RICE UNIVERSITY

Characterization of Block Memory Operations

by

Michael Calhoun

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Scott Rixner, Chair and Advisor
Assistant Professor of Computer Science and
Electrical and Computer Engineering

Alan L. Cox
Associate Professor of Computer Science

Peter Varman
Associate Professor of Electrical and
Computer Engineering and Computer
Science

Houston, Texas

Apr, 2006

Characterization of Block Memory Operations

Michael Calhoun

Abstract

Block memory operations are frequently performed by the operating system and consume an increasing fraction of kernel execution time. These operations include memory copies, page zeroing, interprocess communication, and networking. This thesis demonstrates that performance of these common OS operations is highly dependent on the cache state and future use pattern of the data. This thesis argues that prediction of both initial cache state and data reuse patterns can be used to dynamically select the optimal algorithm. It describes an innovative method for predicting the state of the cache by using a single cache-line probe. The performance of networking, which is dominated by kernel copies, is improved by the addition of dedicated hardware in the network interface. Finally, based upon the behavior of block memory operations, this thesis proposes improvements such as a hardware cache probe instruction, a dedicated memory controller copy engine, and centralized handling of block memory operations to improve performance in future systems.

Acknowledgments

First and foremost I would like to thank Dr. Scott Rixner for all of his help during my graduate career. Without his motivation, advice, and support, I would not have made it this far. He took a chance on me during a tumultuous time in the department, for which I owe him for all of the progress and learning I have achieved since. Also, Dr. Alan L. Cox was a significant contributor to my work along with providing valuable insight into the operating systems issues that I considered. I would like to thank my Thesis Committee for working within the constraints that I imposed on them regarding my defense and submission time-frame.

Additionally, Hyong-Youb Kim provided valuable assistance in the programming intensive parts of my research. The rest of the Rice Computer Architecture Group (Paul, Jeff, Mike, Supratik) was always there for valuable moral and technical support throughout my time as a member of the group.

I would like to thank Amy, without whom every day would be much more difficult. Sure, we picked a bad time to simultaneously get engaged, plan a wedding, complete a graduate degree, finish a thesis (each!), get a new job, and move to a new city, but years from now we will be happy we did.

Finally, my parents and family were constant in their support, always believing that if I tried, I would succeed.

Contents

Abstract i

List of Illustrations vi

List of Tables viii

1 Introduction 1

1.1 Contributions 2

1.2 Organization 4

2 Background and Related Work 6

2.1 Kernel Block Memory Operations 6

2.1.1 Cache Policies 7

2.1.2 Prefetching Policies 9

2.1.3 Existing Copy Algorithms 10

2.2 Networking 11

2.2.1 TCP Offloading 11

2.2.2 Zero-Copy I/O 13

2.2.3 Asynchronous Data Movement 14

2.3 Summary 15

3 Block Memory Operation Behavior 16

3.1 Pipes 17

3.2 Process and Memory Management 19

3.3 Summary 21

4 Block Memory Operation Performance 22

4.1 Memory Behavior of Block Memory Operations 22

4.2 Copy Algorithms 23

4.3 Copybench 25

4.4 Copybench and the Pentium 4 30

4.5 Summary 31

5 Algorithm Selection 34

5.1 Predicting the Current Cache State 34

5.2 Predicting Data Reuse 37

5.3 Software Cache Probe 38

5.4 Hardware Cache Probe 41

5.4.1 Hardware Cache Probe Versus Informing Loads 42

5.4.2 Hardware Probe Implementation 44

6 Memory Controller Data Movement 46

6.1 Memory Controller Copy Engine Architecture 47

6.2 Copy Engine Performance 48

6.3 Issues with an Asynchronous Copy Engine 50

6.4 Potential Uses of a Memory Controller Copy Engine 51

7 Networking 53

7.1 TCP Segmentation Offload 54

7.1.1 Netperf Results 56

7.1.2 Apache Results 59

7.2 Remote Direct Memory Access (RDMA) 60

7.2.1 MPICH Results 63

7.2.2 RDMA, TSO, and Block Memory Operations 64

8 Future Work 66

9 Conclusions 70

Bibliography 73

Illustrations

3.1 Communication Between Applications Using Kernel Pipes 17

3.2 Block Memory Operations in Interprocess Communication 19

4.1 Efficient block copy algorithm. 24

4.2 Opteron Copy performance (cycles per byte copied) for different copy sizes when the source and destination are cached or uncached. 26

4.3 Opteron Copy performance (cycles per byte copied) for different copy sizes when the destination is uncached and the target data is read after the copy. 27

4.4 Pentium 4 Copy performance (cycles per byte copied) for different copy sizes when the source and destination are cached or uncached. 33

5.1 Method for dynamically selecting of the appropriate copy algorithm. 35

5.2 Software cache probe routine (IA-32 assembly). 39

6.1 Proposed Architecture of a Memory Controller Copy Engine 47

6.2 Bandwidth Achieved by Memory Controller Copy Engine as Buffer and Block Size Vary 49

7.1 Block Memory Operations in TCP/IP Networking 54

7.2 TCP Segmentation Offload 55

7.3 TCP Segmentation Offload and Netperf 57

Tables

3.1 Kernel copy profiles. 17

5.1 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. 36

5.2 Probe accuracy by block size for the Networking Benchmarks. 36

5.3 Percentage of cache lines that are the destination of a block memory operation that are reused or unused before eviction from the cache 37

5.4 Performance of the Software Probe Algorithm 41

7.1 Netperf and TSO. 56

7.2 ApacheBench and TSO. 60

7.3 NAS 2.0 Benchmarks - MPICH MPI Library - RDMA and TSO 63

Chapter 1

Introduction

The disparity between memory bandwidth, memory latency, and the performance of superscalar microprocessors has made memory copies a performance bottleneck for many years. Further compounding this problem is the failure of memory systems to keep pace with improvements in microprocessors. In modern systems, it is not uncommon to see latencies of hundreds of processor cycles for a memory transaction. Because memory latency and bandwidth do not improve as quickly as microprocessor performance, an increasing fraction of execution time is devoted to block memory operations. Kernel block memory operations, such as page zeroing and copying between user and kernel space, are especially problematic. These operations are often critical for moving data between applications, communication with devices or remote systems, and protection within the system, so they cannot be easily eliminated.

This thesis characterizes the behavior of block memory operations and proposes a methodology to improve kernel block memory operation performance. It describes common block memory operations and evaluates their performance as the cache state and data reuse pattern is varied. The differing performance achieved by the various block memory algorithms motivates the methodology of dynamically selecting the optimal copy algorithm after determining the current system state. The thesis further argues that the frequent and

predictable behavior of block memory operations could be further improved with a dedicated hardware innovation located in the memory controller. This thesis discusses the behavior and provides a performance analysis of hardware mechanisms in the network interface designed to accelerate block memory operations during networking.

1.1 Contributions

The contributions of this paper are as follows:

1. This thesis demonstrates that performance of block memory operations on modern systems is highly dependent on the state of the system at the time of the operation. The performance of the best algorithm is dependent 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. Even though these block memory operations are becoming a bottleneck in modern operating system performance, current systems do not utilize this information to determine whether the source should be prefetched prior to the copy and if the destination should be written with temporal or non-temporal store instructions.

2. This thesis arrives at a surprising result 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 accounting for the location of the source and destination region of memory along with the reuse pattern of the data, the kernel can adaptively use non-temporal stores and prefetch instructions when they are most

beneficial and avoid using them when the destination is already cached or when the data will likely be reused.

3. This thesis further demonstrates that the special behavior of block memory operations in networking lend themselves to optimizations that use specialized hardware in the network interface to reduce or eliminate the copies required to send data over the network. It disproves a common perception that networking is a compute bound operation by demonstrating that the benefits of TCP Segmentation Offload are due to improvements in the performance of block memory operations. These optimizations are essential to improving the performance of the networking subsystem as communication capacity increases.

