Fast Byte Copying: A Re-Evaluation of the Opportunities for Optimization

Jon Inouye
Jonathan Walpole
Ke Zhang

(jinouye,walpole,zhang@cse.ogi.edu)

Technical Report CSE-95-010

Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology

June 1995

Abstract

High-performance byte copying is important for many operating systems because it is the principle method used for transferring data between kernel and user protection domains. For example, byte copying is commonly used for transferring data from kernel buffers to user buffers during file system `read` and IPC `recv` calls and to kernel buffers from user buffers during `write` and `send` calls. Because of its impact on overall system performance, commercial operating systems tend to employ many specialized byte copy routines, each one optimized for a different circumstance.

This paper revisits the opportunities for optimizing byte copy performance by discussing a series of experiments run under HP-UX 9.03 on a range of Hewlett-Packard PA-RISC processors. First, we compare the performance improvements that result from several existing byte copy optimizations. Then we show that byte copy performance is dominated by cache effects that arise when source and target addresses overlap. Finally, we discuss the opportunities and difficulties associated with choosing appropriate source and target addresses to optimize byte copy performance.

1 Introduction

The motivation for this paper developed, almost by accident, as a result of our quest to understand some unexpected fluctuations in file system `read` performance in the Synthetix [2] and HP-UX [3] operating systems running on Hewlett-Packard PA-RISC processors. As part of our experiment we were attempting to measure the performance of the *fast path* through the `read` system call in both systems. The fast path represents the case in which the required data already resides in the file system buffer cache, i.e.., no disk I/O is involved.

We were surprised to observe up to an order of magnitude variation in execution cost for different runs of the same code, despite the fact that we were careful to ensure that all the necessary data was resident in memory. Since a significant component of the `read` cost (for large reads which hit in the buffer cache) is the cost of byte copying between kernel and user buffers, we suspected that the observed variations in read performance might be a reflection of performance variations in the underlying byte copy routine. This suspicion turned out to be valid. In other words, on this architecture performance can vary by as much as an order of magnitude between runs of the same code which simply copies a fixed number of bytes from a kernel buffer to a user buffer.

We were already aware of the fact that there were a multitude of different byte copy routines in HP-UX 9.0, each one optimized for a particular situation. However, the performance variations we were observing appeared to be unrelated to the choice of underlying byte copy implementations. In addition, we suspected that the performance variation we were observing not only dominated other factors relating to byte copy performance, but also presented a new opportunity for optimization: if some runs execute an order of magnitude faster than others, the operating system code should attempt to ensure that the fast cases occur more frequently, or conversely, it should avoid the slow cases.

In this paper we study, experimentally, the performance of various byte copy optimizations used by HP-UX and investigate some new opportunities for optimization. The remainder of this paper is organized as follows. Section 2 describes the support for efficient byte copying provided by the PA-RISC architecture and the HP-UX operating system. This section also contains a description of the implementation-dependent features of PA-RISC implementations that caused the erratic `read` performance. Section 3 describes the experiments we ran to measure the performance impact of different byte copy implementations and section 4 presents and analyzes the results. We conclude with a discussion about our experiences.

2 Byte Copying on HP Series 700 Workstations

2.1 Architectural Support for Efficient Byte Copying

The Hewlett-Packard PA-RISC architecture [14, 8] has several features to improve the performance of byte copying.

1. Instructions for unaligned byte copies. If the source and destination addresses are not aligned[1] with each other, then it is impossible to transfer the data using word[2] loads and stores. The PA-RISC provides two instructions to assist in copying unaligned data. Once source data is loaded into registers, `vshd` (variable shift double) instructions can be used to align the data to the destination address. The `stbys` instruction can then be used to store a variable number

[1]For the purposes of this paper, alignment indicates the least significant 2 bits of the addresses are identical.

[2]In this paper, a word is four bytes, a double word is eight bytes, and a quad word is 16 bytes in length.

of bytes from a register to memory. These instructions allow unaligned copy routines to avoid transferring data at the byte granularity.

2. Cache hints. Hints are used to avoid reading in a cache line on a store miss. A store miss can generate up to three bus transactions: (1) writing back the data in the cache (if dirty), (2) reading in the designated cache line from memory, and (3) writing the designated line back to memory. When software writes the *entire* cache line, hints can be used to avoid the read from memory. Since the contents of the entire line are being modified, there is no need to preserve the old contents of the cache line. Note that the third bus transaction is rarely measured by copy benchmarks since most PA-RISC implementations use a write-back cache. Hence, this cost is passed on to other routines in the same manner that the first cost is inherited by the copy benchmark.

3. The floating point (FP) co-processor has 64-bit registers and supports double word loads and stores. Using the FP registers may increase the amount of data transferred by each load and store.[3]

2.2 `copyout` in HP-UX

The HP-UX operating system [3] uses the routine `copyout` to copy data from the kernel's address space to the user's address space. `copyout` must ensure the target of the copy is a valid user virtual address and resident in physical memory. In addition, it must also set up a recovery environment to ensure that any memory access faults occurring during the copy are assigned to the user code and not to the kernel. Finally, `copyout` calls `ulbcopy` to transfer the data to the user's buffer. While it is possible to specialize all three of these operations, this work focuses on the last operation.

The HP-UX 9.0 version of `ulbcopy` is already very specialized. The routine examines the source and destination addresses along with the number of bytes to be copied to decide the proper approach to take. For copying small amounts of data, a byte-by-byte routine is used. If the source and destination address are not aligned with each other then `vshd` and `stbys` instructions are used in conjunction with standard word loads and stores. If the source and destination are aligned, `stbys` is used to get to the nearest word boundary and then a word-aligned copy is performed. When the source and destination are aligned on 64 byte boundaries, one of three possible code loops is used. During kernel initialization, HP-UX measures the performance of the three loop implementations and modifies a branch target address to use the loop implementation that achieves the best performance. The first loop uses the PA-RISC's general registers to hold the data being copied. The loop performs four word loads followed by four word stores. The second loop uses floating point co-processor registers and instructions. This loop performs four double word loads followed by four double word stores. The third loop uses a slight variation of the second. Rather than using double load stores, it uses an implementation-specific instruction, `fstqs`, to store a quad word. Though the HP-UX assembler accepts this instruction, not all PA-RISC implementations support it.[4] In order to perform this kind of low-level code specialization, operating system programmers must be aware of the various hardware mechanisms provided by the implementation.

[3]All the machines we tested support a 64-bit data path between the processor and the cache. This wide data path allowed FP load/store instructions to transfer twice as much data as conventional load/store instructions when hitting the cache.

[4]This instruction is not supported by the PA7100 and follow-on processors.

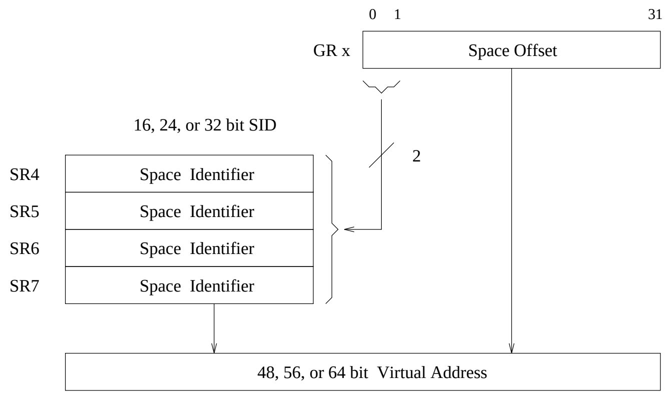

Figure 1: Virtual Address Calculation

2.3 Memory Effects and PA-RISC Cache Implementations

In order to determine the memory effects on byte copy performance, we needed to support a well-defined memory environment for our experiments. This required an understanding of the PA-RISC memory management unit architecture and the implementation features of the cache. The HP 9000 series 700 workstations use a direct-mapped, virtually indexed, physically tagged cache. For control purposes, we wanted to know where data was being placed in the cache. This process requires converting a 32-bit UNIX address, referred to as a *short pointer*, into a cache address. We describe this in two steps: obtaining a virtual address given a short pointer, then obtaining the cache address from the virtual address.

The PA-RISC supports a segmented 64-bit virtual address space. The 2^{64} address space is partitioned into 2^{32} *spaces*. Each space is identified by a *space identifier (SID)*. The number of SIDs supported depends on the level of the PA-RISC implementation. The workstations we used for our experiments all support level 1 implementations that have 2^{16} SIDs; so the implementation only supports a 48-bit virtual address space. A 32-bit short pointer is converted into a 64-bit address in the following manner: the most significant two bits are used to select one of four space registers. The contents of the selected space register are concatenated with the 32-bit short pointer to form a virtual address. Figure 1 illustrates this conversion.

Since the cache is virtually indexed, the cache address is generated using the virtual rather than the physical address. Figure 2 shows how the cache address is calculated from the virtual address. Some of the least significant bits of the SID are hashed with the middle bits of the space offset. These are concatenated with the lower bits of the space offset to form the cache address.[5] Given the hash index function, the cache address can be calculated using the space offset and the space identifier. This allowed us to choose appropriate pairs of virtual addresses to guarantee that source and destination buffers would not overlap in the cache.

[5]It is possible to disable the hash function using special diagnostic instructions. Disabling the hash function removes the space identifier bits in the cache address calculation.

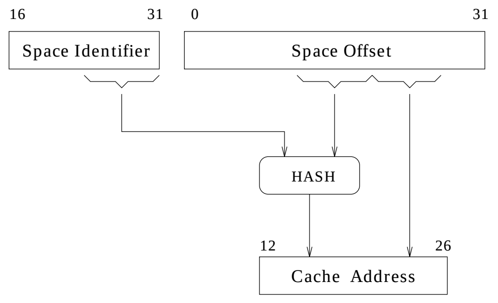

Figure 2: HP 700 Cache Index Function

3 Experiments

We designed a set of experiments to analyze the effects of data alignment and bulk copy optimizations, and how they are influenced by cache temperature.

3.1 Data Alignment

We used the standard HP-UX 9.03 libc routine `memcpy` to test the impact of non-aligned data copies. An analysis of `memcpy` shows that it performs the equivalent non-aligned optimizations performed by `ulbcopy`, but never uses the floating point registers to transfer data. The experiment uses two 4 KB buffers, known not to conflict with each other in the cache. The contents of the first buffer are transferred to the second. Starting with a source and destination address that are page-aligned, the destination buffer is moved one byte at a time while timing the performance of the copy.

3.2 Bulk Copy Optimizations

We implemented four different versions of 32-byte-aligned copy routines. Three of them are similar in nature to the loops used in the HP-UX `ulbcopy` routine except they deal only with 32-byte-aligned addresses and sizes of 32 byte multiples. Table 1 describes the various copy implementations. As a base for our measurements, we used the C library routine `memcpy`. Note that the optimized loop in `memcpy` performs only a four word load followed by four word stores.

Table 1: Data Copy Experiments

Routine	Description
memcpy	HP-UX 9.03 library call
8Wld-8Wst	ASM function whose main loop performs 8 word loads to registers followed by 8 word stores.
8Wld-8WstH	Identical to 8Wld-8Wst but stores use cache hints
4Dld-4Dst	ASM function whose main loop performs 4 double word loads to registers followed by 4 double word stores.
4Dld-2Qst	ASM function whose main loop performs 4 double word loads to registers followed by 2 quad word stores.

3.3 Cache Environment

In addition to the various implementations, we wanted to measure the effect the memory hierarchy has on copy performance. In order to measure the memory effect, we needed to control the temperature of the cache. Cache temperature is an indication of the percentage of memory references that hit the cache. For the experiments outlined above, both the source and destination addresses were placed in one of three uniform temperature states.

Dirty: The memory region was considered *dirty* if the contents of the region were not present in the cache and the cache lines consigned to that region were modified, i.e., write back required.

Clean: The memory region was considered *clean* if the contents of the region were not present in the cache and the cache lines assigned to that region were clean, i.e., no write back required.

Hot: The memory region was considered *hot* if the contents of the region were present in the cache.

4 Results

The experiments were run on three different HP platforms: the HP 9000/720, 9000/755, and 9000/712-80 "Snakes" workstations. The HP 720 uses an early PA-RISC 1.1 implementation CPU with a clock speed of 50 MHz [7]. The HP 720 has split instruction and data caches. The data cache is direct-mapped, 256 KB in size, with a line size of 32 bytes. The HP 755 uses the PA7100 CPU running at 99 MHz [4]. The 755's data cache is also direct-mapped, 256 KB in size, with a line size of 32 bytes. The 712-80 uses a PA7100LC CPU running at 80 MHz [12]. The 712 has a combined data and instruction cache of 256 KB. The cache is direct-mapped with a line size of 32 bytes. The DRAM memory speed of the machines was not specified in the documentation we had available. All machines were running HP-UX 9.03.

All experiments were compiled using each platform's C compiler with no optimization. The level of optimization should not affect the results since all the measured routines are either in the C library (libc) or written in PA-RISC assembler.[6] The experiments were run in user mode using real-time priority to avoid context switches and interference from other user processes. Each run performed twenty copies in a tight loop. Programs were run ten times and the lowest value was recorded. Cycle counts were measured using the PA-RISC interval timer, a register-based counter that increments at an implementation-dependent rate. For the PA-RISC implementations used in our experiments, all rates were based on the clock speed of the CPU. Appendix A contains the complete results of the experiments but the following sections attempt to present the results in a more enlightening manner.

4.1 Alignment Results

Figure 3 shows the effects of source and destination alignment for the HP-UX `memcpy` routine. Having the source and destination word aligned is important, more so on the older architectures. Source and destination alignment have a smaller effect on the 712 since its processor has two integer units. This allows it to execute concurrently `vshd` instructions with stores.

[6]In fact, using the +O3 level of optimization removed the cache temperature control loops since they generated memory references but did not perform any noticeable computation.

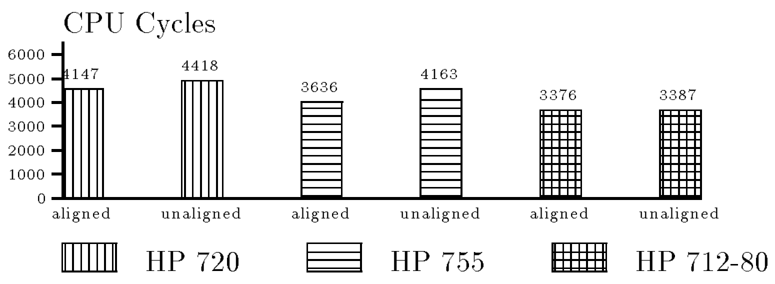

Figure 3: Aligned vs. unaligned Copies (hot-to-hot)

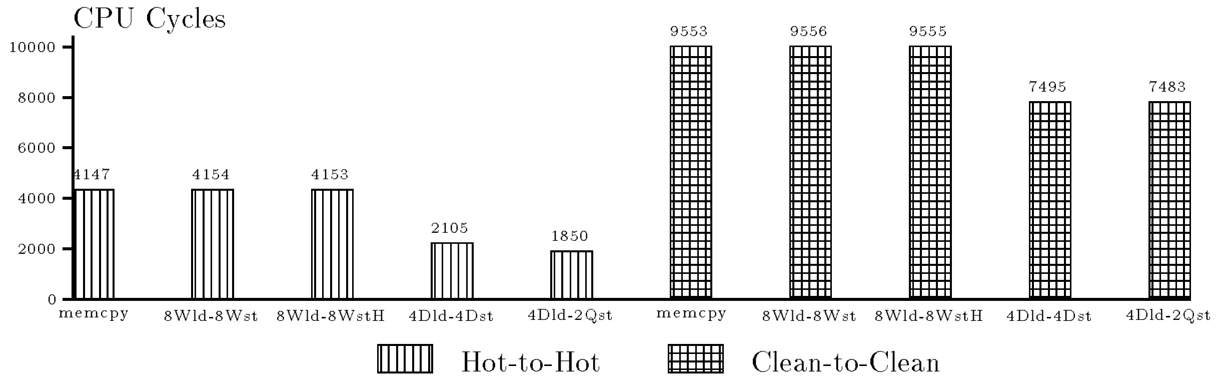

Figure 4: HP 720 4 KB Copy Results

4.2 Bulk Copy Results

Figure 4 shows the results of our experiments run on the HP 720 for two cache environments. The first cache environment measures cache-to-cache copy speed while the second environment tests memory-to-memory speed. Of all the byte copy implementations, the ones using the floating point registers have the best performance. As expected, the temperature of the cache has more impact on performance than the implementation of the various copy routines. The cache-to-cache copies are roughly twice the speed of the memory-to-memory copies. On the other hand, cache temperature emphasizes the difference between the various copy implementations. The ratio between the best and worst copy is 0.45 in a 'hot' cache, but nearly doubles to 0.78 in a 'clean' cache environment.

Figure 5 illustrates the performance results of some experiments run on the HP 755. The relative performance of the copy implementations was similar to that of the 720 with the exception of the 4Dld-2Qst implementation. This result is not shown in the figure because it is nearly two orders of magnitude larger than any other. As mentioned previously, the floating point quad word store is not part of the PA-RISC instruction set. The HP 755 floating point unit does not support the quad word store so that instruction is executed by the HP-UX floating point emulation library. The severe increase in cost is caused by handling a FP emulation trap, executing within the emulation library, and returning from the exception.

The growing divergence between cache performance and memory performance can be seen when comparing the 755's numbers to the 720's. We expected the difference between cache and memory performance to grow as the speed of the processor increases. Figure 6 shows this effect. Using 4Dld-4Dst as an example, the cost of a memory to memory copy (with no write back) on the 50 MHz HP 720 is 350% the cost of the cache to cache copy. This becomes 660% on the 100 MHz HP 755! It is unusual that the cold copy performance of the HP 712 exceeds that of the HP 755.

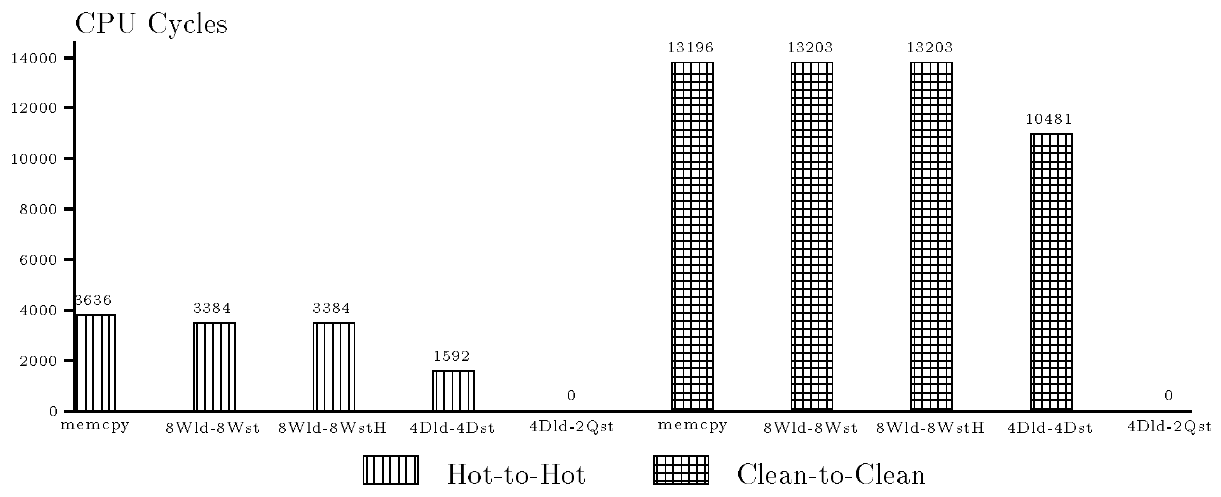

Figure 5: HP 755 4 KB Copy Results

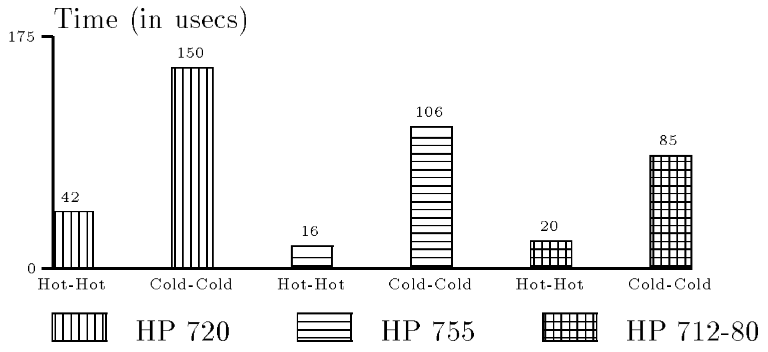

Figure 6: Comparing Cache Temperature Effects (4Dld-4Dst)

One reason for this may be that the PA7100LC processor handles memory to memory copies better than the older PA7100 due to the additional features that mask the cost of cache misses [12].

We had hoped that the cache hints would improve the performance of the memory copy operation since it would avoid reading the line from memory. However, as figure 7 shows, cache hints have no effect on store performance to memory on any of the architectures. While the HP 720 CPU does not implement cache hints, the 755 and the 712 CPUs should implement them. One possible reason for the non-effect of cache hints could be that the benchmarks were run in user mode. Due to security reasons, implementations of the PA-RISC 1.1 architecture must zero the cache line before writing to the line. On the 755 and 712, the implementations may simply ignore cache hints while in user mode.

5 Discussion

While there is a measurable difference between aligned and unaligned copies, alignment does not have much effect on `memcpy` performance. Alignment has more of an effect on `copyout` performance since the floating point unit can only be used when source and designation are aligned on 32-byte boundaries. The differences between the copy implementations is noticeable but performance is dominated by cache temperature. Any differences from the various optimizations were reduced

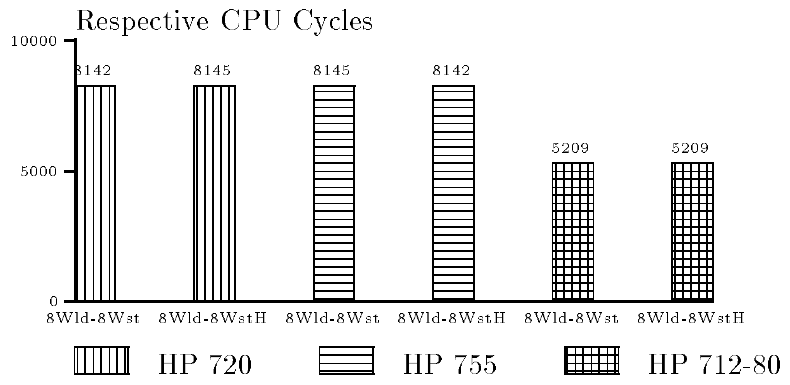

Figure 7: Effect of Cache Hints on Store (Hot to Cold) Performance

when copies required more memory traffic. This result indicates that operating systems should investigate means to reduce the amount of memory traffic incurred while copying.

On a direct-mapped virtually indexed cache, memory traffic can increase when the source and destination buffer occupy the same cache addresses. This situation generates cache conflict misses which lower the temperature of the cache and degrade copy performance. Cache conflict misses can be removed by assuring that the source and destination addresses of the copies are placed in virtual memory regions that do not overlap in the cache. The operating system has little choice of the user address since most UNIX system calls transfer data to a user-specified buffer. For example, the `read` interface transfers data to an address specified as an input parameter to the system call. Given this situation, the kernel can only determine the address of the kernel buffer.

Selecting the appropriate non-conflicting kernel address is complicated. Removing an existing conflict requires the kernel to change the virtual address of its buffer. One way is to copy the buffer to a different region but this may pollute the cache and may be more expensive than simply performing a single conflicting copy. Re-mapping the kernel buffer to a different virtual address is another option. While it avoids physical copying, re-mapping operations may require cache flushes to preserve address translation consistency [9]. In either case, removing the conflict can be expensive and only worthwhile if the conflict is persistent. Since `read` system calls tend to sequentially access a file there is little temporal locality in the kernel buffers so conflicts are rarely persistent.

Rather than resolve conflicts once they are detected, the operating system may attempt to avoid creating conflicts through clever buffer cache management. If application programs reuse user buffers, than the kernel can avoid future conflicts by placing read-ahead buffers at "safe" virtual addresses that do not overlap the user buffer in the cache. This policy will not avoid conflicts if application programs vary the buffer addresses they transfer data into. Choosing non-conflicting addresses is complicated by the cache index function.[7]

Another possible optimization is to choose buffer cache addresses for a particular file so all the buffers map to the same region of the processor cache which does not conflict with the user buffer. Sequentially reading a file would continuously use that small region of the cache, and not flush the entire cache for large reads.

[7]This is rather ironic since the index function was originally implemented to avoid cache conflicts measured during transaction processing benchmarks.

6 Related Work

In this paper, we have concentrated on physical copying (duplicating data in physical memory) but other copying techniques avoid this problem by using logical copying techniques, such as copy-on-write (COW). COW techniques duplicate data in the virtual memory domain but avoid duplicating data in the physical memory domain unless data integrity needs to be preserved. In the COW scheme, physical memory data may be shared by multiple virtual memory regions as long as it is not modified. Data integrity is preserved by performing a physical memory copy when writes are detected. Many research projects have implemented techniques that perform data movement *without* duplicating physical memory. A couple of examples are fbufs and Accent. Fbufs move data from network devices to user space and back again with a minimum of both virtual and physical memory duplication [5]. Accent used copy-on-write memory for high-performance local IPC [6]. Many contemporary virtual memory designs support the notion of copy-on-write or copy-on-reference memory to support both local bulk data movement and efficient copying of address spaces using virtual memory (VM) techniques that manipulate the TLB and VM data structures [16, 1, 10]. VM techniques cannot be used in all cases, such as when the data being copied is smaller than a page or not page-aligned with the user's buffer address. This limits the number of cases where `read` can benefit from COW techniques. In addition to these limitations, VM techniques have their own overhead and complexity associated with their use.

There has been some work at removing persistent conflicts using dynamic page remapping policies [15]. These policies focus on conflicts within an application program running on a processor with a direct-mapped physically addressed cache. Possible conflicts are detected using coloring[8] information and hardware mechanisms such as cache miss counters. Once conflicts are detected they are removed by *recoloring* operations, which change the color of a virtual page by moving data from one physical page to another and adjusting the address translation accordingly. Diagnosing performance-degrading conflicts is the most difficult part of this process. Only persistent conflicts that cost more than recoloring operations should be removed. Cache miss counters help to identify times when large number of cache misses are occurring, but fail to distinguish between *capacity* and *conflict* misses. Recoloring a virtually addressed cache is more difficult because the page color is dependent on the virtual, rather than the physical, address.

There will be more architectural support for byte copying in future Hewlett-Packard workstations. The new PA7200 chip has an internal 64 entry, fully associative cache in addition to the traditional direct-mapped off-chip cache [13]. On each memory reference, the tag is compared to 65 entries (the 64 fully associative entries and the single entry returned by the off-chip cache). A cache line sent by memory to satisfy a miss is placed in the internal cache. The internal cache line selected for replacement is sent to the off-chip cache unless a "spatial locality hint" is given in which case it is returned to memory. This allows large byte copies to be performed without polluting the larger off-chip cache. Now the operating system must make another choice while copying. Should it leave the data in the off-chip cache or return it to memory? This decision depends on what the user plans to do with the data! If the data is going to be operated on immediately, then it might be better to leave it in the off-chip cache to avoid future cache misses. If the data has low temporal locality, then it is preferable to return it to main memory and avoid polluting the off-chip cache. Unfortunately, the operating system may not have sufficient information to make the right decision. Operating system can provide applications with meta-interfaces that allow programmers to specify the expected behavior of their programs [11]. For example, BSD UNIX introduced the `madvise` system call as a meta-interface for application programs to inform the operating system

[8]Cache is divided into separate page-size colors. Conflicts may occur when two pages have the same color.

of a memory region's paging behavior.

7 Summary

This paper examines the various optimizations performed by the HP-UX `ulbcopy` routine and show that, while useful, they do not significantly affect performance compared to cache temperature. The temperature of the cache can be reduced by cache conflict misses when the source and destination overlap each other in a direct-mapped cache. While it is important for the operating system to make use of the hardware features of processor implementations, it is also important for the operating system to reduce the number of conflict misses generated by copies.

In the case of the `read` system call, this can be done using a variety of methods. Sophisticated buffer cache management may avoid conflicts by inferring user buffer addresses based on past behavior and choose non-conflicting buffer cache addresses. However, since the operating system has to "guess" about user addresses this method is not 100% effective. Operating systems can also provide meta-interfaces to allow applications to specify their behavior. While meta-interfaces provide more accurate information, they place more of a burden on application programs and require the system call interface to be extended.

The architecture world has addressed this problem by providing higher associativity and cache hints to reduce cache pollution by data with low temporal locality. But this hardware poses essentially the same problem. The operating system must still determine the best policy to use to sustain a low number of cache conflict misses. The hardware has provided a new mechanism, but the operating system still has to determine the policy. This provides operating system researchers with an opportunity to improve byte copy performance by minimizing the amount of memory traffic, both immediate and future, in the system.

8 Acknowledgements

Eric Hamilton and Jerry Huck of Hewlett-Packard helped enlighten us about many of the cache-related features provided by the PA-RISC architecture and its implementations.

References

[1] Vadim Abrossimov, Marc Rozier, and Marc Shapiro. Generic Virtual Memory Management for Operating System Kernels. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 3-6 1989. Also published as technical report CS/TR-89-18.

[2] Andrew Black, Charles Consel, Calton Pu, Jonathan Walpole, Crispin Cowan, Tito Autrey, Jon Inouye, Lakshmi Kethana, and Ke Zhang. Dream and Reality: Incremental Specializatoin in a Commercial Operating System. Technical Report CSE-95-001, Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, March 1995.

[3] Frederick W. Clegg, Gary Shiu-Fan Ho, Steven R. Kusmer, and John R. Sontag. The HP-UX Operating System on HP Precision Architecture Computers. *Hewlett-Packard Journal*, 37(12):4–22, December 1986.

[4] Eric DeLano, Will Walker, and Mark Forsyth. A High Speed Superscalar PA-RISC Processor. In *COMPCON 92*, pages 116–121, San Francisco, CA, February 24-28 1992.

[5] Peter Druschel and Larry L. Peterson. Fbufs: A High Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 189–202, Asheville, North Carolina, December 1993.

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

[7] Mark Forsyth, Steve Mangelsdork, Eric DeLano, Craig Gleason, and Jeff Yetter. CMOS PA-RISC Processor for a New Family of Workstations. In *COMPCON 91*, pages 202–207, San Francisco, CA, February 25 - March 1 1991.

[8] Hewlett-Packard. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, second edition, September 1992.

[9] Jon Inouye, Ravindranath Konuru, Jonathan Walpole, and Bart Sears. The Effects of Virtually Addressed Caches on Virtual Memory Design & Performance. *Operating Systems Review*, 26(4):896–908, October 1992.

[10] Yousef A. Khalidi and Michael N. Nelson. The Spring Virtual Memory System. Technical Report SMLI TR-93-09, Sun Microsystems Laboratories, Inc., February 1993.

[11] Gregor Kiczales and John Lamping. Operating Systems: Why Object-Oriented? In *Proceedings of Third International Workshop on Object Orientation in Operating Systems (IWOOOS-III)*, pages 25–30, Asheville, NC, December 1993.

[12] Patrick Knebel, Barry Arnold, Mick Bass, Wayne Kever, Joel D. Lamb, Ruby B. Lee, Paul L. Perez, Stephen Undy, and Will Walker. PA7100LC: A Low-Cost Superscalar PA-RISC Processor. In *COMPCON 94*, pages 441–447, San Francisco, CA, February 22-26 1993.

[13] Gordon Kurpanek, Jason Zheng, Eric DeLano, and William Bryg. PA7200: A PA-RISC Processor with Integrated High Performance MP Bus Interface. In *COMPCON 94*, pages 375–382, San Francisco, CA, February 28 - March 4 1994.

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

[15] Theodore H. Romer, Dennis Lee, Brian N. Bershad, and J. Bradley Chen. Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 255–266, Monterey, California, November 1994.

[16] Michael Young, Avadis Tevanian Jr., Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, and Robert Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Procedings of the 11th Symposium on Operating System Principles*, pages 63–75, November 1987.

A Experimental Results

Table 2 summarizes the various attributes of the machines used for the experiments. Table 3 and 4 show the effect of source and destination alignment on byte copy performance. Table 5 contains the results of the bulk copy experiments in respective machine cycles. Table 6 presents the result in terms of microseconds.

Table 2: Machine Configuration

Make/Model	CPU	Clock	Cache
HP 9000/720	PA89	50 MHz	128 KB D-cache, 128 KB I-cache
HP 9000/755	PA7100	99 MHz	256 KB D-cache, 256 KB I-cache
HP 9000/712-80	PA7100LC	80 MHz	256 KB combined cache

Table 3: Byte Alignment Effects (Warm-to-Warm)

Platform	0-byte	1-byte	2-byte	3-byte	4-byte
HP 9000/720	4147	4418	4418	4418	4147
HP 9000/755	3636	4163	4163	4163	3636
HP 9000/712-80	3376	3387	3387	3387	3376

Table 4: Byte Alignment Effects (Clean-to-Clean)

Platform	0-byte	1-byte	2-byte	3-byte	4-byte
HP 9000/720	9583	9830	9830	9830	9583
HP 9000/755	13216	13978	13977	13977	13207
HP 9000/712-80	9420	8306	8306	8306	9420

Table 5: Data Copy Time (in CPU cycles)

HP 9000/720									
copy	Cache Temperature								
method	hot-hot	hot-clean	hot-dirty	clean-hot	clean-clean	clean-dirty	dirty-hot	dirty-clean	dirty-dirty
memcpy	4147	6973	7475	6707	9553	10035	7357	10616	11110
8Wld-8Wst	4154	6975	7482	6714	9556	10042	7363	10217	10756
8Wld-8WstH	4153	6974	7481	6713	9555	10041	7357	10216	10755
4Dld-4Dst	2105	4923	5447	4683	7495	8045	5307	8554	9109
4Dld-2Qst	1850	4925	5448	4410	7483	8046	5062	8361	8853

HP 9000/755									
copy	Cache Temperature								
method	hot-hot	hot-clean	hot-dirty	clean-hot	clean-clean	clean-dirty	dirty-hot	dirty-clean	dirty-dirty
memcpy	3636	8519	8552	8518	13196	13211	8550	15176	15185
8Wld-8Wst	3384	8127	8190	8124	13203	13605	8186	14328	14817
8Wld-8WstH	3384	8127	8190	8124	13203	13605	8186	14325	14820
4Dld-4Dst	1592	5081	7409	6213	10481	12818	7418	12434	14798
4Dld-2Qst	283226	286288	286605	288051	291109	291442	288068	291121	291442

HP 9000/712-80									
copy	Cache Temperature								
method	hot-hot	hot-clean	hot-dirty	clean-hot	clean-clean	clean-dirty	dirty-hot	dirty-clean	dirty-dirty
memcpy	3384	5314	6510	6328	8287	9983	6640	11268	12972
8Wld-8Wst	3255	5193	6510	6199	8158	10491	6518	10639	12968
8Wld-8WstH	3258	5196	6513	6202	8161	10495	6521	10642	12971
4Dld-4DstH	1592	3270	6460	4548	6767	9967	6506	9712	12928
4Dld-2Qst	357609	359118	359705	360565	362126	362427	360567	362126	362417

Table 6: Data Copy Time (in usecs)

HP 9000/720									
copy	Cache Temperature								
method	hot-hot	hot-clean	hot-dirty	clean-hot	clean-clean	clean-dirty	dirty-hot	dirty-clean	dirty-dirty
memcpy	82.9	139	150	134	191	201	147	212	222
8Wld-8Wst	83.1	140	150	134	191	201	147	204	215
8Wld-8WstH	83.1	140	150	134	191	201	147	204	215
4Dld-4Dst	42.1	98.7	109	93.9	150	161	106	171	182
4Dld-2Qst	37.0	98.7	109	88.3	150	161	101	167	177

HP 9000/755									
copy	Cache Temperature								
method	hot-hot	hot-clean	hot-dirty	clean-hot	clean-clean	clean-dirty	dirty-hot	dirty-clean	dirty-dirty
memcpy	36.7	86.1	86.4	86.0	133	133	86.4	153	153
8Wld-8Wst	34.2	82.1	82.7	82.1	133	137	82.7	145	150
8Wld-8WstH	34.2	82.1	82.7	82.1	133	137	82.7	145	150
4Dld-4Dst	16.1	51.3	74.8	62.8	106	129	74.9	126	149
4Dld-2Qst	2861	2892	2895	2910	2941	2944	2910	2941	2944

HP 9000/712-80									
copy	Cache Temperature								
method	hot-hot	hot-clean	hot-dirty	clean-hot	clean-clean	clean-dirty	dirty-hot	dirty-clean	dirty-dirty
memcpy	42.3	66.4	81.4	79.1	104	125	83.0	141	162
8Wld-8Wst	40.7	64.9	81.4	77.5	102	131	81.5	133	162
8Wld-8WstH	40.7	65.0	81.4	77.5	102	131	81.5	133	162
4Dld-4DstH	19.9	40.9	80.8	56.9	84.6	125	81.3	121	162
4Dld-2Qst	4470	4489	4496	4507	4527	4530	4507	4527	4530