op == DMA_WRITE OR op == DMA_READ OR w != c)
    then if (need_data) then
            flush_cache_page(w);
        else
            purge_cache_page(w);
        end
        P[p].cache_dirty = FALSE;
    end
end

if ((op == CPU_READ OR op == CPU_WRITE)
    AND P[p].stale[c]) then
    if (NOT will_overwrite) then
       purge_cache_page(c);
    end
    P[p].stale[c] = FALSE;
end

/* DMA input operations and write operations force
 * all mapped and stale cache pages to stale, and
 * all mapped pages to unmapped. */
if (op == DMA_WRITE OR op == CPU_WRITE) then
    P[p].stale = bitwise_or(P[p].mapped, P[p].stale);
    bitwise_clear(P[p].mapped);
    /* For a write, mark the target cache page
     * as not stale, dirty, and mapped. */
    if (op == CPU_WRITE) then
        P[p].stale[c] = FALSE;
        P[p].cache_dirty = TRUE;
        P[p].mapped[c] = TRUE;
    end
end

if (op == CPU_READ) then
    P[p].mapped[c] = TRUE;
end

/* Set mappings for all virtual addresses that
 * map to p to prevent inconsistencies from
 * being perceived, to detect subsequent accesses,
 * and to allow the current operation to complete.*/
foreach (virtual_address v in P[p].mappings) do
    c = va_to_cache_page(v);
    if (P[p].stale[c]) then
        set_protection(v, NO_ACCESS);
    else if (NOT P[p].mapped[c]) then
        set_protection(v, NO_ACCESS);
    else if (op == CPU_WRITE) then
        set_protection(v, READ_WRITE_ACCESS);
    else if (op == CPU_READ) then
        set_protection(v, READONLY_ACCESS);
    end
end
end CacheControl.
```

Figure 1: *Pseudo-code sequence implementing consistency for a virtually indexed write-back cache.*

Together, the bit vectors and the dirty bit encode the consistency state of every cache page c with respect to all virtual addresses that map to that physical page and cache page. This encoding is shown in Table 3.

The CPU's access through any virtual address V, where V maps to physical page p and cache page c, is determined by $P[p].mapped[c]$ together with $P[p].cache\_dirty$. If $P[p].mapped[c]$ is false (the cache page is empty or stale), then CPU access through V must fault. This allows the consistency state to be updated by the cache control algorithm. If $P[p].mapped[c]$ is true but $P[p].cache\_dirty$ is false (the cache page is present), then write access through any V that maps to p must be disabled in order to catch the write, and to mark the cache page as dirty and other cache pages as stale.

The hardware does not guarantee consistency between separate instruction and data caches. Consequently, virtual addresses containing data never align with those containing instructions, even when they are equivalent. In the implementation, it is necessary to maintain cache page state for both caches, and to interpret a virtual address in the context of the cache in which it will be found. For simplicity, though, the algorithm shown in Figure 1 assumes that a read access includes both instruction and data fetches.

Explanation of the code

The code in Figure 1 is broken into six stanzas. The first stanza computes the physical page and target cache page corresponding to the target virtual address.

The second stanza removes the contents of a dirty cache page in the case that it is not the target cache page. A dirty page can be mapped through only one cache page, and the operation `find_mapped_cache_page` returns that cache page.

The third stanza ensures that the target cache page is not stale. This is only relevant for a CPU access.

The fourth stanza ensures that writes into the memory system cause all mapped pages to become stale and thus no longer mapped. In the case of a CPU-write, the written page is marked as mapped and not stale, and the physical page is marked as dirty. The actual implementation includes an optimization that sets $P[p].cache\_dirty$ whenever the virtual memory system sets the *page-modified* bit yet the number of mapped bits is one. Finally, note that the data structures used by the algorithm lend themselves to efficient state modification. For example, all mapped (present) pages can be marked stale with a bitwise-or of $P[p].mapped$ and $P[p].stale$ into the $P[p].stale$ vector, and a bitwise-clear of the $P[p].mapped$ vector.

The fifth stanza sets the mapped bit for the page on a CPU-read to indicate that the cache page may contain data from the physical page.

The final stanza sets the virtual memory page protections for all mappings to the physical page to be consistent with the cache page state.

Two simple optimizations

The code sequence in Figure 1 includes two simple optimizations that reduce the frequency of purge and flush operations. These are driven by the two parameters `will_overwrite` and `need_data`. In order, their use is described below.

A straightforward way to eliminate stale data from the cache is to purge it through the virtual address at which the data is known to be stale. Another way to eliminate the stale data, though, is to completely overwrite it with new data from the CPU. In particular, if a stale line is known to become completely overwritten before it will be read, then a preliminary purge is not necessary. This situation commonly arises when the virtual memory system prepares a page of memory with either a *copy-page* or *zero-fill* operation. In both cases, the CPU completely overwrites the page with new data before any other access to the page occurs. In such cases, the page can be allowed to leave the stale state without a preliminary purge (`will_overwrite` is true).

A second optimization relates to the assumption that dirty data is also useful data. This assumption is not always valid. For example, consider the case of a new mapping where a physical page that had previously been mapped into one address space is being remapped into another, and then copied into or zeroed. Clearly, the previous contents of the physical page are no longer useful. Therefore, if the page is dirty, it can be purged instead of flushed (`need_data` is false).

4.2 Eliminating inconsistencies

Although the algorithm delays consistency operations and handles aligned mappings, it does not reduce the frequency of unaligned virtual addresses. We found that such accesses occurred frequently in Mach because the system was initially designed for use on machines with physically indexed caches where virtual address selection was not a factor in cache performance. Consequently, we made three changes to the system to reduce the frequency of unaligned virtual mappings. These changes allow the virtual memory system to select the virtual address at which a physical page is mapped so that it aligns with the previous virtual address bound to that physical page. None of the changes have af-

Cache page state	$P[p].mapped[c]$	$P[p].stale[c]$	$P[p].cache\_dirty$
Empty	false	false	–
Present	true	false	false
Dirty	true	false	true
Stale	false	true	–

Table 3: *Correspondence between cache page state and data structures maintained by the algorithm.*

fected the system's functionality, only its performance, which is discussed in the next section.

Pages passed during IPC operations

A large number of virtual memory remapping operations correspond to physical pages being passed as part of interprocess communication (IPC) messages. The kernel's IPC code transfers a physical page from one virtual address to another [Young et al. 87]. The kernel is free to select any destination virtual address, so choosing one that aligns with the source address guarantees that no cache management operation is necessary. The destination virtual address, though, was originally chosen according to a first-fit strategy, so the source and destination virtual addresses rarely aligned. Consequently, the old virtual address, which would generally be dirty since it contained data generated by the sender, would be flushed, and the new virtual address would be purged.

We modified the IPC code to select an address in the receiver that aligns in the cache with the sender's.

Preparing new pages with copy and zero-fill

The kernel can prepare a new page with data using copy and zero-fill. The first operation copies data from one physical page to another, and the second clears a physical page by filling it with zeros. Page preparation in Mach is split between machine-independent and machine-dependent components. The machine-independent component deals with virtual addresses and initiates page preparation in response to demands on the virtual memory system. The machine-dependent component deals with physical pages and implements the copy and zero-fill operations. With a virtually indexed cache, a page should be prepared through a virtual address that aligns with the ultimate mapping for the page. This consideration is unimportant on a machine with a physically indexed cache, however, so the preparation routines (which were designed assuming such a cache) were not passed the ultimate virtual address.

We extended the machine-dependent interface so that the machine-independent layer could pass the ultimate virtual address down to the page preparation routines. A similar extension was introduced by the Tut project [Chao et al. 90].

Shared pages in the Unix server

Mach's user-level Unix server allocates and shares several pages of memory with each Unix process. These pages are expected to be used as a high-bandwidth, low-latency channel for passing information between applications and the Unix server. In the initial version of the system, the Unix server requested that the shared pages be allocated at a specific virtual address in its own and each process' address space. These pages did not align, so accesses resulted in frequent consistency faults.

We changed the Unix server so that these pages can be allocated at addresses determined by the virtual memory system, thereby aligning.

5 Performance

We have measured the three benchmark programs described in Section 2.5 on the HP 9000 Series 700 (Model 720). We ran each benchmark on six successive configurations of the Mach 3.0 kernel. Table 4 shows the performance statistics averaged over the last two of three consecutive runs of each benchmark on an otherwise unloaded system. The table shows the elapsed time, operation counts, and average cycle counts across various configurations for each of the programs. The cycle counts were gathered using the processor's on-chip cycle counter.