4. This thesis argues that the behavior of block memory operations suggest a series of innovations that could improve the performance of the operating system in future systems. First, a hardware cache probe instruction that can determine if a given address is resident in the cache would significantly reduce the overhead required to select an optimal copy algorithm. Secondly, an enhancement to the memory controller that provides a means of asynchronously performing cache-to-memory and memory-to-memory copies with low overhead would improve performance in many cases. Finally, it proposes a centralized block memory operation handler that can better consider all of the information available to the operating system when making decisions about the correct algorithm to use.

1.2 Organization

The remainder of this thesis proceeds as follows:

The next chapter discusses previous work in studying and improving the performance of block memory operations. Caches have been shown to provide significant benefit to the memory system, but the behavior of block memory operations do not always work well with the temporal nature of cache algorithms. Prefetching is essential to reduce the bottleneck of block memory operations, but unnecessary or useless prefetches can degrade overall system performance. Many of the algorithms in this thesis are adapted from existing copy routines, and they are discussed in Section 2.1.3. Finally, previous work has demonstrated the improvements in block memory operations that are possible in networking, such as direct cache access, TCP Offload, and zero-copy schemes.

Chapter 3 shows the frequency and breakdown of block memory operations in the kernel. It then discusses the behavior of block memory operations in the kernel and illustrates how data moves between applications and the operating system using block memory operations.

Chapter 4 describes the behavior and performance of efficient block memory operations. It discusses the different algorithms used in optimizing kernel block memory operations and how they interact with the hardware. Finally, the chapter concludes by illustrating the differing performance of block memory operations depending on cache state, data reuse patterns, and processor architectures.

Chapter 5 demonstrates how to use information about the current state of the system in

order to optimize block memory operation performance. By considering only a single cache line for an arbitrarily large region of memory, an accurate predictor of cache residency can be used to dynamically select the appropriate copy algorithm. This further motivates a hardware improvement that can lower the overhead of probing the current system state.

Chapter 6 proposes a hardware improvement to accelerate block memory operations. By augmenting the existing memory controller, the processor and cache can be freed from some of the performance issues associated with block memory operations. This hardware modification increases the functionality of the memory controller and provides a means of moving data around the system asynchronously from other processing.

Chapter 7 discusses improvements to the networking sub-system and their effect on block memory operations. TCP Segmentation Offload and Remote DMA are both algorithms that reduce or eliminate kernel block memory operations when moving data from the application memory to the network.

Finally, Chapters 8 and 9 conclude the thesis with a discussion of future work and a summary of the important contributions of this thesis.

Chapter 2

Background and Related Work

Previous research has demonstrated frequency and potential performance impact of block memory operations. This chapter introduces the related work associated with block memory operations. Caching and prefetching affect the performance of block memory operations, and both have been shown to have non-ideal behavior when interacting with block memory operations. Previous implementation decisions regarding existing copy algorithms have shown that software optimizations for block memory operations have not made their way into current systems. Finally, past research on networking has attempted to eliminate or reduce the performance impact of block memory operations.

2.1 Kernel Block Memory Operations

Previous researchers have observed the poor cache behavior of the operating system. 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. However, they did not determine the contribution of memory copying and zeroing to the operating system's miss

rate separately from other memory accesses.

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 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.

2.1.1 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 [9, 33, 41, 43]. In order to improve cache performance for Lisp programs, Peng and Sohi [33] advocated an ALLOCATE instruction which allocates a line in the cache without fetching it from memory. Both Wilson *et al.* [41] and Diwan *et al.* [9] 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 [17]. For handling write misses, he came to the same conclusion as Diwan *et al.* [9]: 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.

2.1.2 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 [4, 24, 32, 1, 36]. 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 [38], 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 [37, 38] and *stream buffers* [18], 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 [34]. It is possible to adaptively determine in hardware the amount of sequential prefetching to be performed [8, 22]. However, most of the performance benefit is lost due to the overhead of calculating the optimal prefetch strategy and the time it takes to adapt to the current application.

To achieve the accuracy of software prefetching and the low overhead of hardware prefetching, integrated hardware/software prefetch engines are necessary [10, 6, 39, 40]. Such schemes may possibly 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.

2.1.3 Existing Copy Algorithms

Existing copy routines in common use do not follow the model of optimized copy routine of Figure 4.1 exactly. Instead, unoptimized copies, string copies, or MMX copies are often used. The C library implementation of memcpy() copies data 32 bits at a time in a simple C language loop. This routine does not use prefetch instructions and is implemented using temporal stores. Because this copy routine is not optimized for any particular archi-

tecture, its performance is not optimal for all situations. The FreeBSD and Linux kernel use an optimized assembly language copy routine implemented with IA32 string move instructions without prefetching. This routine has lower overhead than any of the optimized copy routines and also performs as well as any block memory algorithm that uses temporal stores and no prefetching. The hardware-optimized string move instructions make this copy routine superior for small in-cache copies. Finally, the Linux kernel also employs MMX instructions, if the architecture supports them, in order to take advantage of the 8-byte loads and stores. This routine uses a series of non-blocking pre-fetch instructions to fetch six 64-byte cache lines before it begins the copy. It then performs the copy 64-bytes at a time, pre-fetching the next cache line before performing the copy on the current 64-bytes.

In each case, the decision on which copy algorithm to use is determined at compile time based on the architecture of the system. The operating system then executes the same statically-selected copy routine for each call of that function. Since the most-prevalent, but far from only, access pattern of data is one of temporal reuse, currently no operating systems compile non-temporal stores into their block memory routines.

2.2 Networking

2.2.1 TCP Offloading

Bilic et. al. [2] demonstrated a methodology of offloading a portion of the TCP processing to the network interface, called Deferred Segmentation. This algorithm allowed the operating system to create large TCP frames which were processed by the network interface

into multiple smaller frames. Though similar to TCP Segmentation Offload (TSO), their algorithm maintained connection state meta-data in order to collect acknowledgments from the receive system and send only one ACK back to the host system for processing. This requires significantly more processing capability on the network interface, and no current implementations are available on the market. This algorithm should show similar performance to the TSO algorithm, with reduced time spent processing ACKs. However, their paper does not show any performance numbers.

TCP Offload Engines, as proposed by the 10GbE Alliance [42], further attempt to move TCP/IP processing tasks to the network interface. Here, entire portions of the network stack are pushed onto the card, including connection setup, tear down, and flow-control. Because much of the TCP/IP processing would be handled by the NIC, this improvement should reduce the computational burden on the host CPU. However, the effect on block memory operations is indeterminate, and the overall effect on system performance may be small due to the memory bound nature of networking. In addition, the authors point out many of the difficulties in moving operating system functionality to an asynchronous device, such as communication between the host system and NIC, the significant processing and memory requirements of the network interface, and additional complexity in firmware and drivers.

Hyong-Youb Kim of the Rice Computer Architecture Group described a mechanism to perform TCP Offload by means of connection handoff to the network interface [23]. In this algorithm, the operating system is responsible for initiating the TCP connection between two endpoints. Once the connection is established, the kernel passes the responsibility

of connection management, flow control, and acknowledgment processing to the network interface. This reduces the complexity required of the network interface compared to other offload algorithms and improves the performance of the operating system by eliminating much of the computation required for TCP/IP processing.

2.2.2 Zero-Copy I/O

Zero-Copy I/O is a methodology to eliminate the copy between the application and kernel when performing I/O [7]. It requires the application to create data that is aligned on an integral number of pages. These pages are then remapped using the Virtual Memory system into the kernel memory space, where the operating system network stack creates headers and sends them to the network interface. Any fragments of data that are not aligned are processed using normal application-to-kernel copy semantics (see Chapter 7). Additionally, the application must be prevented from writing to these buffers for some time, as the network stack must retain them for possible retransmits as per the TCP/IP protocol.

Zero-Copy I/O works well when the data being transmitted is static in nature so that the application can create the aligned pages required by the algorithm. However, dynamically generated content poses a difficult problem for page alignment. Copying data in user-space to align buffers and use zero-copy is self-defeating, so this type of content typically uses the single-copy networking pathway. Finally, application programmers must write to differing APIs when using zero-copy and single-copy, complicating the task of obtaining performance and reliability from web-server software.

2.2.3 Asynchronous Data Movement

Recently, Intel engineered a protocol to move networking data directly from the network interface to the highest level of processor cache [12]. Because of the statically predictable behavior of networking memory accesses, systems using Direct Cache Access (DCA) explicitly move data for networking headers, payload, DMA descriptors, and notification messages into the cache. This modification to the standard DMA engine and coherence protocol has particular benefit for TCP/IP receive, as it can eliminate many of the compulsory misses required to process data placed in main memory by the network interface.

User-level DMA [3, 27] is an attempt to provide explicitly provide asynchronous DMA transfer facilities to application programmers. By initiating the DMA transfer in user-space, these mechanisms remove the overhead of involving the operating system to set-up, execute, and notify the user of the DMA. User applications are prevented from directly passing physical addresses to the DMA engine to prevent memory protection violations. Instead, shadow addressing or proxy addressing is used to map stores to certain virtual addresses into stores to physical addresses that the DMA controller interprets as arguments to a DMA transfer. Notification is provided via a memory-mapped status register that must be read by the user application before using memory involved in the DMA transfer. Marakatos' approach uses the PAL mode of DEC Alpha processors to execute setup code for user DMA, which allows the addition of user-level DMA without kernel modification [27]. Blumrich's approach requires modifications to the kernel context switch handler to initiate the user DMA transfer and to perform correctness actions, in this case, squashing any currently ex-

ecuting DMAs on a context switch [3]. User-level DMA allows asynchronous copying of data, but does not allow any other context to run on the processor while another context's DMA transfer completes. This requires that an application programmer explicitly program application code to overlap computation with asynchronous DMA transfers.

2.3 Summary

This chapter demonstrates the many previous innovations to improve the performance of block memory operations in the operating system kernel. However, current systems do not take advantage of the available optimizations and instead use a statically selected method to perform all block memory operations. This decision, while greatly simplifying the task of writing block memory operation code, ensures that the performance of block memory operations cannot reach an optimal level. As the rest of the thesis will show, the varied behavior of block memory operations argue for an adaptive approach to optimization.

Chapter 3

Block Memory Operation Behavior

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 [20] and Chat [15] benchmarks, as shown in Table 3.1. These block memory operations account for up to 58.7% of the applications' total execution time.

These operations typically exhibit worse cache behavior than other operations, making them relatively more expensive to execute. The significant fraction of bytes copied in these applications cause cache misses. In these applications, 89–92% of the data being copied is not found in the cache, and 82–88% of the target locations of the copies are not in the cache at the time of the copy.

Though hardware prefetching can attempt to bring some of this data into the cache before it is needed, one of the main reasons why block memory operations are such a large fraction of the total cycles spent in the kernel is their 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 3.1 shows that microbenchmarks targeted

Table 3.1 : Kernel copy profiles.

		FreeBSD Compile	PostMark [20]	bw_pipe [30]	Netperf [14]	Chat [15]
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	76.1	64.3
	Total	3.1	51.5	74.0	73.8	58.7

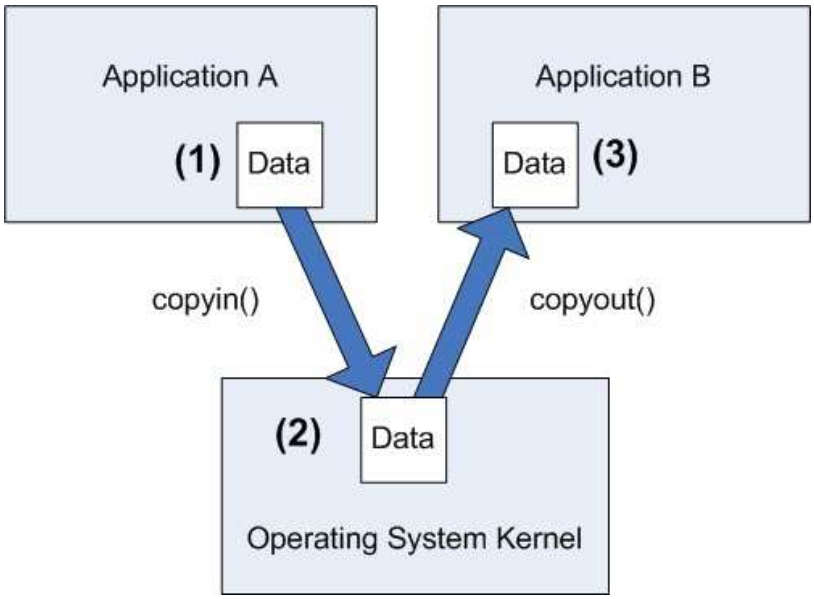


Figure 3.1 : Communication Between Applications Using Kernel Pipes

at these operations, bw_pipe [30] and Netperf [14], are dominated by kernel copies.

3.1 Pipes

Figure 3.1 shows a simple example of block memory operations within the operating system kernel when two applications communicate using a pipe. Application A begins by creating some data that it wishes to send to Application B. It calls the operating system to

transfer this information using the write() system call at point (1). The kernel performs a copy to transfer the data into the kernel's memory space using the copyin() function at (2). Finally, Application B requests the data using the read() system call and the data is copied to Application B's memory space using the copyout() function. The two applications are able to communicate using an operating system-provided pipe. By requiring the intervention of the kernel, the operating system prevents the application from writing directly into the memory space of another application. Of the benchmarks considered in this work, two commonly exhibit this type of behavior.

The FreeBSD Kernel Compile uses operating system provided pipes to store the intermediate files generated by the compiler after successive passes through the source code. Due to the large code base being compiled, a significant amount of these temporary files are created and stored in kernel pipes until the next compiler iteration reads them. This results in some degree of cache thrashing, where many of the reads and write are cache misses due to the capacity of the cache being unable to hold all of the data.

The bw_pipe benchmark from the LMBench suite attempts to saturate the system by reading and writing a fixed message size of 64KB as fast as possible between a writer and reader thread. This results in a significant number of block memory operations performed between kernel and user space. However, because the message can fit comfortably in the cache, most of the access are cache hits.

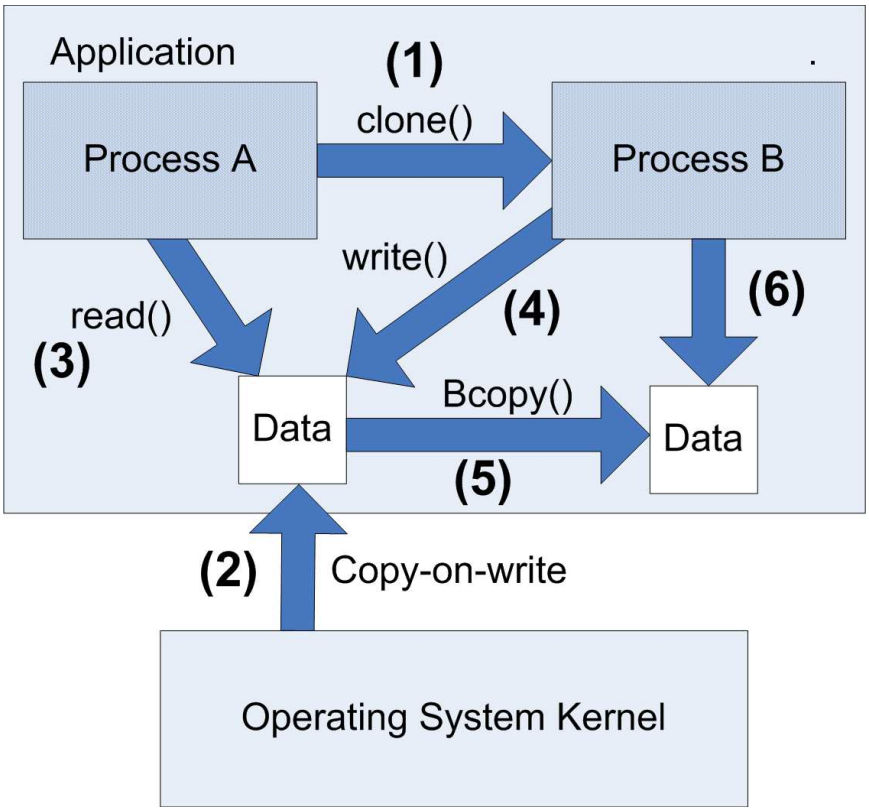


Figure 3.2 : Block Memory Operations in Interprocess Communication

3.2 Process and Memory Management

Figure 3.2 demonstrates some common block memory operations performed by the operating system during process and memory management. At point (1), the application decides to make a copy of one of its processes using the clone() or fork() system call. This requires the operating system to perform a memory copy to recreate the process and provide it with all of the appropriate descriptors used by the parent. In order to minimize memory use, shared structures are not immediately copied, but are marked copy-on-write by the operating system at (2). Thus, when Process A reads the data at (3), the shared copy of the data

is accessed and no additional memory is required. However, when Process B attempts to write to the data at (4), the operation cannot proceed as the write may cause an incorrect value to be read by Process A at a later time. The memory system, having marked the data as copy-on-write, throws an exception which is handled by the operating system. The data is copied by the kernel to a new location in the application space at (5) and Process B's descriptors are updated to reflect the change. Control then returns to Process B to complete the write at step (6). Because the operating system is responsible for the management of the virtual memory system and the scheduling of processes in the multiprogram environment, the kernel is required to perform a series of block memory operations to ensure consistency of data between processes and applications.

The PostMark file system benchmark exhibits this behavior in the buffer cache when sharing files between processes. It begins by creating 50,000 files located in 100 directories, with each file ranging from 1-10KB in size. Next, PostMark forks a series of threads which begin randomly creating, appending, and deleting files to simulate the behavior of a mail-server on the file system. As the threads interact with the files in the buffer cache, data structures must be shared between threads, processes are forked and reclaimed, and new pages must be created and zeroed for use. All of these actions require the kernel to perform block memory operations. Since the working set of the buffer cache far exceeds the capacity of the processor caches, some of the data that is the destination of a block memory operation will not be reused before cache eviction.

The Chat benchmark clones processes and data structures in a similar fashion, in com-

bination with its network-intensive behavior. A separate process is created for each room on the chat server, and it is responsible for receiving messages from the chat clients and forwarding each message to all of the other client nodes in a given room. The mix and behavior of block memory operations occurring in the Chat benchmark vary greatly over the course of the benchmark run and does not lend itself to static prediction.

3.3 Summary

In the examples in Figures 3.1 and 3.2, the data movement is predictable from one location to another. The source data for the block memory operations is frequently cached, and the destination data is very likely to be used again by the second application, child process initialization, or a read or write performed on a shared data structure. Current operating system implementations perform well in this case, but do not optimize performance when the block memory operations exhibit differing behavior.

Chapter 4

Block Memory Operation Performance

Block memory operations within the operating system can consume a significant fraction of kernel execution time. These operations, including page zeroing, memory copies, networking, interprocess communication, and copying between user and kernel space, cannot easily be eliminated because they are often critical for communication and protection within the system. This chapter illustrates the behavior of copy operations that constitute the large majority of kernel block memory operations. It describes some common implementations of these copy algorithms and how their performance is affected by cache state and data reuse.

4.1 Memory Behavior of 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 4.1. 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 4.1, 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.

4.2 Copy Algorithms

To better understand the behavior of block memory operations in the kernel, a series of experiments performing block memory operations was devised. There are two main axes

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

Figure 4.1 : Efficient block copy algorithm.

along which the copy algorithm presented in Figure 4.1 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.

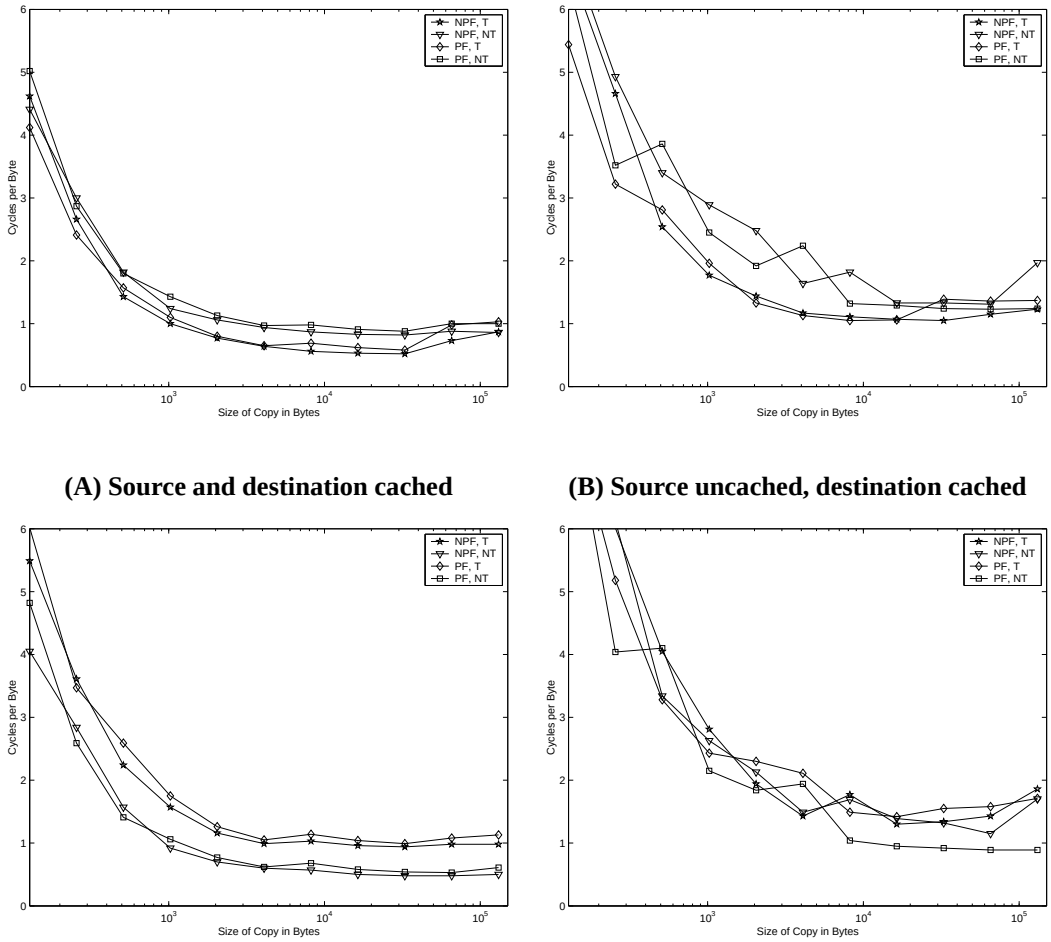
Second, the store instructions can either leave the destination data within the cache or they can provide non-temporal hints. If the destination data is not already in the cache, it may improve performance to write directly to memory, eliminating the need to first pull in data from memory that is not 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 4.1 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 `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.

4.3 Copybench

A simple microbenchmark illustrates the performance of the copy routine shown in Figure 4.1. Figures 4.2 and 4.4 show the results of this experiment. 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



(A) Source and destination cached

(B) Source uncached, destination cached

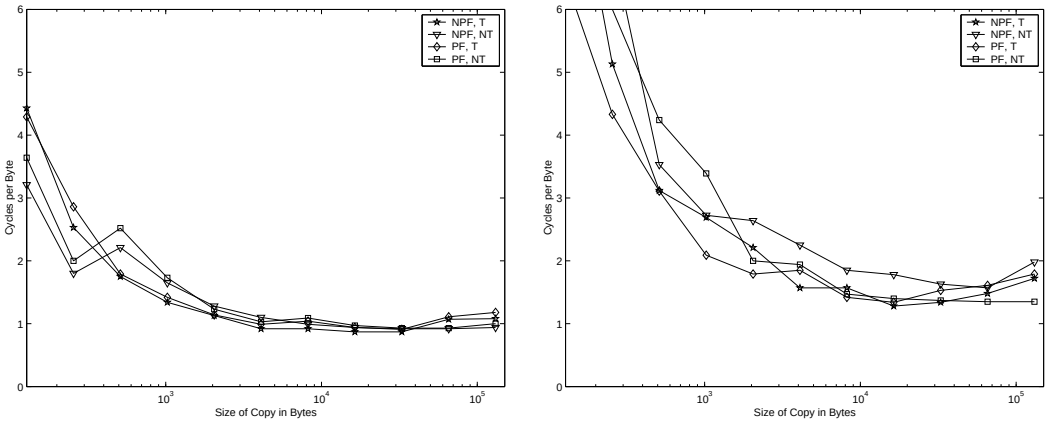
(C) Source cached, destination uncached

(D) Source and destination uncached

Figure 4.2 : Opteron Copy performance (cycles per byte copied) for different copy sizes when the source and destination are cached or uncached.

provides important performance information about each of the copy algorithm optimizations as the cache state and copy size varies.

Figure 4.2 shows the performance of the copy routine described in Figure 4.1 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



(A) Source cached, destination uncached **(B) Source and destination uncached**

Figure 4.3 : Opteron Copy performance (cycles per byte copied) for different copy sizes when the destination is uncached and the target data is read after the copy.

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 4.1 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 illustrate three key concepts 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 4.2A 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. In contrast, 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 when both regions were cached.

Figure 4.3 illustrates how the behavior of the copy algorithms differs when the destination of the copy is read immediately after the copy completes. Figure 4.2 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 4.3. 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 figures indicate, 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 4.2C and 4.3A 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 the best for all but the smallest and largest copies.

These tests were repeated on an AMD Athlon processor running at 1.53 GHz and using 266MHz DDR Memory. Though the nominal performance was significantly worse due to the lower clock speeds of the processor and memory, the performance trends were nearly identical. This argues that the effects of prefetching and non-temporal store instructions on kernel block memory operations are largely independent of the specific processor architecture.

4.4 Copybench and the Pentium 4

Figure 4.4 shows the performance CopyBench on a Pentium 4 3.2 GHz processor with 400 MHz DDR Memory. Like the graphs for Opteron Data, the figures show the performance for the different possible copy algorithm optimizations such as prefetching (PF) and non-temporal stores (NT). However, these graphs show an additional optimization, that of using string move instructions (labeled 'WC' for copies of the granularity of one word) or cache line size groups of loads and stores (CL). The inner loop of the copy can perform the copy one word at a time, or it could be unrolled. Unrolling the loop so that a single cache line is copied each iteration may be appear to be the obvious choice, but modern processors are optimized when performing string move instructions to amortize loop overhead (counter decrement and break on zero) by using a single fused instruction (movsb, movsl, and movsq for 1B, 4B, and 8B copies, respectively). Because this optimization did not have a significant effect on performance for the Opteron Data presented in Figure 4.2, it was omitted for the sake of clarity.

Overall, the performance trends seen here are similar to those of the Opteron. Prefetching improves performance when the source data is uncached, and non-temporal stores improve performance when the destination is uncached. The performance of block memory operations when the destination is read after the copy (not shown for the P4) is similar to that of Figure 4.3, where temporal stores are much faster.

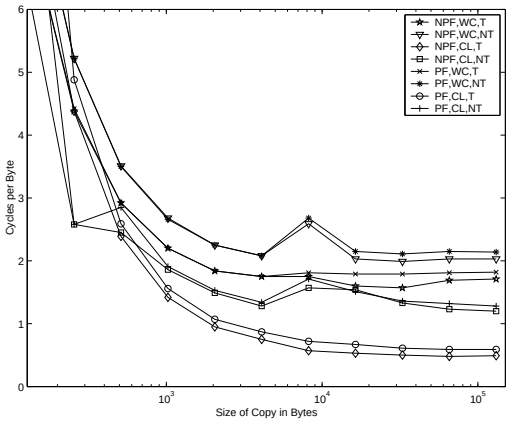
However, there were a few differences to note. The cache-line blocked copies (CL) showed better performance for each of the four cache states as shown in Figures 4.4 A,

B, C, and D. Non-temporal stores utilizing word-sized copy loops performed the worst in all cases. For Figures 4.4 A, B, C, and D, there is an anomalous behavior at 8KB copy size. This is due to an interaction with the L1 Data Cache of the Pentium 4, which is exactly 8KB in size. Figure 4.4 E shows the behavior of the benchmark when the source and destination are both cached, but removed from the L1 cache and into the L2 cache. The data is much smoother, and does not show any anomalies at 8KB. Finally, Figure 4.4 F shows the behavior of non-temporal stores in the Pentium 4 for dirty data. The source and destination are cached in the L2 as in Figure E, but the destination data is dirtied (modified state for multiprocessor cache coherence). Here, we see that the algorithms that use temporal stores all perform equally well, but the algorithms that utilize non-temporal stores perform particularly poorly. This is because the non-temporal store implementation of the Pentium 4 forces the dirty data to first be written back to memory before performing the store operations. This result means that there is a large performance penalty for using non-temporal stores on cached data that is dirty. This same test did not yield meaningfully different results on the Opteron, suggesting that the implementation of non-temporal stores does not suffer from the same problem on the Opteron.

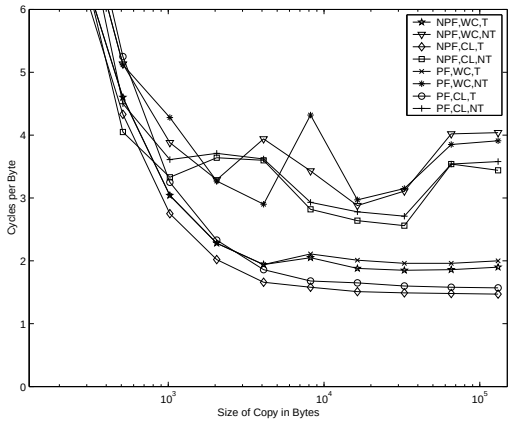
4.5 Summary

The performance of block memory operations is heavily dependent on the cache state, copy algorithm, and reuse pattern of the data. Optimizations such as prefetching, non-temporal stores, optimized copy loops, and hardware string move instructions can have a significant

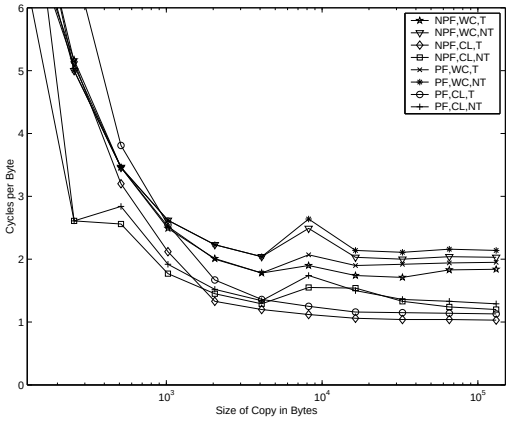
effect on performance, both positive and negative depending on the situation. The variety of processor architectures available today have similar performance trends when executing block memory operations, and the optimizations discussed in this chapter can applied to many systems almost independent of the underlying architecture.



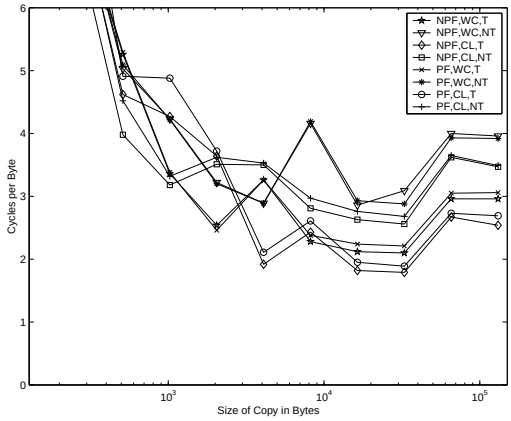
(A) Source and destination cached



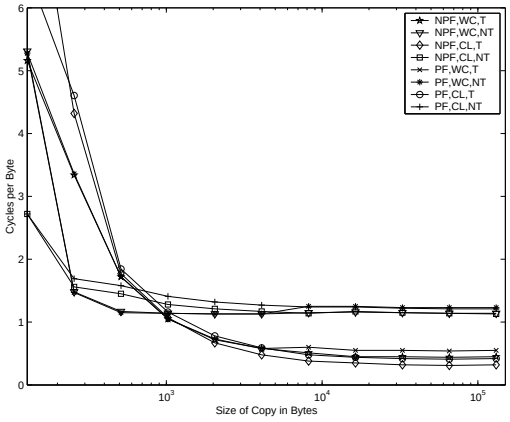
(B) Source uncached, destination cached



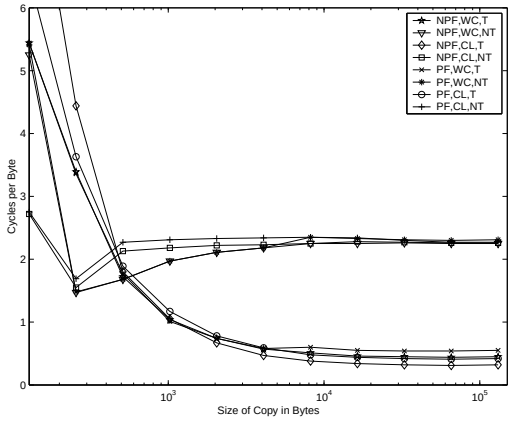
(C) Source cached, destination uncached



(D) Source and destination uncached



(E) Source and destination cached in L2 only



(F) Source and destination cached in L2 and Dirty

Figure 4.4 : Pentium 4 Copy performance (cycles per byte copied) for different copy sizes when the source and destination are cached or uncached.

Chapter 5

Algorithm Selection

The data from the previous two sections supports the proposition 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 5.1 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.

5.1 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 could involve accessing large amounts of data. Fortunately, 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. The information

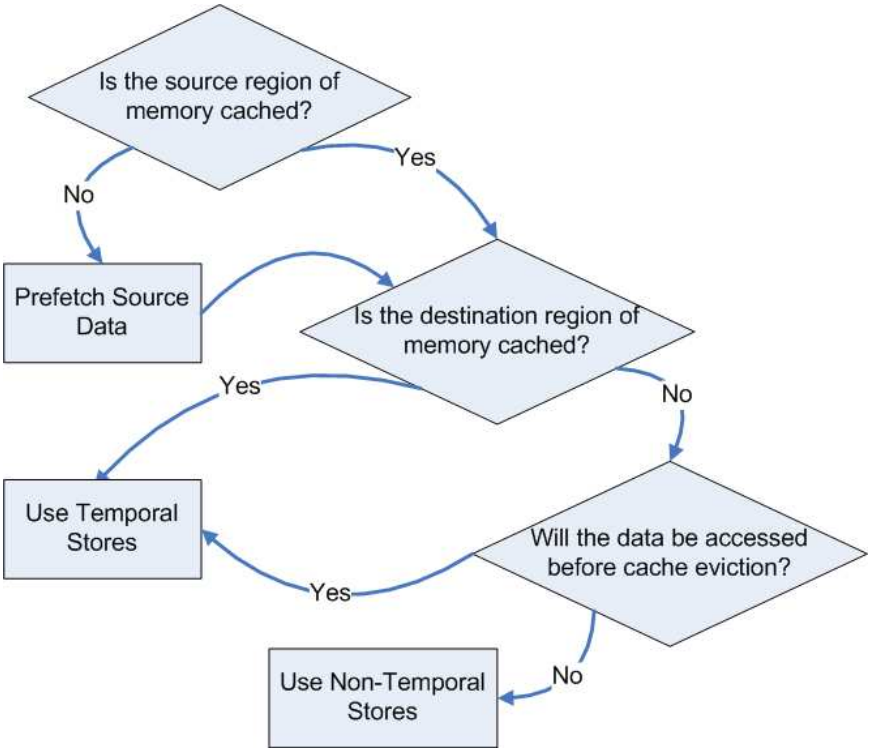


Figure 5.1 : Method for dynamically selecting of the appropriate copy algorithm.

in Tables 5.1 and 5.2 demonstrate 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%

Table 5.1 : 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 Size	**FreeBSD Compile**			**PostMark**			**bw_pipe**		
	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
256 B	12433	95.8	43.1	2220	98.6	46.2	6	100.0	31.3
512 B	12666	99.2	82.7	3721	98.2	99.8	13	100.0	100.0
1 KB	8021	99.5	97.3	3696	100.0	92.3	0	—	—
2 KB	9890	98.1	96.9	9498	99.9	96.8	0	—	—
4 KB	65282	91.2	92.3	17253	97.6	97.3	170	61.8	96.4
8 KB	12410	99.6	91.4	20908	99.9	94.3	0	—	—
16 KB	18658	91.0	90.3	10430	99.7	71.4	1	100.0	100.0
32 KB	22	100.0	94.6	0	—	98.7	0	—	—
64 KB	17	100.0	100.0	0	—	—	1125555	58.4	99.3
Total	**1355082**	**99.3**	**97.9**	**712543**	**99.8**	**99.0**	**200554**	**99.9**	**99.9**

Table 5.2 : Probe accuracy by block size for the Networking Benchmarks.

Block Size	**Netperf**			**Chat**		
	Blocks	**Src**	**Dst**	**Blocks**	**Src**	**Dst**
128 B	660667	99.9	99.9	116394	99.9	99.9
256 B	140	72.9	21.8	36651	99.8	82.5
512 B	853	95.1	99.8	24759	98.9	99.4
1 KB	0	—	—	5516	89.3	94.7
2 KB	1953838	99.9	50.0	785462	93.2	96.5
4 KB	2159	82.1	99.4	48940	91.9	97.3
8 KB	0	—	—	23028	37.8	11.0
16 KB	83	100.0	100.0	15710	67.7	5.1
32 KB	0	—	—	75	100.0	60.0
64 KB	0	—	—	0	–	–
Total	**2617740**	**99.9**	**67.9**	**1056547**	**92.6**	**93.3**

of the time for PostMark. As the table shows, the cache state of the first word 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, that information can be used as a very accurate

Table 5.3 : Percentage of cache lines that are the destination of a block memory operation that are reused or unused before eviction from the cache

Benchmark	Reused	Unused
FreeBSD Compile	97.3	2.7
PostMark	89.5	10.5
bw_pipe	99.9	0.1
Netperf	51.3	48.7
Chat	73.3	26.7

predictor of the cache state of the entire block, allowing the optimal copy algorithm to be selected.

5.2 Predicting Data Reuse

Determining whether 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 5.3 shows that significantly more than half of the cache lines that are targets of block memory operations are reused before eviction from the cache. A common behavior in kernel block memory operations is the overwriting of one block memory operation by another before the first has been evicted from the cache. System memory is often allocated and reused in socket buffers used for networking, kernel buffers used in pipes, and the buffer cache for file system I/O. 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 in the Netperf and Chat Benchmarks, a good portion of the data is not

reused before eviction. 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 or read by the system after the application initiates the send system call. This reduces cache pollution by not occupying space in the cache before the network interface initiates a DMA transfer to move the data over the PCI bus.

Another opportunity to use non-temporal stores is when 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.

5.3 Software Cache Probe

Section 4.1 motivates the necessity of determining the cache state in advance of the copy in order to select the appropriate variant of the copy algorithm given in Figure 4.1. Because we only need to consider a single cache line to accurately determine the cache state of a larger region of memory, this thesis proposes an online method to probe the cache and determine if a word is resident.

To determine the location of a region of memory, the time it takes to complete a load is a reasonable metric to predict an element's location within the memory hierarchy. 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

```
...
  sfence                ; Fence: complete all memory operations
  rdtsc                 ; Read time stamp counter
  movl 0(%esi), %ebx    ; Load from the source
  movl %eax, %ebx       ; Save the lower bits from the rdtsc
  sfence                ; Fence: complete timed memory operation
  rdtsc                 ; Read time stamp counter
  subl %ebx, %eax       ; Subtract the two time stamp values
  cmpl THRESHOLD, %eax  ; Compare to cache hit time threshold
  jb HIT                ; Jump if cache hit
...
```

Figure 5.2 : Software cache probe routine (IA-32 assembly).

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. Figure 5.2 shows an x86 assembly implementation of a software cache probe.

Memory fence instructions function to ensure that we are timing the appropriate memory reference. 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 requires that the processor's pipeline be flushed of in-flight instructions and that all loads and stores are retired. 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 creates a phase of processor wind-down as the CPU completes outstanding memory references and empties the pipeline, a period of slow operation as the processor issues the timed load and waits for its return, and a wind-up phase as the processor begins to fetch and issue instructions to refill the pipeline and continue out of order execution. Though it is difficult to measure the performance impact of the wind-down and wind-up phases of the software probe, the inclusion of the algorithm in a working kernel suggests significant slowdown. Preliminary results for an implementation of this software algorithm in a FreeBSD kernel showed a significant performance improvement for only the Netperf application as shown in Table 5.4. This is because the non-temporal stores used for the copyin() prevent the network data for polluting the processor cache in addition to executing faster than temporal stores as shown in Figure 4.2 C. The second column shows the minimum exposed latency of the probe as a percent of the total runtime. This is the percent of the execution time we would expect to recover if the probe had no exposed overhead. However, it does not account for the serialization of memory accesses and pipeline flush which significantly contribute to the overhead of the software probe.

The FreeBSD Kernel Compile showed a 1% performance degradation, but the kernel profiler recorded a 2.4% overhead caused by the exposed latency of the software probe.

Table 5.4 : Performance of the Software Probe Algorithm

Benchmark	Normalized Runtime	Minimum Probe Latency (%)
FreeBSD Compile	1.01	2.4
PostMark	1.03	0.22
bw_pipe	1.09	3.9
Netperf	0.29	3.4
Chat	1.12	6.2

If this overhead could be reduced, and the behavior of the software probe strongly suggests this, then this application should see at least a 1.4% performance improvement when the latency of the probe is overlapped with other execution using pipelining and other latency-hiding techniques common to modern microprocessors. Each of the other applications showed a slight performance penalty in overall execution time because of the large overhead of the software probe mechanism, and the minimum overhead of the probe was smaller than the measured overall application slowdown. Because of the difficulty in estimating the true overhead of a software probe mechanism due to the significant overheads involved, the preliminary results can only be considered inconclusive for these benchmarks.

5.4 Hardware Cache Probe

The high overhead of a software based cache probe motivates the addition of a new hardware cache probe instruction. A cache probe instruction does not need to be complicated to be useful. In its simplest form, such an instruction returns 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 DRAM to retrieve the data. This access to DRAM when the block of memory being probed turns out to be uncached is the primary 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 as short as the time to access the tag array alone. A hardware cache probe instruction 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. When implemented in hardware, this instruction requires no timing information or memory-fence and would not incur many of the overheads associated with the software probe algorithm. Additionally, the instruction could return any of the information held in the tag array, such as the LRU bits or MOESI state, along with the information of which cache level the data was located. This cache meta-data is not currently available by standard means, and could be used by a more advanced predictor in combination with other factors to determine optimal block memory behavior.

5.4.1 Hardware Cache Probe Versus Informing Loads

This hardware probe compares favorably to previous proposals that attempted to probe the memory system in an online fashion. Horowitz *et al.* [11] proposed a mechanism called

informing loads that combined a load instruction with a conditional branch. If the load hit in the primary cache, then the processor would execute the memory reference as normal and the next instruction in the sequence would skipped. If the load missed in the primary processor cache, then the next instruction, typically a jump to a different region of code or a prefetch of a location expected to miss in the future, is executed. This offers similar functionality to the hardware probe instruction, in that it can inform the programmer of an uncached location via load. However, informing loads always execute and may fetch data into the processor caches that is not needed. In the case where we want to know if a piece of data is cached, informing loads can only tell us that it *was* cached, not *if* it was cached. The hardware probe is a simpler primitive that can be used to determine if a load, store, or prefetch will miss in the cache without changing the location of that data. The hardware probe can be combined with a conditional instruction to perform the exact same behavior as informing loads, such as jumping to a different section of code when a miss is expected. Additionally, informing loads require modification of the branch-prediction logic due to their implied conditional branch. In a superscalar pipeline processor, this additional communication between the memory system and branch logic is not trivial to implement. By limiting its scope, the hardware probe instruction can accomplish significantly more functionality with minimal modification to existing hardware.

5.4.2 Hardware Probe Implementation

Probing the cache is effectively performing a memory reference without concern for the data at the address in question. The probe instruction takes the virtual address of the memory to be probed. It accesses the L1 TLB to begin the translation of the virtual address to a physical address while simultaneously indexing into the L1 Data Cache tag array. The cache compares the translated physical address to the tag in each of the cache ways looking for a hit. If one of the tags match, the instruction returns a value indicating the data was found in the L1 cache plus cache meta-data including LRU and MOESI state. If no virtual to physical translation exists or if no tag matches the physical address of the memory reference, the request is a miss and is passed to the next level of cache. The process of determining the virtual to physical translation is repeated in the L2 TLB if no translation was available from the L1 Data-TLB. The L2 cache is indexed by physical addresses, so no indexing or tag check can begin until a physical to virtual translation occurs. Once the translation is complete, the tags for each way of the cache are checked to determine if they match the address being probed, and information about the cache line is returned to the processor as before. A request that misses in the cache is forwarded to the next level of cache for probing. If the request misses at the highest level of the processor cache, the request does not go to memory and generate a DRAM or memory controller transaction. It simply returns a value to the processor cache indicating the data is uncached. If a translation is not available or if the load/store queue is full, the instruction can return a value indicating an indeterminate result without requiring an exception be thrown, a processor stall, or a TLB

fill. Such an instruction requires neither the memory serialization, pipeline flush, nor timing information required of the software probe, and thus it would have significantly lower overhead.

Chapter 6

Memory Controller Data Movement

As Figure 4.2 shows, block memory operations are most expensive when the source is not found in the processor caches. It takes almost twice as many cycles per byte for a copy in that case and, while prefetching can help in some situations, it does not eliminate the performance gap. In fact, the achieved copy bandwidth is significantly worse than the peak DRAM bandwidth despite the sequential access pattern to the DRAM. Therefore, if the fact that the source is uncached can be determined, by the previously proposed cache probe instruction for instance, then the memory controller is likely to be able to more efficiently move the data. Furthermore, with a processor copy, the uncached source data will be brought close to the core just to copy it, and it is unlikely that this data will be reused again soon, so moving it close to the core is unnecessary and wasteful. Given the recent trend of integrating the memory controller onto the processor die [21, 19], it makes sense to allow the memory controller to participate directly in block memory operations.

This thesis argues that integrating an copy engine into the memory controller will accelerate block memory operations. By performing data movement operations within the memory controller, processor cycles can be better used for other computations, cache pollution can be minimized, and block memory operation performance is likely to be limited only by the DRAM bandwidth.

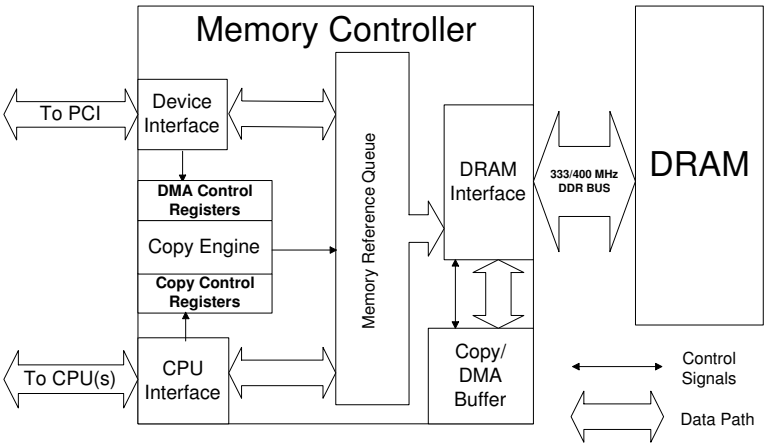


Figure 6.1 : Proposed Architecture of a Memory Controller Copy Engine

6.1 Memory Controller Copy Engine Architecture

Figure 6.1 shows the architecture of such an asynchronous copy engine. The copy engine resides within the memory controller and can be accessed through memory-mapped I/O. When the processor requests a block memory operation from the engine, it can then asynchronously perform the data transfer with no further intervention from the processor. The engine can schedule the copy optimally for the given DRAM architecture using internal storage. So, a block of the source will be read into internal buffers, which can then be written to the destination. This can be repeated until the entire transfer is complete. In the case of a zeroing operation, the destination can be written at full DRAM bandwidth with no buffering. Upon completion of the operation, the processor can either poll a status register or be interrupted by the copy engine. Since the engine works asynchronously to the processor, a queue of pending requests can be stored in the memory controller, allowing the processor to enqueue multiple data transfers. Once the queue is full, the processor would

have to wait until entries are freed when previous operations complete.

Such a mechanism is especially suitable for use by the operating system, which can easily control the use of the destination data. By switching to other tasks during the copy, the OS can ensure that the processor resources are efficiently used, and that the destination data is not consumed until the transfer has completed. Often the kernel can gracefully deal with the asynchrony of such a copy engine simply by not scheduling the application thread that will consume the result of the copy until it has completed. For example, when the kernel must copy from kernel to user space or zero a page for an application, then the kernel never touches the destination of the transfer. So, the scheduler could be made aware that the application thread must continue to block until the transfer has completed. Furthermore, the large amount of copying that occurs within the kernel, as shown in Section 4.1, likely justifies the reorganization of other parts of the kernel to accommodate such asynchronous copies if there is a noticeable performance improvement.

6.2 Copy Engine Performance

If it is known that the destination of the block memory operation is going to be used in the near future, it would also be possible for the copy engine to push the destination directly into the cache, instead of returning it to DRAM. In that case, at the end of the asynchronous operation, the destination would reside in the cache, and the source will not have consumed any cache space. This could result in significant performance improvements.

Placing the copy engine within the memory controller should allow these copies to pro-

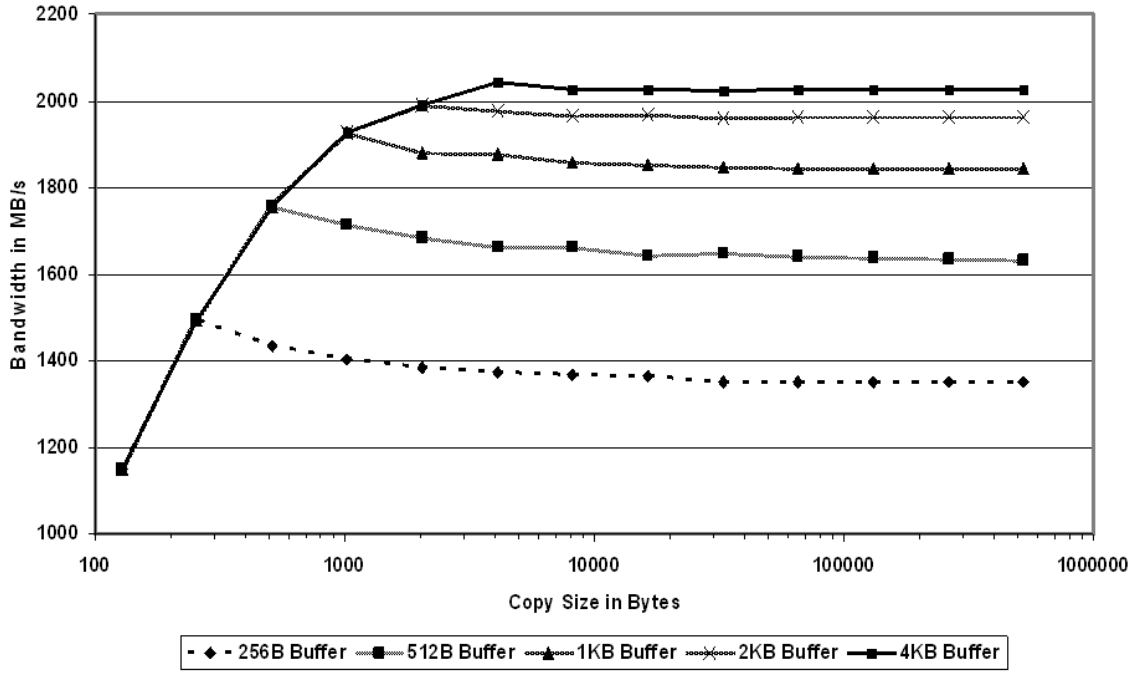


Figure 6.2 : Bandwidth Achieved by Memory Controller Copy Engine as Buffer and Block Size Vary

ceed at close to the peak DRAM bandwidth. Some preliminary experiments, using the cycle accurate DRAM simulator described in [35], show that with a 1 KB buffer in the memory controller, a copy engine could achieve over 90% of the peak bandwidth of the DRAM. The results for a variety of sizes of copies and copy engine buffer sizes are shown in Figure 6.2. This is as much as a 20% performance improvement over the best copy algorithms shown in Figures 4.2B and D. The reason a copy engine within the memory controller can perform so well is that it can schedule the source and destination copy accesses, as well as other outstanding memory accesses in the system, in such a way as to minimize row activations. Since copies access the memory sequentially, the memory controller can sustain the peak

memory bandwidth of the DRAM for significant periods of time. With double buffering, and more advanced scheduling, the performance could improve beyond what we achieved with our initial experiments.

6.3 Issues with an Asynchronous Copy Engine

However, there are three main issues that must be overcome in the design of such an asynchronous copy engine. Fortunately, the first two are easily solved in software. First, the memory controller uses physical addresses, so a single transfer cannot span discontinuous physical pages. This is easily handled by the operating system: all transfers must be broken up into individual transfers, each of which do not cross any page boundaries that are discontinuous in physical memory. Since most page-allocation systems seek to maintain physical continuity for consecutive virtual pages, this is a minimal problem.

Second, memory copies whose source and destinations overlap will have to be carefully scheduled. The operating system can ensure that data transfers are enqueued to the asynchronous copy engine in the appropriate order so as to preserve the semantics of the copy operation.

The final major complication of an asynchronous copy engine within the memory controller is cache coherence. Data movement performed by the memory controller can not ignore data that may be stored within the processor caches. Therefore, each transfer potentially requires a coherence transaction on the bus. A mechanism like token coherence [29, 28] could potentially eliminate many of these coherence transactions if the bulk

of the data is uncached, and therefore owned by the DRAM. Presumably, this would often be the case, as the engine would primarily be utilized when the cache probe yields a prediction that the data is uncached. So, the memory controller would have all the tokens for most of the cache lines, and would therefore know that it did not have to perform any coherence transactions. Furthermore, since the memory controller knows the length of the copy, it could begin requesting the appropriate tokens well ahead of time, further optimizing the copy performance in a way that would be difficult to do using conventional processor copying techniques.

6.4 Potential Uses of a Memory Controller Copy Engine

An asynchronous data transfer mechanism within the memory controller can also have many other potential uses within the system. For example, Intel has recently proposed the use of direct cache access (DCA) by peripherals [13]. While this is sometimes effective, it can also pollute the cache and degrade performance. Allowing the processor to asynchronously move data into the cache as necessary using the proposed transfer engine could possibly achieve the benefits of DCA without the drawbacks, when combined with the hardware cache probe instruction.

Another use of the transfer engine could be to allow the processor to initiate direct memory access (DMA) reads and writes to peripherals directly. In the current system model, peripherals initiate DMA accesses, forcing the memory controller to satisfy both processor and peripheral memory requests on demand. This can create conflicts in the

memory controller, leading to degraded performance [35]. If the processor were to enqueue these DMAs into the memory controller, using the copy mechanisms described here, then the memory controller would be free to schedule the transfers so as to minimize conflicts and maximize performance.

Finally, the asynchronous copy engine provides all of the facilities required for User-Level DMA, allowing application programmers to explicitly initiate intra-memory or memory-to-device transfers of data. User-level control of the copy engine frees the kernel from the burden of scheduling independent work while the asynchronous data movement is progressing. This would allow an application to perform similarly to the RDMA protocol, moving data from user space to a different application, device, or over the network without intervention of the host operating system.

Chapter 7

Networking

With the advent and rapid expansion of the Internet, network connectivity between computers has been growing at an extraordinary rate. