March 1992

The effects of virtually addressed caches on virtual memory design and performance

Jon Inouye

Ravindranath Konuru

Jonathan Walpole

Bart Sears

The Effects of Virtually Addressed Caches on Virtual Memory Design and Performance

Jon Inouye
Ravindranath Konuru
Jonathan Walpole
Bart Sears

Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Engineering
19600 N.W. von Neumann Drive
Beaverton, Oregon 97006-1999

Technical Report No. CS/E 92-010

March 1992

The Effects of Virtually Addressed Caches on Virtual Memory Design and Performance*

Jon Inouye, Ravindranath Konuru, and Jonathan Walpole
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
(jinouye,konuru,walpole@cse.ogi.edu)

Bart Sears
Hewlett-Packard Laboratories
(sears@hplabs.hpl.hp.com)

ABSTRACT

Recent times have witnessed rapid advances in microprocessor technology resulting in an order of magnitude performance improvement every few years. These developments in hardware have been paralleled by several prominent trends in operating system design, the most notable being a move towards message-passing micro-kernels. However, operating system performance has not kept pace with that of the underlying hardware. It has become apparent that design changes to enhance processor performance can have adverse effects on operating system performance. This problem arises when the architectural assumptions implicit in an operating system's design are inappropriate for the architectures on which it executes.

This paper examines one specific area in which operating system design assumptions appear to be in conflict with trends in modern processor architecture. We focus on the performance effects of virtually addressed caches on two contemporary operating systems (Mach and Chorus). We present experimental results to illustrate the impact of virtually addressed caches on the performance of primitive virtual memory operations, and higher-level operations, such as interprocess communication, that utilize these primitive operations. The main goal of the paper is to encourage operating system designers to revisit some of the basic architectural assumptions implicit in modern operating system designs.

1 Introduction

There is an increasing awareness that operating system performance has not scaled with hardware performance in recent years [Ousterhout 90]. This decline in relative operating system performance is attributable to conflicting design assumptions in operating systems and the architectures on which they execute [Anderson et al. 91]. In particular, it

*This research is supported by the Hewlett-Packard Company, Chorus Systèmes, and the Oregon Advanced Computing Institute (OACIS).

has become apparent that design changes to enhance processor performance can have adverse effects on operating system performance. This problem arises when the architectural assumptions implicit in an operating system's design are inappropriate for the architectures on which it executes.

Cache design is a key area in which operating system assumptions about computer architecture have become inappropriate. The design of many contemporary operating systems is based on the implicit assumption of a physically addressed cache. However, the prominent recent trend in computer architecture has been a move towards virtually addressed caches in order to decrease cache access times (by allowing parallel translation look-aside buffer (TLB) and cache lookups) and hence support shorter cycle times [Lee 89, Bakoglu et al. 90, MIPS 90]. Since processor speed is also increasing faster than memory access speed, cache effects are becoming more and more important.

A salient feature of these virtually addressed cache architectures is that they often impose the task of maintaining *address translation consistency* on software. Specifically, address aliases must now be resolved by the virtual memory management component of the operating system. This requirement has a major impact on the expense of certain primitive virtual memory operations. For example, mapping multiple virtual addresses to the same physical address can be more costly on architectures with virtually addressed caches.

Contrary to these developments in computer architecture, contemporary operating systems make significantly more use of memory mapping techniques than their predecessors. The prominent recent trend in operating system design is the emergence of micro-kernel operating systems which construct higher-level operating system functionality from multiple server components that interact via message passing [Accetta et al. 86, Armand et al. 89]. Since message passing is central to these systems, the efficiency of message passing is a critical issue in their design. The most common approach to enhancing the performance of message passing is to implement it using memory mapping operations. However, the validity of this approach depends heavily on assumptions about the performance of certain primitive virtual memory operations. This, in turn, depends on the underlying cache architecture.

This paper examines the architectural assumptions implicit in the virtual memory designs of two contemporary operating systems and investigates the suitability of these assumptions for architectures with virtually addressed caches. We determine, experimentally, the effects of virtually addressed caches on the cost of various primitive virtual memory operations as well as higher-level operations, such as interprocess communication (IPC), that utilize these primitive operations. Performance figures are gathered from implementations of Mach and Chorus on Hewlett-Packard Precision Architecture RISC (PA-RISC) workstations.

Section 2 presents the basic characteristics of virtually addressed caches and discusses the problems associated with them. Section 3 explores the architectural assumptions implicit in contemporary virtual memory designs and outlines the implementation of certain primitive virtual memory operations on architectures with virtually addressed caches. The performance of these primitive operations and the implications for higher-level operating system performance, particularly IPC, is investigated in section 4. A brief survey of related work is presented in section 5. Finally, section 6 presents our conclusions.

2 Virtually Addressed Caches

The key distinction between virtually and physically addressed caches is that virtually addressed caches are indexed using part of a virtual address rather than a physical address. Virtually addressed caches offer potentially faster access times by avoiding the delay associated with address translation. While a physically addressed cache requires the TLB translation before it can be accessed, a virtually addressed cache can be accessed in parallel with the TLB reducing the amount of time required for cache access.

Faster cache access times do not come without cost, however. Architectures that use virtually addressed caches must resolve problems associated with homonyms and synonyms [Koldinger et al. 91, Smith 83]. Homonyms are created when a single virtual address is mapped to two or more different physical addresses. This situation can arise on architectures that support private per-process address spaces. Synonyms are created when two or more virtual addresses are mapped to the same physical address. Because contemporary architectures avoid homonyms by either providing a global address space or process tags for each page table entry, we have concentrated on synonyms, henceforth referred to as *address aliases.*

Address aliases are potentially dangerous because they can result in multiple copies of the same data being present in the cache concurrently. Figure 1 illustrates the different ways in which data can be replicated within the PA-RISC's virtually addressed cache. The cache may also be indexed using a physical address when virtual translation is disabled or certain privileged instructions are used. This may create additional address aliases if both virtual and physical addresses are used for the same page. In figure 1, the physical address of **X** is mapped to both the virtual address of **A** and **B**. Three distinct copies of the same cache line containing **X** can appear in the cache as a result: one due to a reference to **A**, one due to a reference to **B**, and a third due to a reference to the physical address of **X**. If any of these values are modified or left in the cache as *stale data*, the contents of the cache will become inconsistent.

In order to ensure that correct values are returned for accesses via the cache, the cache or higher-level software must maintain *address translation consistency.* Address translation consistency is one facet of the more general problem of cache consistency. In the remainder of the paper we will simply use the term *cache consistency.* This should not be confused with multiprocessor cache consistency or split instruction and data cache consistency.

Many cache implementations, including the PA-RISC, store the physical page number as part of the cache tag. This allows aliases falling within the same set to be resolved by hardware. Such aliases are said to be *cache aligned.* Alias resolution between different cache sets is a more difficult problem. Rather than resolving this problem in hardware, most computer architectures pass the cache consistency problem up to the operating system, since only the operating system can generate address aliases in the first place. In the remainder of the paper we investigate the impact of this approach on contemporary operating systems which generate large numbers of address aliases.

3 Virtual Memory Design Assumptions

A characteristic feature of contemporary operating systems is their increased use of virtual memory operations to support functionality that was not previously associated with virtual memory management. Examples include support for IPC, shared memory, and lazy

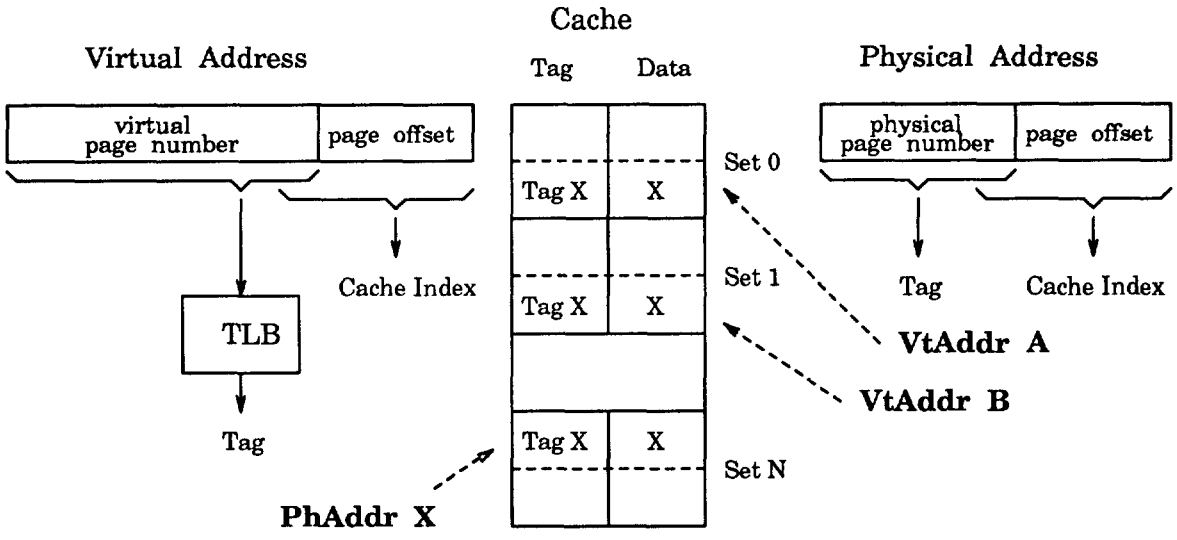

Figure 1: Data replication caused by address aliases on the PA-RISC. The dashed arrows represent the cache index calculated from the address.

copying of data. The primary motivation in using virtual memory operations to support these features is to improve performance. However, the rationale for this approach is based on a number of key assumptions about the relative costs of certain primitive virtual memory operations.

For example, consider the motivation for using memory mapping techniques, such as copy-on-write, to support IPC. The rationale is that by mapping the data of a message, rather than copying it, the operating system can delay, and hopefully avoid altogether, the cost of byte copying. This is an optimistic approach that is makes the assumption that modifying protection information and setting up and maintaining a memory mapping is cheaper than copying the data, and that the possible copying of data at a later time is not significantly more expensive than copying it eagerly.

On architectures with physically addressed caches, the trade-off between the cost of copying and the cost of setting up and maintaining a mapping typically pays off for messages beyond a few K-bytes in size. However, because of the overhead of managing consistency, this is not necessarily the case on architectures with virtually addressed caches.

The standard operating system approach to resolving address aliases is to allow only one instance of an alias to exist within the cache at any point in time. For example, if two virtual addresses map to the same physical address, only one virtual address is allowed to be present in the cache at any moment. This approach is called *pseudo-aliasing* because aliases are not allowed to occur within the machine-dependent layer of the virtual memory system, but appear to be supported at higher layers. The effects of this technique on the implementation of primitive virtual memory operations are illustrated below.

- Creating an initial mapping to a physical page requires the cache lines associated with the page to be flushed if any access has been made to the page using physical

addresses, unless the new mapping causes the page to become equivalently mapped[1].

- Creating a subsequent mapping to a physical page requires the previous mapping to be invalidated, the TLB entry purged, and the cache lines associated with that page to be flushed. Sufficient information is maintained by the virtual memory system to allow further accesses using the invalidated virtual addresses to cause a *pseudo page fault*. The handling of a pseudo page fault involves validating the access and restoring the old mapping by invalidating any existing mapping to that physical page and flushing the cache lines associated with it.

- Invalidating mappings to a physical page requires the cache lines associated with the page to be flushed, or purged, to avoid leaving stale data in the cache. Flushing is used if the consistency of the physical page is important following the unmap operation.

- Modifying a mapping to a physical page requires the cache lines associated with the previous mapping to be flushed, the TLB entry purged, and the new mapping to be established.

Pseudo-aliasing is not the only method for managing the consistency of a virtually addressed cache. However, it is the basis for many other approaches, and the description above illustrates the potential impact of these virtually addressed caches on virtual memory design assumptions. In particular, setting up, maintaining, and destroying mappings to a page involve cache flushes that are not necessary on architectures with physically addressed caches. Depending on the magnitude of these costs, it may be necessary to alter virtual memory designs and/or revisit the assumptions on which they are based. The remainder of this paper attempts to initiate this process by presenting a quantitative and qualitative study of the impacts of virtually addressed caches on primitive virtual memory operations.

4 Virtual Memory Performance on a Virtually Addressed Cache

The performance figures presented in this section were gathered from implementations of the Chorus v3.3 nucleus and Mach 2.0 running on Hewlett-Packard 9000/834 and 9000/835 workstations, respectively. Both systems use the PA-RISC 1.0 processor with virtually addressed instruction and data caches. The write-back, two-way set-associative data cache is 128 K-bytes in size with a 32-byte line size. Both systems use a physical page size of 2 K-bytes. Mach used a logical page size of 2, 4, or 8 K-bytes, and Chorus used a logical page size of 2 K-bytes.

4.1 Low-Level Virtual Memory Primitives

Virtually addressed caches have already been shown to affect the performance of primitive virtual memory operations. Anderson, et al., noted that 536 out of the 559 instructions required to change a page table entry (PTE) for the Intel i860 are concerned with flushing the virtually addressed cache [Anderson et al. 91]. This caused the cost of changing a PTE on the i860 to be an order of magnitude more expensive than changing a PTE on the other platforms studied in the paper. Our experience with Chorus on the PA-RISC reinforces these observations. The primary cost of changing a PTE on the PA-RISC is cache flushing cost. Table 1 shows the cost (in microseconds) of selectively flushing

[1] A page is equivalently mapped if its virtual and physical addresses are cache aligned.

Table 1: Cache Flush Times (in μsecs)

Cache State	Size of Data Block			
	1 Kb	2 Kb	4 Kb	8 Kb
Empty	36	49	75	128
Clean	36	49	75	128
Dirty	80	150	291	579

Table 2: Byte Copying Performance (in μsecs)

Cache State	Number of Bytes Copied			
	1 Kb	2 Kb	4 Kb	8 Kb
Best Case	57	113	224	446
Worst Case	212	408	793	1562

a data block from the data cache on a Hewlett-Packard 9000/834 workstation[2]. The cache state, *Empty*, *Clean*, and *Dirty* indicates whether all cache lines are *not present*, *present but clean*, and *present and dirty* respectively. In measuring *empty* performance, the entire data cache was flushed prior to the selective flush. *Clean* performance was measured by flushing the entire data cache and then reading each line in the block prior to the selective flush. Worst case performance was measured by writing each line in the block prior to the selective flush.

In order to understand the potential impact of these cache flushing costs on PTE operations, we measured the cost of unmapping a 2 K-byte page without flushing the cache. This cost fluctuated between 15 and 20 μsecs during the experiments. This variability is a result of the time it takes to remove the PTE from the page table[3]. Adding the flush cost to the unmap operation increases its cost by 150% in the best case and 1000% in the worst case.

For comparison, table 2 illustrates the cost of byte copying on the same machine for various cache states. In measuring the best case performance, we repeatedly copied data from one buffer to another. This causes both the source and destination data to appear in the cache[4]. To measure worst case performance, both the source and destination data were flushed from the cache. The cache was then filled with dirty lines, by writing each line in a 256 K-byte buffer that did not overlap either the source or destination buffer, before measuring the cost of copying from the source to the destination.

The figures presented above show that, under certain circumstances, the cost of copying a page can be similar to, or even faster than, flushing the page. In particular, copying

[2]We used the HP-UX routine pdcache() which flushes 16 cache lines per loop iteration.

[3]The PA-RISC uses a hashed inverted page table and all PTE's falling into a hash bin are organized as a linked list which must be traversed to find the preceding entry in order to delete a PTE.

[4]The source and destination buffers were chosen so they would not be cache aligned. While this may not be significant on a two-way set-associative cache it can affect performance on a direct mapped cache.

a page that is completely in the cache to a page that is also completely in the cache (best case copy) is faster than flushing the same page from the cache when the lines are dirty (worst case flush). This is because flushing dirty lines requires access to physical memory, whereas cache to cache copying does not. Of course, it is highly likely that the dirty lines left in the cache by a copy will eventually need to be written back to memory.

The performance figures presented above illustrate that a significant portion of the cost of primitive virtual memory operations on architectures with virtually addressed caches can be associated with cache flushing. This cost is not present on physically addressed cache architectures. Furthermore, different cache states can lead to considerable variation in the relative performance of primitive mapping and copying operations.

It is apparent that moving an operating system from an architecture with a physically addressed cache to one with a virtually addressed cache can change the relative costs of various low-level virtual memory operations. In order to understand the significance of these changes, the following subsections study higher-level operating system functionality that makes implicit assumptions about these relative costs. In particular, we focus on the use of virtual memory operations to support IPC and page initialization.

4.2 Chorus IPC Performance

Chorus supports two variants of IPC. The first variant is semantically a *copy* of the message from the sender's address space to the receiver's. The second is semantically a *move*. The main distinction is that the contents of the sender's message buffer are undefined following a send-by-move. The implementation of each of these IPC variants is optimized for good performance, particularly on physically addressed cache architectures. These optimizations are used when message data is page-aligned and multiple-page-sized. For such data, the Chorus IPC-copy is implemented as follows[5]:

1. The sender writes the message to a send buffer in its address space and then calls `ipcSend`.
2. The IPC system allocates a new physical page.
3. The IPC system obtains a virtual address in the kernel address space and uses it to map the newly allocated physical page (the IPC buffer)[6].
4. The IPC system copies the sender's data into the IPC buffer, i.e., it copies from addresses in the sender's address space to addresses in the kernel's address space. The speed of this copy depends on the state of the cache at the time of the copy. Since the IPC buffer has only just been allocated, it is likely that the destination addresses of the copy will not be in the cache. The source of the copy may be in the cache if the sender writes the message immediately before sending.
5. The IPC system checks whether the virtual page of the receive buffer in the receiver's address space is currently mapped to a physical page. If it is, the page is unmapped

[5]Each step is repeated for every page in the message before the next step is started.

[6]Rather than using physical addresses, Chorus maps all physical memory into a special section of the kernel's address space.

and destroyed, resulting in a cache flush. If the page is not shared, then the contents can be purged instead of flushed[7].

6. When the receiver calls `ipcReceive`, the IPC system unloads the page used for the IPC buffer from the kernel address space. This requires a cache flush. Note that all the dirty lines in the page must be written back to memory at some point either before or during this flush operation.

7. The IPC system lazily maps the physical page used for the IPC buffer into the receiver's address space, i.e., the physical page is associated with the receiver's segment cache[8], but the mapping is not set up until the receiver attempts to access the page.

8. The receiver accesses the page containing the receive buffer. This causes a pseudo page fault which forces the mapping of the physical page containing the data into the receiver's address space. This does not require a cache flush because the page was unmapped during step 6.

Note that the IPC operation described above is semantically a copy, but is implemented as a copy from the sender to the kernel, plus a move from the kernel to the receiver. This is an optimization based on the assumption that remapping a page from the kernel to the receiver is cheaper than copying it. We will revisit this assumption a little later.

Chorus IPC-move is implemented as follows:

1. The sender writes the message to a send buffer in its address space and then calls `ipcSend`.

2. The IPC system unloads the page from the sender's address space. This requires a cache flush which may involve a number of dirty cache lines if the sender writes the message immediately before sending.

3. When the receiver calls `ipcReceive`, the IPC system determines whether the receiver has a physical page allocated to the receive buffer. If so, it unmaps and destroys the page. This requires a cache flush (or purge if the page is not shared).

4. The IPC system lazily maps the physical page used for the sender's message buffer into the receiver's address space.

5. As described in the IPC-copy example above, the receiver takes a page fault when it first attempts to access the page. This causes the page to be mapped into the receiver's address space. Note that the contents of the receiver's buffer will not be present in the cache after an IPC-move operation.

The IPC implementation outlined above is based purely on remapping. On physically addressed caches, this is expected to offer better performance than an implementation based on copying. This alone, however, does not explain the use of move rather than copy

[7]The cost of a purge is independent of cache state and faster than the best case flush on the PA-RISC. However, the current implementation of the nucleus always flushes when unloading because it does not differentiate page unloads prior to mapping from unloads prior to destruction.

[8]A *segment* is Chorus' abstraction for storage. A *segment cache* is a collection of physical pages used to cache parts of a segment.

semantics however. Copy-on-write techniques can be used to implement IPC that is semantically a copy without any physical byte-copying (as in Mach). Operating systems, such as Accent, have shown that copy-on-write IPC can outperform IPC based on byte-copying on many physically addressed cache architectures [Fitzgerald and Rashid 86]. The motivation for using move semantics rather than copy-on-write (or copy-on-reference) is that it avoids the overhead of maintaining information about shared physical pages. Manipulation of the data structures (*shadow objects* in Mach [Rashid et al. 88], and *history objects* in Chorus [Abrossimov et al. 89]) for managing this information can be expensive [Nelson and Ousterhout 88].

The key point is that both copy-on-write and move-based IPC are based on the implicit assumption that remapping a page is dramatically cheaper than actually copying it. It has been argued that the difference in performance on physically addressed caches more than offsets the additional complexity of mapping, even for copy-on-write. In Chorus the difference in performance between the copy and move-based IPC variants is expected to be large enough to warrant the use of the less palatable move semantics for some applications.

In order to test the validity of this contention, we implemented a third variant of Chorus IPC. This variant, which would be considered naive on physically addressed cache architectures, is semantically a copy. It is implemented as two copies: one from the sender to the kernel, and the other from the kernel to the receiver. Our implementation involves the following steps:

1. The sender writes the message to a send buffer in its address space and then calls `ipcSend`.
2. The IPC system obtains a virtual address for an IPC buffer in the kernel address space. If no such buffer exists, a new physical page is allocated and mapped to the returned virtual address. A cache flush is unnecessary in either case. In the former case the original mapping is being used and the entire page is about to be written blindly. In the latter case the page was not previously mapped.
3. The IPC system copies the sender's data into the IPC buffer. The speed of this copy depends on the state of the cache at the time of the copy. If an existing IPC buffer was allocated in the previous stage, and if that buffer was recently used, it is possible that the destination of the copy will already be in the cache. Similarly, if the sender has recently written the message to the send buffer it is possible that the source of the copy will also be in the cache, i.e., in the best case this can be a cache to cache copy.
4. When the receiver calls `ipcReceive`, the IPC system copies the message from the IPC buffer to the receive buffer in the receiver's address space.
5. The receiver accesses the page containing the receive buffer. Since the data has been copied rather than mapped into the receive buffer, the IPC operation has a chance to warm the cache. This is only significant if the receiver reads the message soon after receiving it.

Table 3 presents a comparison of the costs of these different IPC variants. The numbers represent the cost of a uni-directional `ipcSend` operation containing various amounts of data.

Table 3: Local Page-Aligned, Page-Sized ipcSend Cost (in μsecs)

Operation	null	1 KB	2 KB	4 KB	8 KB	16 KB
Chorus IPC-copy	337	786	1668	2838	5097	9562
Chorus IPC-move	N/A	N/A	1531	2424	4225	7649
Pure-copy IPC	N/A	N/A	1355	2282	3806	7273

In each experiment two separate Chorus user actors, a client and a server, are created with their own communication ports. During each message exchange, the client writes every word in its buffer, checks the time, and sends the message to the server using an `ipcSend` system call. Once the server receives the message, it reads one word from each page of the message, and then checks the time[9]. The time of reception is then passed back to the client and the server waits for the next message. We repeatedly performed these message exchanges until the average message time did not fluctuate significantly. In all three cases, the sender's and receiver's buffer were not cache aligned, i.e., they did not compete for the same cache sets.

Table 3 shows that for page-aligned, multiple-page-sized messages, the pure-copy IPC approach outperforms both the Chorus IPC-copy and IPC-move. In this experiment, Chorus IPC-move requires two flushes (the sender's buffer and the receiver's old buffer) and a page fault for each page in the message. Flushing the sender's buffer is very expensive since the sender writes every word prior to sending the message. The cost of flushing the receiver's old buffer is a best case flush since no lines are written by the receiver during the experiment.

Each message exchange requires two copies for the pure-copy IPC. A closer examination of the pure-copy experiment revealed that IPC buffers were being recycled. Instead of being passed to the receiver as in the IPC-copy experiment, buffer pages were being reused for the next message exchange. Since the receiver's buffer was constantly being written, it was also present in the cache. Because of this situation, our measurements of the pure-copy performance involved two best case copy costs[10]. From table 1 and table 2 we can see that the sum of a best and worst case cache flush is less than the sum of two best cache copies. The IPC-move performance is worse than the pure-copy performance because of the cost of the fault required to map the message into the receiver's address space.

In order to confirm this, we measured the cost of taking a pseudo page fault in the IPC-move operation and found it to be around 400 μsecs, which is approximately the cost of the worst case copy. If Chorus eagerly mapped the page instead of doing it lazily this page fault could be avoided.[11]. An IPC-move implementation that eagerly mapped the message would outperform the pure-copy IPC in this experiment.

When interpreting these performance figures it is important to note that there are several hidden costs that did not show up in table 3, but may show up in measurements of overall system performance. In the v3.3 kernel, the move semantics may result in a page

[9]Times were taken using the processor's interval timer, a 1/15 μ-second resolution clock register.

[10]There is also the additional cost of setting up a recovery mechanism whenever copies take place between user actors and the kernel.

[11]We learned that later versions of the virtual memory manager do perform eager mapping.

fault when the sender attempts to access the contents of the virtual page previously sent using an IPC-move. This page fault allocates a new physical page to replace the physical page sent to the receiver.

Pure copying also contains some hidden costs involving dirty cache lines. Copying to the IPC buffer and then to the receiver leaves dirty cache lines belonging to the pages representing the IPC buffer and receiver. These cache lines will need to be written back to memory when they are replaced[12]. Note that copying to the receiver will leave the receiver's message data in a "warm" cache state while remapping it will leave the data in a "cold" cache state.

In this example we have shown that operating system designers need to be careful about making implicit assumptions about the cache design of the machine. The Chorus optimization of passing messages by moving pages (remapping) assumes remapping a page is significantly faster than copying it. This assumption ignores the cost of unmapping a page on machines using virtually addressed caches. Our results show that on such machines, pure-copy IPC may perform close enough to Chorus IPC-move to override the performance incentive of using the less desirable move semantics. The important lesson is that the flush overhead of handling a remapping may balance the costs between copying and manipulating PTE's. The next section analyzes the impact of flushes caused by the use of physical addressing.

4.3 Mach pmap Performance

The machine-dependent section of the Mach virtual memory (VM) system is the *pmap module*. The original Mach design tried to make very few assumptions about the VM hardware capabilities of the machine. They needed the capability to initialize a new page in order to satisfy a page fault and needed to guarantee that no other thread in a multiprocessor system would be able to access that page until it was completely mapped. The method used was to leave the page unmapped and to access the page using physical addresses until the page was completely initialized. At that point, the page could be mapped and would be available to all. This solution requires very little hardware support and works well on machines with a physically addressed cache. There is no performance penalty to access a page using its physical address and that access may even go faster than a virtual access as it is not necessary to go through the TLB to get the virtual-to-physical translation. Unfortunately, while the routines which actually initialize and map a page are part of the pmap module, the routines which decide not to map the page until it has been initialized are in the machine independent code, even though the choice to access the page in physical mode has the implicit machine-dependent assumption that this is an efficient way to access a page. In addition to unmapping costs discussed in the previous section, this assumption is incorrect on a machine with a virtually addressed cache.

Before accessing a page using its physical address, it is necessary to flush any virtually indexed lines of that page from the cache. After accessing the page, it is necessary to flush the physically indexed lines of that page before mapping it to a virtual address. It is worth noting that because the copy was done with physical addresses, none of the data has been "prestaged" in the cache by the copy (i.e., when the page is finally used, the cache will

[12]The lines belonging to the IPC buffer may be purged once the message has been sent, but the next copy to that buffer will need to access physical memory for each line in the message.

be "cold"). If a given virtually addressed machine has the capability of mapping a page but marking it as **NO-ACCESS** for everyone including the kernel (except for a couple of special routines which are only used to initialize these pages), it is possible to do the page initialization safely using virtual addresses instead of physical addresses. The examples below show how this works and explains why this is a performance win on a machine with a virtually addressed cache.

Copy-on-write memory is typically marked as *read-only* until a process writes to the page. At that point, a new page is allocated, the old page is copied to the new page and the writing process gets a private writable page. For these examples, we assume that the original page is at least partially in the cache (due to reads or writes before it was marked copy-on-write or due to reads after that point) and that the new page (probably from the free pool) is not in the cache.

Case 1: On a physically addressed cache architecture, the old page is copied to the new page. This involves a copy from a partially warm page to a cold page. After the copy, the new page would be warm, which is an advantage as the user is actively using this page (which is why we faulted).

Case 2: On a virtually addressed cache architecture using the original Mach code, the old page would be flushed, there would be a physical-to-physical copy from the old (now cold) page to the new (cold) page, the physical lines (warm) for the old page would have to be flushed from the cache, and the new (now warm) page would have to be flushed and then mapped. At this point the new page is cold. With a little bit of extra work in the copy routine (machine-dependent), it is possible to look up the virtual address of the original page and do a virtual-to-physical copy which at least saves the double flush of the old page. This would be even faster if the copy routine was given the virtual address of the old page.

Case 3: On a virtually addressed machine which can protect a page as **NO-ACCESS**, it is possible (in the Mach machine independent code) to map the new page (no flush is necessary as the page is known to be cold since it is from the free pool) and then do a copy from a warm virtual address to the new cold virtual address. As in case 1 above, after the copy the new page will be warm.

Case 1 and 3 above have the same relative cost since in both cases, the majority of the work is the copy from a warm page to a cold page. Using the data from section 4.1, it can be shown that case 2 is much more expensive. At best, it requires the flush of the warm physically addressed new page and the cost to copy a cold page to a cold page. Without the optimization of doing a virtual-to-physical copy, it takes a double flush of the old page and a single flush of the new page in addition to the cost of a cold-to-cold copy.

The Tut project at Hewlett-Packard Laboratories measured a 1-3% *overall* system improvement on several different benchmarks when changing from case 2 above to case 3 [Chao et al. 90]. This change involved making some minor modifications to the Mach machine-independent code. As in section 4.2, the important point is that operating system designers need to be careful about their implicit assumptions about the cache design of the machine.

5 Related Work

Both hardware and software solutions to the synonym problem have been proposed. Hardware designers have attempted to avoid making the cache visible to software by providing special hardware to detect aliases [Smith 83]. One hardware mechanism, loosely called a reverse translation buffer (RTB), allows aliases to be detected within the cache. On a cache miss, the TLB produces a physical address and the subsequent request to main memory is made in parallel with an RTB search. If the RTB produces a virtual address that is already present in another cache line, the line must be renamed and moved to the new location. As alluded to in section 2, an RTB implementation can be very simple if aliases can be forced to fall within the same set, i.e., if aliases are cache aligned. It is possible to force aliases to be cache aligned by using only the page offset bits to index the cache. This scheme limits the cache size to the logical page size multiplied by the cache set-associativity. The drawback of this solution is that it results in small cache sizes unless a high degree of associativity is supported[13].

Commercial operating systems have avoided problems with virtually addressed caches by limiting alias generation. Most versions of Hewlett-Packard's HP-UX operating system do not support copy-on-write memory or the UNIX `mmap()` system call[14]. Shared text is implemented by sharing the same global virtual address segment instead of using memory mapping. System V shared memory is supported only if each shared memory segment is used at the same virtual address by each process using it [Clegg et al. 86].

The Tut project at Hewlett-Packard Laboratories ported Mach 2.0 to PA-RISC and compared simple minded cache flushes to prevent aliases with both using read-only aliasing and the shared global address space techniques present in HP-UX [Chao et al. 90]. (Read-only aliasing allows virtual address aliasing of read-only memory to exist at the hardware level. Since the memory is shared read-only, there is no consistency problem as long as all aliases are purged from the cache if the memory becomes writable again.) Using read-only aliasing significantly improved the overall system performance (6-10%). Using both read-only aliasing and the shared global address space gave an additional 2-3%. This was on a machine with a large cache and a large number of TLB entries. It is expected that the addition of shared global address space techniques will give even better performance on machines with small caches and small TLBs as the replicated cache lines and TLB entries needed for pure read-only aliasing will have a bigger negative impact.

Several modifications to SunOS Release 3.2 were necessary to manage the virtually addressed cache in the Sun-3 series 200 workstation [Cheng 87]. Three methods were used to maintain cache consistency: flushing the page on any address mapping invalidation, cache alignment points (128K), and non-cacheable pages. Measurements showed that cache flushing costs were smaller than expected. The worst benchmark result showed that 3.0% of the total time was spent flushing the cache. In this benchmark, increased use of the Direct Virtual Memory Access (DVMA) operation caused many page flushes. The DVMA operation creates a new address mapping (to the DVMA region) for a page which already contains a virtual address. This requires one cache flush when establishing the new DVMA mapping, and another upon completion.

The Sprite operating system used a combination of copy-on-write (COW) and copy-on-

[13] Associativity can be very expensive in integrated primary caches.

[14] This functionality is supported by certain versions of HP-UX 8.0.

reference (COR) during process creation [Nelson and Ousterhout 88]. In this modification of a pure COW scheme, a fork operation causes the pages of the parent to be marked copy-on-write, and the child's pages to be marked copy-on-reference. One of the reasons behind the choice of COW-COR over a pure COW mechanism was the existence of a virtually addressed cache. Sprite was targeted to run on the Spur workstation, a virtually addressed cache architecture [Wood et al. 86]. Because the SPUR's virtually addressed cache stored protection information in cache lines, any change in a page's protection required the contents of the page to be flushed from the cache. This cost associated with a protection change is not as expensive on the PA-RISC since protection information is stored in the TLB. Changing the protection of a page requires purging a TLB entry instead of cache lines.

We have concentrated on address translation consistency in single processor environments. However, other researchers have already begun to study the problem of multiprocessor virtually addressed cache consistency [Cheriton et al. 86, Goodman 87].

6 Conclusion

In this paper we have examined the effect of virtually addressed caches on primitive virtual memory operations. In addition, we have shown how the cost of these primitive operations can affect design decisions in higher order operations in Mach and Chorus. Both Chorus' IPC and Mach's pmap were designed assuming physically addressed cache architectures.

In the case of Chorus IPC, we have shown the effect that virtually addressed caches have on page remapping performance. Unmapping a page requires all its virtually addressed lines to be flushed from the cache. The cost of an unmap operation can cause a simple pure-copy implementation to perform comparably with a Chorus IPC-move operation.

The Mach pmap experiment showed that by ignoring the underlying cache architecture, virtual memory designs can suffer unnecessary performance degradation. By making minor modifications in the machine-independent sections to allow for machines with different cache designs, it may be possible, as it was in this experiment, to run both on machines with physically addressed caches and machines with virtually addressed caches with no performance penalty.

If operating systems continue to receive the burden of providing address translation consistency, operating system designers will need to re-evaluate their design decisions regarding the cost of virtual memory operations. We have shown that the failure to take into account cache designs can have significant effects on operating system design and performance.

References

[Abrossimov et al. 89] Abrossimov, V., Rozier, M., and Shapiro, M. (1989). Generic Virtual Memory Management for Operating System Kernels. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles.*

[Accetta et al. 86] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. (1986). Mach: A New Kernel Foundation For UNIX Development. In *Proceedings of the 1986 Summer USENIX Conference*, pages 93–112, Atlanta, Georgia.

[Anderson et al. 91] Anderson, T. E., Levy, H. M., Bershad, B. N., and Lazowska, E. D. (1991). The Interaction of Architecture and Operating System Design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, Santa Clara, CA.

[Armand et al. 89] Armand, F., Gien, M., Herrmann, F., and Rozier, M. (1989). Revolution 89 or "Distributing UNIX Brings it Back to its Original Virtues". In *Proceedings of the Workshop on Experiences with Building Distributed and Multiprocessor Systems.* Also published as technical report CS/TR-89-36.

[Bakoglu et al. 90] Bakoglu, H. B., Grohoski, G. F., and Montoye, R. K. (1990). The IBM RISC System/6000 processor: Hardware Overview. *IBM Journal of Research and Development*, 34(1):12–22.

[Chao et al. 90] Chao, C., Mackey, M., and Sears, B. (1990). Mach on a virtually addressed cache architecture. In *Proceedings of the USENIX Mach Workshop.*

[Cheng 87] Cheng, R. (1987). Virtual Address Cache in UNIX. In *Proceedings of the Summer 1987 USENIX Technical Conference and Exhibition*, pages 217–224.

[Cheriton et al. 86] Cheriton, D., Slavenburg, G. A., and Boyle, P. (1986). Software-Controlled Caches in the VMP Multiprocessor. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 366–374.

[Clegg et al. 86] Clegg, F. W., Ho, G. S.-F., Kusmer, S. R., and Sontag, J. R. (1986). The HP-UX Operating System on HP Precision Architecture Computers. *Hewlett-Packard Journal*, 37(12):4–22.

[Fitzgerald and Rashid 86] Fitzgerald, R. and Rashid, R. (1986). The Integration of Virtual Memory Management and Interprocess Communication in Accent. *ACM Transactions on Computer Systems*, 4(2):147–177.

[Goodman 87] Goodman, J. R. (1987). Coherency For Multiprocessor Virtual Address Caches. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 72–81.

[Koldinger et al. 91] Koldinger, E. J., Levy, H. M., Chase, J. S., and Eggers, S. J. (1991). The Protection Lookaside Buffer: Efficient Protection for Single-Address Space Computers. Technical Report 91-11-05, University of Washington, Department of Computer Science & Engineering. In preparation.

[Lee 89] Lee, R. B. (1989). Precision Architecture. *IEEE Computer*, 22(1):78–91.

[MIPS 90] MIPS (1990). *MIPS R4000 Preliminary Users Guide.* MIPS Computer Systems Inc. Preliminary Revision 2.0.

[Nelson and Ousterhout 88] Nelson, M. and Ousterhout, J. (1988). Copy-on-Write For Sprite. In *Proceedings of the 1988 Summer USENIX Conference*, pages 187–201.

[Ousterhout 90] Ousterhout, J. K. (1990). Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, Anaheim, CA.

[Rashid et al. 88] Rashid, R., Jr., A. T., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W. J., and Chew, J. (1988). Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions on Computers*, 37(8):896–908.

[Smith 83] Smith, A. J. (1983). Cache memories. *ACM Computing Surveys*, 14(3):473–530.

[Wood et al. 86] Wood, D. A., Eggers, S. J., Gibson, G., Hill, M. D., Pendleton, J. M., Richie, S. A., Taylor, G. S., Katz, R. H., and Patterson, D. A. (1986). An In-Cache Address Translation Mechanism. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 358–365.