The configurations ranged from one having only minimal cache consistency machinery (that described as "old" in Section 2.5 and labeled "A" in the table) to one having all of the machinery described in the previous section (that described as "new" in Section 2.5 and labeled "F" in the table). Each successive version provides a cumulative and more efficient solution to consistency management than the previous by including an additional optimization. In order, we (B) delay flush and purge operations until a virtual address is reused (*+lazy unmap)*, (C) allow the kernel to select virtual addresses for multiply mapped pages so that they align in the cache (*+align pages*), (D) support aligned page preparation (*+aligned prepare*), (E) replace flushes with purges when old data will never be used (+`need_data`), and (F) eliminate purges when the destination cache page is being completely overwritten (+`will_overwrite`). The bottom two rows of the table show total counts and times for the three benchmarks running on the final and most efficient configuration.

In the benchmarks, all DMA activity is due to disk access. There are no disk reads, which correspond to DMA-writes, for either of the first two benchmarks. This is because all file system reads are satisfied by the Unix buffer cache. The third benchmark, which accesses substantially more file system data than the other two, does require disk reads. The low cyle count for DMA-read flushes is because the file system's write-behind policy introduces delays between the dirtying and subsequent flushing of a buffer cache block, so the dirty lines tend to be written back naturally.

5.1 Interpretation of results

Careful cache management improves application performance. Moving downward over successive configurations for a given benchmark shows that performance improves by delaying cache consistency operations, by aligning pages, and by exploiting the semantics of data use. For example, the decrease in page purges between configurations "A" and "B" reflects the fact that a physical page is often unmapped, and then remapped

Program	Time	VM Mapping Faults		Consistency Faults		Total Page Flushes		DMA Read Flushes		Data to Inst. Copies		Page Purges (Instr.)		Page Purges (Data)		DMA Write Purges	
	secs	cnt $\times 10^3$	avg cyc	cnt $\times 10^3$	avg cyc	cnt $\times 10^3$	avg cyc	cnt $\times 10^3$	avg cyc	cnt $\times 10^3$	avg cyc	cnt $\times 10^3$	avg cyc	cnt $\times 10^3$	avg cyc	cnt $\times 10^3$	avg cyc
A. *afs-bench*	66.0	60.65	1507	24.07	2165	77.41	1327	0.726	316	0	0	18.30	1099	43.79	387	0	0
B. +lazy unmap	64.1	60.61	1786	23.74	1413	70.36	1148	0.703	313	7.012	827	4.874	1105	42.86	295	0	0
C. +align pages	62.0	60.58	1712	0.046	1288	54.45	1328	0.695	304	7.058	831	4.803	1106	42.76	294	0	0
D. +aligned prepare	59.5	60.56	1272	0.046	1293	25.19	605	0.681	311	7.058	701	4.755	1107	14.25	297	0	0
E. +`need_data`	59.9	60.59	1266	0.046	1338	7.753	669	0.695	322	7.058	703	4.767	1103	31.80	470	0	0
F. +`will_overwrite`	59.4	60.58	1264	0.046	1336	7.754	666	0.696	310	7.058	701	4.857	1105	17.75	613	0	0
A. *latex-paper*	5.8	1.481	1335	0.963	2235	2.029	1253	0.056	311	0	0	0.273	1126	1.250	456	0	0
B. +lazy unmap	5.8	1.477	1647	0.952	1480	1.649	1232	0.052	324	0.088	540	0.003	1127	0.485	301	0	0
C. +align pages	5.8	1.474	1469	0	0	1.107	1621	0.052	303	0.088	547	0.003	1120	0.361	296	0	0
D. +aligned prepare	5.6	1.479	1135	0	0	0.423	729	0.055	305	0.088	539	0.002	1122	0.161	296	0	0
E. +`need_data`	5.7	1.472	1131	0	0	0.140	448	0.052	304	0.088	534	0.003	1120	0.310	1197	0	0
F. +`will_overwrite`	5.5	1.481	1139	0	0	0.143	463	0.055	310	0.088	560	0.002	1118	0.278	1310	0	0
A. *kernel-build*	678.9	408.5	1388	249.3	2223	602.9	1354	5.306	310	0	0	81.11	1098	337.3	392	0	0
B. +lazy unmap	661.3	407.7	1772	247.6	1457	541.9	1115	5.365	324	52.06	608	28.31	1104	309.1	298	1.000	368
C. +align pages	641.9	407.7	1706	1.001	1381	405.4	1278	5.258	326	52.58	608	27.06	1105	303.0	294	1.549	400
D. +aligned prepare	626.8	408.8	1204	1.002	1402	197.6	473	5.283	321	52.58	552	30.61	1104	101.1	297	1.572	477
E. +`need_data`	627.6	408.9	1211	1.015	1406	57.88	531	5.300	318	52.57	553	29.77	1104	239.6	390	1.600	497
F. +`will_overwrite`	620.9	407.6	1199	0.986	1409	57.88	529	5.305	328	52.58	549	29.02	1104	142.6	452	1.537	462
Total for "F"	685.8	469.7	3602	1.032	2745	65.78	1658	6.056	948	59.72	1810	33.88	3327	160.6	2375	1.537	462
Seconds for "F"	685.8	11.34	1.6%	0.03	0%	0.72	.10%	0.04	.01%	0.68	.10%	0.75	.11%	1.51	.22%	0.01	0%

Table 4: *Performance of three benchmark programs using variously configured versions of Mach 3.0 (MK67) running on a 50Mhz HP 9000 Series 700 (Model 720).*

through an aligned virtual address. The slight reduction in purge time for the data cache occurs because the delayed purge reduces the likelihood that the page's data is in the cache. That no such reduction occurs for the instruction cache appears to be an artifact of the 720's implementation which requires constant time to purge the instruction cache, regardless of its contents.

The reduction in flushes between configurations "D" and "E" shows that dirty data is often left in the cache never to be accessed again. This data can be purged, rather than flushed. As expected, the decrease in data cache flushes is offset by an equivalent increase in data cache purges. No increase occurs for purges of the instruction cache because that cache never contains dirty data. Execution time does not improve because the 720 appears to purge no more quickly than it flushes.[4]

The overhead to maintain consistency state is low. The Mach kernel lazily evaluates many virtual memory operations. For example, machine-dependent page table entries are not created until they are first faulted on, thereby enabling sparse but space-efficient virtual address spaces [Rashid et al. 87]. This approach introduces a certain number of *mapping faults*, which occur every time a virtual page is first accessed by an address space. These faults occur regardless of the cache architecture. In contrast, a *consistency fault* occurs whenever a reference to a virtual address requires a cache consistency state transition that cannot be inferred by some other mapping fault. Consistency faults are the result of the cache being virtually indexed, and should be counted as bookkeeping overhead separate from purge and flush overhead. The table shows that mapping faults remain almost constant across configurations, but that consistency faults drop substantially. In the end, the total bookkeeping overhead for the three benchmarks is a small fraction of the total mapping overhead and insignificant compared to total execution time.

Page flushes need occur no more often in a virtually indexed cache than in a physically indexed one. For configuration "F," the number of page flushes is equal to the number of DMA-read flushes plus the number of pages copied from data space into instruction space. Both of these operations require a flush. The flushes due to copying into instruction space arise because of the interaction between separate instruction and data caches, and the operating system's buffer cache. When a process faults on an instruction page, the file system copies the faulted page from its buffer cache into a page in the faulting process' address space. That copy operation writes into the data cache, yet the page is needed in the instruction cache. The page must therefore be flushed from the data cache before it can be used. The destination virtual page, unless empty in the instruction cache, must also be purged. This problem exists with physically indexed caches as well, because dual caches effectively create an aliasing problem.

[4] We have verified the 720's unusual flush and purge behavior independently of the benchmarks.

The "A" configurations all show no data to instruction space copies because the file system first unmaps the dirty data cache page before mapping it into the faulting address space. The unmap forces an immediate flush which is reflected in the column for total page flushes, rather than in the column for data to instruction space copies.

Virtually indexed caches should support a fast page purge operation. For the benchmarks running under configuration "F," page purges represent the largest cost component of virtually indexed cache management. They occur almost 200,000 times during the three benchmarks. Although some of the purges are necessary during DMA-writes (.8%), and when copying instructions from data space to instruction space (17.5%), most (about 80%) are due to the creation of new mappings when a virtual address is assigned to a random physical page from the kernel's free page list. Some of these purges could be eliminated by reducing the associativity of virtual to physical mappings through the use of multiple free page lists.[5] The architecture, however, should also provide support for efficient cache purges. It should be possible to purge an empty, present, or dirty line, and possibly page, in one cache cycle since no interaction with memory is required. A fast purge would also benefit systems with a physically indexed cache, since purges that cannot be eliminated through reduced associativity are necessary there as well. In all, the total savings for the three benchmarks given a single cycle cache page purge would be about 2.26 seconds (.33%) out of 685.8 seconds.

In summary, the total overhead for virtually indexed cache management across the three benchmarks in configuration "F" is 1.53 seconds (.22%). This is the amount of time spent handling consistency faults (.03 seconds) and purging the data cache for reasons other than DMA (1.50 seconds). An additional 1.48 seconds (.21%) is required for operations that must occur regardless of the cache architecture. This is the amount of time spent flushing and purging the cache to drive DMA devices, and to copy from instruction space to data space.

6 Related work

Several operating systems have been implemented for architectures with virtually indexed caches. As mentioned in Section 2, these systems either disallow aliases, allow constrained aliases, or support full aliases through cache flushing and purging. The most important difference between these other systems and our own is not so much in the optimizations that they support, but the style with which they ensure consistency. Our approach, based on state transitions for cache pages with respect to virtual and physical pages, naturally handles the many kinds of inconsistencies and optimizations that arise in the management of a virtually indexed cache. The state of a cache page depends only on its previous state and the current operation, and the state transitions can be encapsulated entirely within a short code sequence. Moreover, adapting that sequence to alternative architectures is straightforward, given the observations made in Section 3.3. In contrast, previous systems, which do not maintain cache page state in any explicit manner, have dealt with inconsistencies and optimizations on a case-by-case basis. As a result, optimizations requiring global information, such as "does a cache page need to be flushed before it can be used as the destination of a DMA-write," are difficult to implement, and are therefore less likely to be found.

Table 5 highlights some of the functional differences between several operating systems implemented for machines with virtually indexed caches. The CMU system is the one described in this paper. The Utah system is a version of Mach that behaves as the one described in Section 2.5. The Tut project [Chao et al. 90] merged Mach's virtual memory system into HP-UX, HP's version of UNIX. The Apollo system is an implementation of OSF/1 done by the Apollo Systems Division of HP. These four systems have been implemented on HP PA-RISC machines. The Sun system is an implementation of 4.2 BSD for Sun-3 200 series machines [Cheng 87].

The column labels describe the behavior of each system with respect to consistency management. All five systems handle unaligned aliases, although the Sun system limits accesses through unaligned aliases to well-behaved operating system code fragments. Otherwise, aliases must be uncached. The Utah, Apollo, and Sun systems clean the cache whenever the last mapping to a physical page is removed, while the CMU and Tut systems delay the consistency operation until the mapping could be reused (lazy unmap). In Tut, if the new virtual address for a page is the same as the old one (as opposed to aligned) then no purge or flush is required. Otherwise the cache pages corresponding to the old and new virtual pages are removed from the cache. The Utah system makes no attempt to select aligning virtual addresses for multiply mapped pages. Tut does so only for program text pages. Tut, like the CMU system, attempts to choose preparatory mappings so that they align with eventual mappings when preparing a page with copy or zero-fill. In situations where addresses do not align, the CMU kernel eliminates many cache control operations by exploiting the semantics of new mappings (`need_data`) and page preparation (`will_overwrite`).

The systems can also be viewed in terms of the cache consistency states that they maintain for data in the cache, physical memory, and virtual memory. None of the Utah, Apollo or Sun systems maintain a stale state, as evidenced by the fact that they purge or flush the cache whenever a mapping is broken, such as on a write to an aliased page or during the removal of a virtual to

[5] This reduction in associativity is only an optimization, and not a requirement [Chiueh & Katz 92].

System	Handle unaligned aliases	Lazy unmap	Try to align pages	Align page pre-pare	Uses "need_data"	Uses "will_overwrite"
CMU	Yes	Yes	Yes	Yes	Yes	Yes
Utah	Yes	No	No	No	No	No
Tut	Yes	Equal	Text	Yes	No	No
Apollo	Yes	No	Yes	No	No	No
Sun	Memory	No	Yes	No	No	No

Table 5: *Characteristics of several different systems for machines with virtually indexed caches.*

physical mapping. The Sun system appears to maintain only present and empty states for physical pages, although in some cases, such as pageout, it uses the fact that a physical page is dirty to avoid a redundant cache flush. Tut associates state with a virtual address, rather than with a cache page, as evidenced by the fact that only equal, rather than aligned, aliased virtual addresses avoid cache management operations.

7 Conclusions

Virtually indexed write-back caches create consistency problems. We have described these problems informally, and have then defined a model that attacks the consistency problems at their core by ensuring that the memory system never returns a stale value to either devices or the CPU. Our consistency model lends itself to an implementation that relies on the virtual memory system to trap memory accesses that could create or reveal inconsistencies.

The performance of our approach on the HP 9000 Series 700 demonstrates that careful cache management is an important factor in overall system performance. Moreover, an analysis of the operations required to ensure consistency reveals that a virtually indexed cache need not incur significantly more overhead than a physically indexed one.

Our experience with implementing Mach's virtual memory system on a machine with a virtually indexed cache has led us to conclude that there exist no quantitative or qualitative reasons to shy away from such machines, as they offer reduced cycle times with insignificant software cost and complexity.

Acknowledgements

Jerry Huck, Michael Mahon, Bart Sears, and John Wilkes of Hewlett-Packard, Jim Hayes of NeXT, and Tom Mistretta of Apollo provided insight into the problems of virtually indexed cache management. They, along with Jeff Chase, Mike Hibler, Ed Lazowska, Hank Levy, Chris Maeda, Steve Schwab, Dan Stodolsky and Matt Zekauskas provided valuable feedback on earlier drafts of this paper.

References

[Accetta et al. 86] Accetta, M. J., Baron, R. V., Bolosky, W., Golub, D. B., Rashid, R. F., Tevanian, Jr., A., and Young, M. W. Mach: A New Kernel Foundation for Unix Development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–113, July 1986.

[Appel & Li 91] Appel, W. and Li, K. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, April 1991.

[Chao et al. 90] Chao, C., Mackey, M., and Sears, B. Mach on a Virtually Addressed Cache Architecture. In *Proceedings of the First Mach USENIX Workshop*, pages 31–51, October 1990.

[Chase et al. 92] Chase, J. S., Levy, H. M., Baker-Harvey, M., and Lazowska, E. D. How to Use a 64-Bit Virtual Address Space. Department of Computer Science and Engineering Technical Report 92-03-02, University of Washington, February 1992.

[Cheng 87] Cheng, R. Virtual Address Cache in Unix. In *Proceedings of the 1987 Summer Usenix Conference*, pages 217–224, 1987.

[Chiueh & Katz 92] Chiueh, T. and Katz, R. Beating The Address Translation Bottleneck. In *Proceedings of the Fifth Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1992. This issue.

[DEC Alpha 92] DEC Alpha. *Alpha Architecture Technical Summary*. Digital Equipment Corporation, 1992.

[Golub et al. 90] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–95, June 1990.

[Jouppi 88] Jouppi, N. P. Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU. In *Proceedings of the 15th Annual Symposium on Computer Architecture*, pages 281–289, June 1988.

[Knapp & Baer 85] Knapp, V. and Baer, J.-L. Virtually Addressed Caches for Multiprogramming and Multiprocessing Environments. In *Proceedings of the 18th Annual Hawaii International Conference on System Sciences*, pages 477–486, 1985.

[Kohn 89] Kohn, L. Description of the Intel i860 64-bit RISC-based Microprocessor. *IEEE Micro*, 4(9), August 1989.

[Lee 89] Lee, R. B. Precision Architecture. *IEEE Computer*, pages 78–91, January 1989.

[Li 92] Li, K., March 1992. Personal communication.

[Rashid et al. 87] Rashid, R., Tevanian, Jr., A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, April 1987.

[Satyanaranyanyan et al. 85] Satyanaranyanyan, M., Howard, J., Nichols, D., Sidebotham, R., and Spector, A. The ITC Distributed File System: Principles and Design. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 35–50, December 1985.

[Wang et al. 89] Wang, W.-H., Baer, J.-L., and Levy, H. M. Organization and Performance of a Two-level Virtual Real Cache Hierarchy. In *Proceedings of the 16th Annual Symposium on Computer Architecture*, pages 140–148, May 1989.

[Young et al. 87] Young, M., Tevanian, Jr., A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 63–76, November 1987.