Optimizing Kernel Block Memory Operations

Michael Calhoun, Scott Rixner, Alan L. Cox
Rice University
Houston, TX 77005
{calhoun,rixner,alc}@rice.edu

***Abstract*— This paper investigates the performance of block memory operations in the operating system, including memory copies, page zeroing, interprocess communication, and networking. The performance of these common operating system operations is highly dependent on the cache state and future use pattern of the data, and no single routine maximizes performance in all situations. Current systems use a statically selected algorithm to perform block memory operations. This paper introduces a method to dynamically predict the optimal algorithm for each block memory operation. This dynamic selection involves the prediction of the current state of the cache as well as whether or not the target data will be reused before it is evicted. By using the results of these predictions to select the optimal software algorithm, the performance of kernel copy operations can be improved.**

I. Introduction

The disparity between memory bandwidth, memory latency, and the performance of superscalar microprocessors has made memory copies a performance bottleneck for many years. Furthermore, memory latency and bandwidth do not improve as quickly as microprocessor performance, increasing the fraction of execution devoted to block memory operations. Kernel block memory operations, such as page zeroing and copying between user and kernel space, are especially problematic because these operations are often critical for moving data between applications, communication with devices or remote systems, or protection within the system, so they cannot be easily eliminated.

Currently, as much as half of the execution time of modern applications can be spent performing block memory operations within the kernel, making them a potentially significant performance bottleneck. For example, on a modern system composed of an AMD Opteron 250 processor with dual-channel PC3200 SDRAM, 59 to 64% of FreeBSD's execution time is spent performing block memory operations when running the PostMark [1] and Chat [4] benchmarks, as shown in Table 1. These block memory operations account for up to 58.7% of the applications' total execution time.

One of the key reasons why these operations are such a large fraction of the total cycles spent in the kernel is that they have high L2 cache miss rates—up to 23% during kernel copies. Block memory operations are largely due to networking, interprocess communication, and other kernel services. Table 1 shows that microbenchmarks targeted at these operations, bw_pipe [2] and Netperf [3], are dominated by kernel copies.

The performance of block memory operations on modern systems is highly dependent on the state of the system at the time of the operation. The best performing algorithm depends on the cache state of the source and destination and whether or not the destination will be reused before it is evicted from the cache. However, current systems do not make use of this information to determine whether or not the source should be prefetched prior to the copy and whether or not the destination should be written with temporal or non-temporal store instructions. Instead, a single algorithm is statically selected to provide reasonable performance in most situations.

This paper demonstrates that the cache state of the source and destination of a block memory operation can be predicted based upon the cache state of the first element of each block. By considering the location of the destination region of memory along with the reuse pattern of the data, the kernel can adaptively use non-temporal stores when they have the most benefit and avoid using them when the destination is already cached or when the data will likely be reused. This paper characterizes the behavior of block memory operations and proposes a methodology to improve kernel memory operation performance by using predictors to determine cache residency and dynamically selecting the best block memory algorithm.

The rest of this paper proceeds as follows. The next section discusses previous work in studying and improving the performance of block memory operations. Section III then describes the behavior and performance of efficient block memory operations. Section IV shows how to use this information in order to dynamically select the appropriate copy algorithm. Finally, Section V concludes the paper.

II. Background

As shown in Table 1, block memory operations within the operating system can consume a significant fraction of kernel execution time. These operations are expensive because they typically exhibit worse cache behavior than other operations.

The poor cache behavior of the operating system is not a new observation. In 1988, Agarwal *et al.* observed that interference between memory accesses by the operating system and memory accesses by the application resulted in higher than expected cache miss rates. Furthermore, they observed that memory accesses by the operating system exhibited higher cache miss rates than memory accesses by the application.

Chen and Bershad [5] investigated the performance of block memory operations when they characterized the memory system performance for two implementations of Unix: Mach, based on a microkernel architecture, and Ultrix, based on a

Table 1. Kernel copy profiles.

		FreeBSD Compile	PostMark [1]	bw_pipe [2]	Netperf [3]	Chat [4]
Breakdown of copies by type (%)	bzero	21.4	84.7	0.0	4.8	36.0
	bcopy	26.5	2.8	0.0	28.8	27.8
	copyin	8.7	4.4	0.6	31.4	33.6
	copyout	43.4	8.0	99.4	35.0	2.6
Fraction of cycles spent copying (%)	Kernel	26.4	59.4	76.6	99.1	64.3
	Total	3.1	51.5	74.0	99.0	58.7

monolithic kernel architecture. Through simulation they derive the memory cycles per instruction (MCPI) for Mach and Ultrix under a variety of workloads. Their results show that memory copying and zeroing by the operating system accounts for a large fraction of its MCPI, specifically, up to 26.6% for Mach and up to 39.3% for Ultrix. Although memory copying and zeroing was a larger fraction of Ultrix's MCPI, in absolute terms the contribution of memory copying and zeroing to MCPI was higher for Mach.

While there has been little work specifically targeting the cache behavior of the operating system, there has been significant cache and prefetching research relevant to the performance of block memory operations.

A. *Cache Policies*

Previous research has shown that the cache policies frequently implemented in modern machines are not necessarily the ideal policies for block memory operations. Several studies have examined the interaction between memory system architecture and copying garbage collection [6], [7], [8], [9]. In order to improve cache performance for Lisp programs, Peng and Sohi [7] advocated an ALLOCATE instruction which allocates a line in the cache without fetching it from memory. Both Wilson *et al.* [8] and Diwan *et al.* [6] studied SML/NJ programs, concluding that the cache should implement a write-allocate policy with sub-block placement. In such a cache, each cache line consists of two or more sub-blocks and each sub-block has its own valid bit. Like Peng and Sohi's ALLOCATE instruction, write allocate with sub-block placement permits the allocation of a cache line without fetching it from memory. The difference is that to avoid the fetch a write miss must completely overwrite the sub-block. Block memory operations are likely to overwrite at least one sub-block, as most of the data written is larger than 1/4–1/2 of a cache line.

Jouppi also studied the effects of different cache write policies on the performance of numeric programs, CAD tools, and Unix utilities [10]. For handling write misses, he came to the same conclusion as Diwan *et al.* [6]: he advocated a combination of a no-fetch-on-write policy, implemented by sub-block placement, and a write-allocate policy. Specifically, he found that reuse frequently occurred before eviction under a write-allocate policy. Hence, a write-allocate policy outperformed a write-around policy.

Unfortunately, neither an ALLOCATE instruction nor sub-block placement are supported by today's widely-used, general-purpose processors. Furthermore, data reuse before eviction of the writes that are performed by block memory operations in the kernel is dependent on operating system and application behavior. These access and reuse patterns vary widely between applications, and the use of no-fetch-on-write or write-allocate with sub-block placement policy is not necessarily possible or appropriate for kernel block memory operations on modern microprocessors. However, modern microprocessors do provide non-temporal store instructions and write combining buffers. In many ways these approximate the behavior of the ALLOCATE instruction. If an entire cache line is written using non-temporal stores, then the data is aggregated in a write combining buffer and is not fetched into the cache. In addition, when the data is released from the write combining buffer, it goes straight to memory. This both eliminates the initial fetch and also prevents the new data from polluting the cache. Unfortunately, there are typically very few write combining buffers and non-temporal stores perform poorly when the data is already cached, as they force an eviction from the cache before the data is written rather than afterwards.

B. *Prefetching Policies*

Past prefetching work has largely focused on predicting what locations to prefetch, attempting to perform those prefetches early enough, and minimizing useless prefetches that waste memory bandwidth. Software prefetching can be quite effective at accomplishing these goals [11], [12], [13], [14], [15]. However, it can also incur significant overhead. For example, prefetching an 8 KB block of 64 byte cache lines would require 128 prefetch instructions. Hardware prefetching, on the other hand, incurs no instruction overhead, but is much more likely to waste memory bandwidth with useless prefetches and to perform prefetches late [16]. Since block memory operations perform memory accesses sequentially, very simple *one block lookahead* prefetchers, such as prefetch-on-miss and tagged prefetch [17], can be quite effective. However, with today's memory latencies one block lookahead may not result in timely prefetches, so more aggressive sequential prefetching, such as *next-N-line* prefetching [17], [18] and *stream buffers* [19], are likely to be necessary in order to prefetch all blocks in a timely fashion. The problem with such aggressive sequential prefetchers is that they will perform numerous useless prefetches and waste memory bandwidth in other portions of code [20].

To achieve the accuracy of software prefetching and the low overhead of hardware prefetching, integrated hard-

```
for each 8 KB block
   prefetch 8 KB
   for each word in the block
      load the word
      store the word
   end for
end for
```

Fig. 1. Efficient block copy algorithm.

ware/software prefetch engines are necessary [21], [22], [23], [24]. It is possible that such schemes can also improve copy performance by having the software inform the hardware prefetcher to perform aggressive sequential prefetching upon entrance to the copy routine. Furthermore, prefetching with write hints in order to prefetch the destination into an exclusive cache state may also allow improved copy performance. A few architectures, including the MIPS, have such prefetching instructions.

III. Block Memory Operations

Given the limitations of the cache policies and prefetching mechanisms of modern microprocessors, it is important to structure block memory operations appropriately. Modern SDRAM provides significantly better performance when consecutive locations within the same row are accessed than when accesses to conflicting rows are interleaved with each other. Therefore, the most efficient copy algorithms must consider and take advantage of the organization of modern SDRAM. An optimized copy routine should be structured to prefetch a block of the source data that can fit within the cache, copy that block of data to the destination, and repeat for the length of the copy, as shown in Figure 1. This procedure maximizes the locality within the SDRAM when either or both of the source and destination are not cached. The example code first prefetches the entire block, accessing the DRAM sequentially if the source block is uncached. It then performs loads and stores in which all of the loads will hit in the cache, as the data has been prefetched. So, if the destination block is uncached, it will also be accessed sequentially in the DRAM. However, the potential write back traffic caused by evicted dirty lines could interfere with sequential access for the destination block.

The block size should be large enough to amortize the row activations within the SDRAM, and small enough to fit comfortably within the processor caches. An 8 KB block, as shown in the Figure 1, satisfies these constraints. For simplicity, the figure does not show the unrolling of the inner loop to perform a series of loads and then stores, or the pre and post loops that would exist in order to align the copy to 8 KB blocks. Alignment of the copy regions eliminates row crossings within each block that can lead to decreased SDRAM performance, and unrolling the inner loop decreases control overhead and allows grouping of loads and stores for better DRAM locality.

A. *Copy Algorithms*

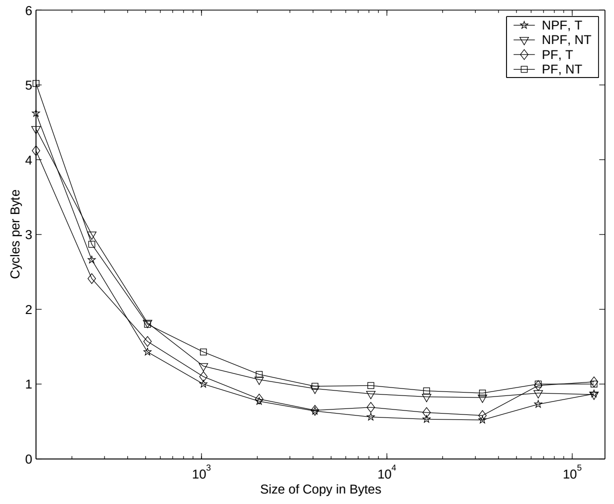
There are two main axes along which the copy algorithm presented in Figure 1 can be varied. First, prefetching of the source can be either included or excluded. If the source is known to be in the cache, then using software prefetch instructions to bring it into the cache is an unnecessary waste. Even if the source is uncached, modern hardware prefetchers perform quite well on sequential accesses and may achieve the same or better performance as long as the destination is cached. However, the organization of the SDRAM will cause significant overhead if the hardware prefetcher is required to fetch the source and destination at the same time.

Second, the store instructions can either leave the destination data within the cache or they can provide non-temporal hints. If the destination data is not already in the cache, it may improve performance to write directly to memory, eliminating the need to first pull in data from memory that is not actually needed. Non-temporal stores can also reduce the interference in the DRAM caused by cache write backs.

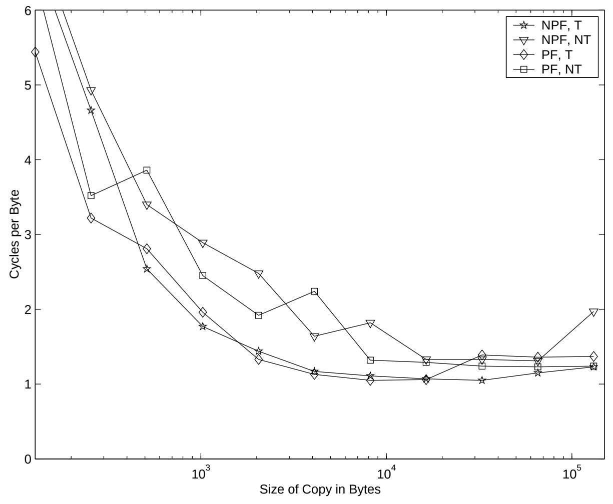
The behavior of the copy algorithm presented in Figure 1 is mostly independent of the particular implementation of the prefetch and store operations, only requiring that the architecture has a means of performing such memory operations. For example, the IA-32 architecture provides the prefetch instruction is `prefetchnta`, which prefetches the data with a non-temporal hint. This instructs the processor that it is unlikely that the source data for a block copy will be reused, so the non-temporal hint helps to minimize cache pollution. Similarly, if the destination is not in the cache, and it is unlikely to be referenced again soon, the `movnti` store instruction is the most appropriate. The `movnti` instruction stores the data with a non-temporal hint. This means that if the memory location is currently in the cache, the cache line is first evicted, then the data is written using write combining. In a block copy, the entire cache line should be aggregated in a write combining buffer and then stored to memory directly. However, if the data is initially in the cache, the initial eviction is both costly and unnecessary, making the `movnti` instruction a poor choice for cached data. The IA-32 architecture does not provide any other mechanisms for efficiently performing non-temporal block writes. Other implementations which do not force eviction of cached data for non-temporal stores may have better overall application performance.

B. *Copybench*

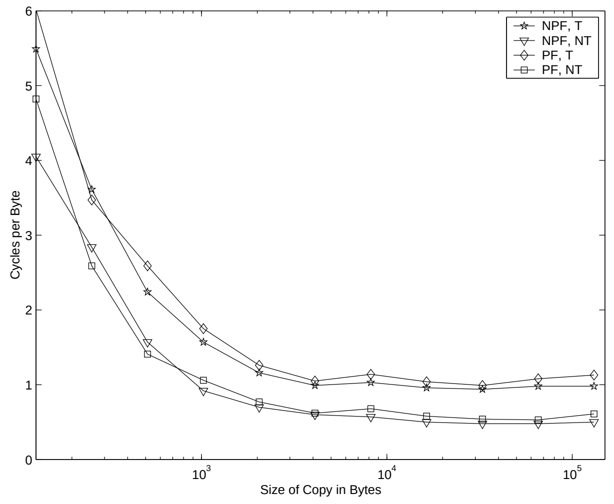
A simple microbenchmark illustrates the performance of the copy routine shown in Figure 1. The microbenchmark allocates the source and destination regions of memory and performs a series of memory operations to ensure both regions are uncached. Next, it moves one or both regions into the cache depending on the test that is being run. Finally, it records the number of cycles it takes to execute a memory copy of the desired size between source and destination. This provides important performance information about each of the copy algorithm optimizations as the cache state and copy size varies.



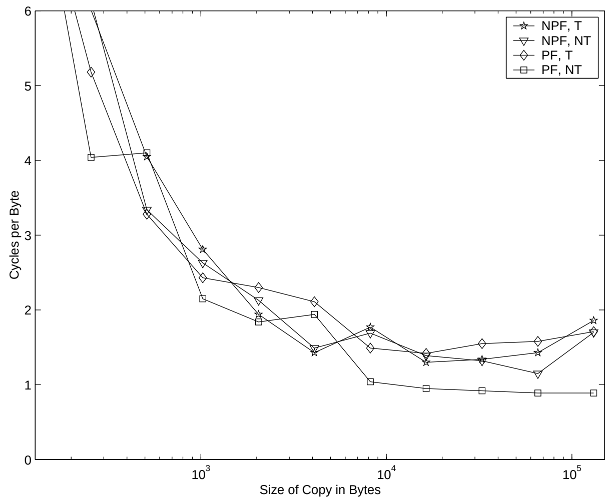
(A) Source and destination cached



(B) Source uncached, destination cached

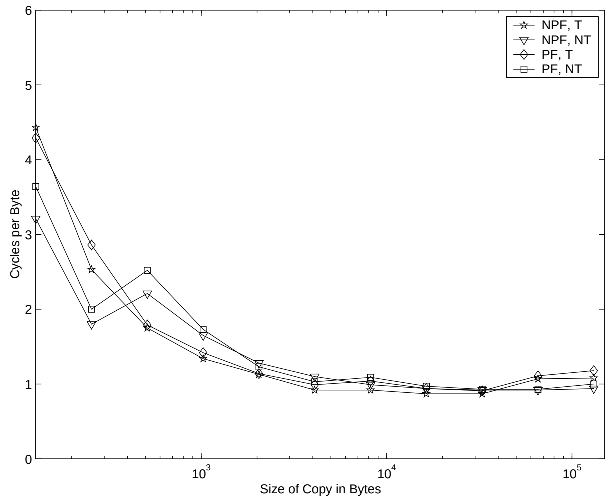


(C) Source cached, destination uncached

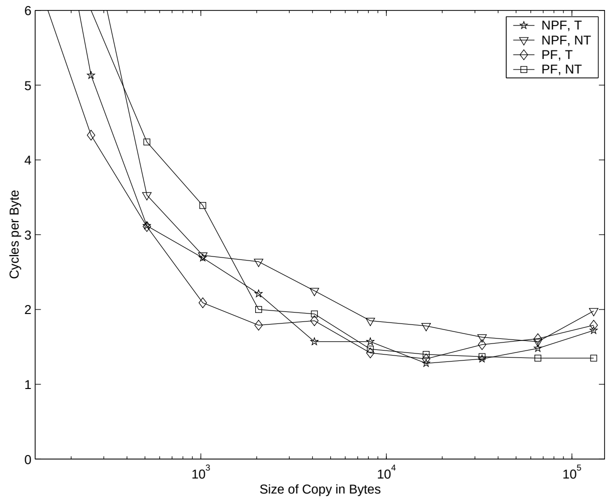


(D) Source and destination uncached

Fig. 2. Copy performance (cycles per byte copied) for different copy sizes when the source and destination are cached or uncached.



(A) Source cached, destination uncached



(B) Source and destination uncached

Fig. 3. Copy performance (cycles per byte copied) for different copy sizes when the destination is uncached and the target data is read after the copy.

Figure 2 shows the performance of the copy routine described in Figure 1 on an AMD Opteron 250 processor with a 64 KB L1 data cache, a 1 MB unified L2 cache, and dual-channel PC3200 SDRAM. The routine is varied along the two axes described above and the source and destination data are either cached or uncached at the start of each copy. The inner loop of the copy algorithm shown in Figure 1 is unrolled to load and store an entire cache line each iteration. Each line in the graphs represents a different variation of the copy algorithm. *NPF* indicates that no prefetching is performed, whereas *PF* indicates that `prefetchnta` instructions were used. *T* indicates that normal temporal stores were used within the copy loop, and *NT* indicates that non-temporal stores were used (`movnti`). For this system, the peak DRAM bandwidth of the machine is 0.375 processor cycles per byte. If the copy has to access the DRAM for both the source and destination, the peak copy bandwidth through the DRAM is 0.75 cycles per byte.

The graphs show three key points that help explain the behavior of block memory operations. First, small copies are extremely expensive. Regardless of the version of the copy algorithm or the cache state, copies less than 1–2 KB are not long enough to amortize the overhead of the copy, resulting in large numbers of cycles per byte. Though all small copies are expensive, the differences among the versions and states still result in slight variations in memory performance. Second, for a given cache state, different versions of the copy algorithm perform differently. For example, Figure 2A shows that when both the source and destination are in the cache, using non-temporal stores instead of temporal stores makes the copy take one half cycle per byte more time. This difference in performance occurs because current implementations of non-temporal stores first evict the target cache line and then perform the write combining store to memory. Finally, the ordering of the versions of the copy algorithm depends upon the cache state. For example, when both the source and destination are cached, then the temporal copies perform best. But, when only the source is cached, the non-temporal copies perform best and achieve a level of performance comparable to the temporal stores in the case where both regions were cached.

Figure 3 illustrates how the behavior of the copy algorithms differ when the destination of the copy is read immediately after the copy completes. Figure 2 shows that, when the destination of the copy is uncached, a non-temporal store will yield the best performance for the copy, performing up to 75% better than the temporal store instructions. This data motivates the selective use of non-temporal stores to improve block memory operation performance when the destination region of memory is initially uncached. However, optimizing the algorithm for copy performance does not yield the best application performance when the data is immediately reused, showing 5–25% worse performance than temporal stores when the destination data must be re-read into the cache for later use. If the data is going to be reused before cache eviction, temporal stores perform better as shown in Figure 3. By fetching the destination region of memory into the cache to write it instead of sending the data directly to memory, the application benefits when the next accesses to the destination data are cached. Non-temporal stores incur a significant performance penalty when the data is written to DRAM but then reread from main memory shortly thereafter. Storing the data directly in the cache instead of memory saves an expensive DRAM transaction when the application accesses the data before eviction. However, as the figure shows, when the block size exceeds the cache size, non-temporal stores become useful again by moving data that will not be reused directly to DRAM, avoiding unnecessary cache fills.

Figures 2C and 3A illustrate this concept clearly. Without reuse of the destination data, non-temporal stores outperform temporal stores by up to 100%: .55 cycles/byte compared to 1.1 cycles/byte. However, when the time to access the data again is taken into account, temporal stores perform better for all but the smallest and largest copies.

IV. Algorithm Selection

The data from the previous section argues that the kernel should dynamically select the appropriate copy algorithm based on the current cache state of the source and destination blocks and the likelihood that the destination block will be reused before it is evicted from the cache. Figure 4 shows the selection process based upon predictions of the cache state and target reuse. If the predictions are accurate, then this process will result in the optimal copy algorithm for the current situation. Therefore, for this to be a viable approach, the cache state of the source and destination blocks and probability that the destination block will be reused must be predicted reliably.

A. Predicting the Current Cache State

Intuitively, it does not seem practical to predict whether or not an entire block is cached. Current architectures do not provide a means of communicating the contents of the cache to the system or the user. A region of memory involved in copy may span a few or hundreds of 64B cache lines, so a determination as to the cache state of a region may involve accessing large amounts of data. However, the cache state of the first word of a region of memory is a good indicator of whether or not the entire block is cached. Table 2 shows the accuracy of such a prediction, collected using an augmented version of the Simics simulator [25]. The applications were run on top of the FreeBSD 4.10 operating system. For this table, we considered a block to be cached if more than 60% of the elements were in the cache, and uncached if less than 40% of the elements were in the cache. For blocks with 40–60% of their elements in the cache (less than 5% of all blocks in these applications), it is unclear what the appropriate copy routine would be, so the probe accuracy is not as important. The data is broken down by block size and by application. For example, the cache state of the first element of the source block for 1025–2048 byte copies is an accurate predictor of the cache state of the entire block 99.9% of the time for PostMark. As the table shows, the cache state of the first word

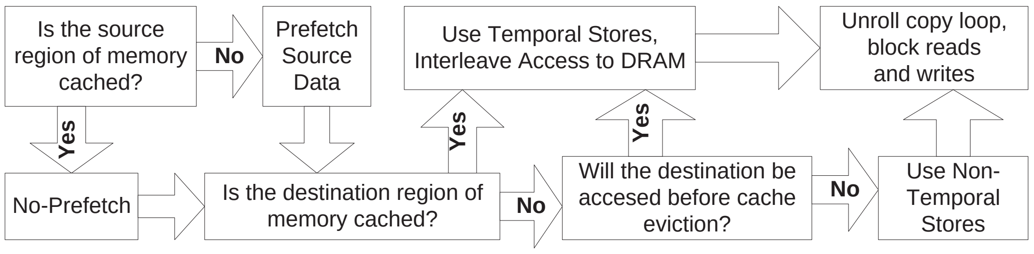

Is the source region of memory cached?
No
Prefetch Source Data
Use Temporal Stores, Interleave Access to DRAM
Unroll copy loop, block reads and writes
Yes
Yes
Yes
No-Prefetch
Is the destination region of memory cached?
No
Will the destination be accesed before cache eviction?
No
Use Non-Temporal Stores

Fig. 4. Method for dynamically selecting of the appropriate copy algorithm.

Table 2. Probe accuracy by block size. The “blocks” columns list the number of source blocks being copied (because of the `bzero` operation, there can be more destination than source blocks). The “src” and “dst” columns list the percentage of probes that accurately determine if the source or destination block, respectively, is cached or uncached. Each row includes all block memory operations up to the given size, but larger than the size given in the previous row. No block memory operations were larger than 64 KB for these benchmarks.

Block	**FreeBSD Compile**			**PostMark**			**bw_pipe**			**Netperf**			**Chat**		
Size	**Blocks**	**Src**	**Dst**	**Blocks**	**Src**	**Dst**	**Blocks**	**Src**	**Dst**	**Blocks**	**Src**	**Dst**	**Blocks**	**Src**	**Dst**
128 B	1215683	99.9	99.1	644817	99.9	99.9	2098078	99.9	99.9	660667	99.9	99.9	116394	99.9	99.9
256 B	12433	95.8	43.1	2220	98.6	46.2	6	100.0	31.3	140	72.9	21.8	36651	99.8	82.5
512 B	12666	99.2	82.7	3721	98.2	99.8	13	100.0	100.0	853	95.1	99.8	24759	98.9	99.4
1 KB	8021	99.5	97.3	3696	100.0	92.3	0	—	—	0	—	—	5516	89.3	94.7
2 KB	9890	98.1	96.9	9498	99.9	96.8	0	—	—	1953838	99.9	50.0	785462	93.2	96.5
4 KB	65282	91.2	92.3	17253	97.6	97.3	170	61.8	96.4	2159	82.1	99.4	48940	91.9	97.3
8 KB	12410	99.6	91.4	20908	99.9	94.3	0	—	—	0	—	—	23028	37.8	11.0
16 KB	18658	91.0	90.3	10430	99.7	71.4	1	100.0	100.0	83	100.0	100.0	15710	67.7	5.1
32 KB	22	100.0	94.6	0	—	98.7	0	—	—	0	—	—	75	100.0	60.0
64 KB	17	100.0	100.0	0	—	—	1125555	58.4	99.3	0	—	—	0	–	–
Total	**1355082**	**99.3**	**97.9**	**712543**	**99.8**	**99.0**	**200554**	**99.9**	**99.9**	**2617740**	**99.9**	**67.9**	**1056547**	**92.6**	**93.3**

Table 3. Percentage of cache lines that are the destination of a block memory operation that are reused, unused, or overwritten by another block memory operation.

Benchmark	**Reused**	**Unused**	**Retargeted**
FreeBSD Compile	89.3	2.7	8.0
PostMark	70.7	10.5	18.8
bw_pipe	0.2	0.0	99.8
Netperf	51.3	24.1	24.6
Chat	73.3	12.5	14.2

of a block indicates whether or not the block is cached well over 90% of the time, except for the destination blocks in Netperf. Therefore, if the cache state of the first element of the source and destination blocks can be determined, then that information can be used as a very accurate predictor of the cache state of the entire block, allowing the appropriate copy algorithm to be selected.

B. Predicting Data Reuse

Determining whether or not the target of a block memory operation will be reused before it is evicted from the cache requires prediction rather than querying the current system state. However, most data is reused within the cache, as would be expected. Table 3 shows that significantly more than half of the cache lines that are targets of block memory operations are reused. The third column in the table shows the percentage of cache lines that were the target of a copy and then, before they could be evicted from the cache, were targeted again by a copy. This behavior is common in kernel block memory operations, as system memory allocated for I/O operations is often reused. Because these regions of memory are frequently reused, the kernel can optimize the copy algorithms to keep these regions of memory in the cache. In the case of sending data over the network, the kernel can use non-temporal stores to copy the data from the application buffer to the socket buffer. The network stack offloads the calculation of the TCP/IP checksum to the network interface, which means that the data segment of the packet is not modified by the system after the application initiates the send. This reduces cache pollution by not occupying space in the cache before the network interface initiates a DMA to move the data over the PCI bus. Additionally, the operating system opportunistically zeros reclaimed pages when it would otherwise be idle. Since these pages are unlikely to be used again soon, the kernel can use non-temporal stores while zeroing them so that useful data is not evicted from the cache.

C. Software Cache Probe

To select the appropriate variant of the copy algorithm shown in Figure 1, it is necessary to determine the cache state prior to the copy. Given that the first element of a block is a good predictor of the cache state of the entire block, the time it takes to complete a load to that first element is a reasonable metric to use to predict the cache state of the block. By forcing all other memory operations to complete using memory fence instructions, a load can be “timed” using the read time-stamp counter, `rdtsc`, instructions. By comparing the time to a cache-locality threshold measured for this particular architectural implementation, the software can predict the cache state for that load and select the best copy algorithm. The locality thresholds for a system can be dynamically measured and set using system controls, and depend on the speed of the specific processor and memory for the current system.

Memory fence instructions ensure that the appropriate memory reference is being timed. Otherwise, there could be several outstanding memory references that delay the probing reference, yielding inaccurate results. However, such fence instructions cause significant performance degradations because they serialize memory accesses. This forces all memory references before the timed load to complete before it begins and all memory references after the timed load to wait until it completes. The execution of the timed load instruction allows only that one operation to proceed in the processor at a time, further slowing execution. If the probed cache line is not in the cache, then the load instruction will access main memory and fetch the required line into the cache, and, because of the use of memory fence instructions, the processor will not be able to perform any other memory references until the return of the cache line from DRAM. Depending on the cache state, the software probe takes between 18 cycles for an L1 hit and 400 cycles for a DRAM access.

The use of these memory fence instructions severely limits memory performance by forcing all memory operations to complete, then performing the single timed load, and then allowing subsequent memory operations to start only after it completes. It is difficult to measure the performance impact of the fence operations in the software probe because of this memory serialization. Preliminary experiments using such a probe in the FreeBSD kernel, however, show significant performance improvements for Netperf and slight slowdowns for the other benchmarks. This indicates that this particular implementation of the software probe is too expensive, both because of the high cost of the timed load itself and the memory serialization effects that prevent overlap between memory references before and after the timed load. Since these memory serialization effects cannot be quantified, it is difficult to draw conclusions about the technique from these results.

D. Hardware Cache Probe

The high overhead of a software based cache probe motivates the addition of a new hardware cache probe instruction. While useful, a cache probe instruction does not need to be complicated. Such an instruction could simply return a value indicating what level of the cache hierarchy contains the data, if any. This only requires accessing the cache hierarchy in the same way that a load instruction would, except if there is a miss in the lowest level of the cache hierarchy, there is no need to access SDRAM to retrieve the data. This access to DRAM when the block of memory being probed turns out to be uncached is a significant cost of the software-based probe. A hardware cache probe instruction simply returns with a value indicating the data is not cached, rather than causing an external memory reference. Such an instruction would be quite efficient, as it would take no longer than the longest cache hit time and could be overlapped with other memory references, yet it would still provide the required data necessary to predict the state of the source or destination region of memory. Because the cache's data array never needs to be accessed, the performance of the cache probe is better than the best-case hit time of the level of cache being probed. Such an instruction requires neither memory serialization nor timing information, so it incurs none of the overheads of the software probe mechanism.

V. Conclusions

Block memory operations occur frequently in the kernel while performing a variety of common system tasks, such as networking, interprocess communication, and I/O. The performance impacts of the memory system requires the kernel to consider the factors described in this paper to be able to select the optimal algorithm for these block memory operations.

The contributions of this paper are threefold. First, the performance of block memory operations are dependent on the cache state and data reuse pattern of the memory involved in the operation. Second, the state of the first word in a block of memory can be used as an accurate predictor of the cache state of the entire block of memory. Finally, the cache can be dynamically probed to determine the state of a cache line by measuring the latency of a memory access or by the addition of a hardware instruction.

Because of the significant overheads associated with the memory timing and serialization required by the software probe mechanism, it is impractical for general use. However, the software probe does demonstrate the feasibility of selecting algorithms for block memory operations dynamically. Furthermore, the high cost of block memory operations within the kernel and the potential benefits of dynamic algorithm selection demonstrated by this paper motivate additional research into the feasibility and benefits of a hardware cache probe mechanism.

References

[1] J. Katcher, "PostMark: a new file system benchmark," Network Appliance, Tech. Rep. TR3022.

[2] L. McVoy and C. Staelin, "lmbench: portable tools for performance analysis," in *Proceedings of the 1996 USENIX Technical Conference*, January 1996, pp. 279–295.

[3] *Netperf: A Network Performance Benchmark*, Information Networks Division, Hewlett-Packard Company, February 1995, revision 2.0.

[4] *Chat - A Simple Chat Room Benchmark*, International Business Machines (IBM), December 2000, revision 1.0.1.

[5] J. B. Chen and B. N. Bershad, "The impact of operating system structure on memory system performance," in *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*. ACM Press, 1993, pp. 120–133.

[6] A. Diwan, D. Tarditi, and E. Moss, "Memory subsystem performance of programs using copying garbage collection," in *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, Oregon, 1994, pp. 1–14. [Online]. Available: citeseer.ist.psu.edu/article/diwan94memory.html

[7] C.-J. Peng and G. S. Sohi, "Cache memory design considerations to support languages with dynamic heap allocation," Computer Sciences Department, University of Wisconsin-Madison, Tech. Rep. 860, July 1989.

[8] R. P. Wilson, M. S. Lam, and T. G. Moher, "Caching considerations for generational garbage collection," in *Proc. 1992 ACM Conf. on Lisp and Functional Programming*, San Francisco CA (USA), jun 1992, pp. 32–42. [Online]. Available: citeseer.ist.psu.edu/wilson92caching.html

[9] B. Zorn, “The effect of garbage collection on cache performance,” University of Colorado, Department of Computer Science, Tech. Rep. CU–CS–528–91, 1991. [Online]. Available: citeseer.ist.psu.edu/zorn91effect.html

[10] N. Jouppi, “Cache write policies and performance,” in *Proceedings of the 20th International Symposium on Computer Architecture*, 1993.

[11] D. Callahan, K. Kennedy, and A. Porterfield, “Software prefetching,” *SIGARCH Computer Architecture News*, vol. 19, no. 2, pp. 40–52, 1991.

[12] A. C. Klaiber and H. M. Levy, “An architecture for software-controlled data prefetching,” *SIGARCH Computer Architecture News*, vol. 19, no. 3, pp. 43–53, 1991.

[13] T. Mowry, M. Lam, and A. Gupta, “Design and evaluation of a compiler algorithm for prefetching,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992, pp. 62–73.

[14] D. Bernstein, D. Cohen, and A. Freund, “Compiler techniques for data prefetching on the PowerPC,” in *PACT ’95: Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, 1995, pp. 19–26.

[15] V. Santhanam, E. H. Gornish, and W.-C. Hsu, “Data prefetching on the HP PA-8000,” in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997, pp. 264–273.

[16] D. Joseph and D. Grunwald, “Prefetching using Markov predictors,” in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997, pp. 252–263.

[17] A. J. Smith, “Cache memories,” *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982.

[18] ——, “Sequential program prefetching in memory hierarchies,” *IEEE Computer*, vol. 11, no. 2, pp. 7–21, 1978.

[19] N. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” in *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990, pp. 364–373.

[20] S. Przybylski, “The performance impact of block sizes and fetch strategies,” in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 160–169.

[21] E. H. Gornish and A. Veidenbaum, “An integrated hardware/software data prefetching scheme for shared-memory multiprocessors,” in *Proceedings of the International Conference on Parallel Processing*, 1994.

[22] T. Chen, “An effective programmable prefetch engine for on-chip caches,” in *Proceedings of the International Symposium on Microarchitecture*, 1995.

[23] S. P. VanderWiel and D. J. Lilja, “A compiler-assisted data prefetch controller,” in *Proceedings of the International Conference on Computer Design*, 1999.

[24] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems, “Guided region prefetching: A cooperative software/hardware approach,” in *Proceedings of the International Symposium on Computer Architecture*, 2003, pp. 1–11.

[25] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, and J. Hogberg, “Simics: A full system simulation platform,” *IEEE Computer*, vol. 35, no. 2, 2002.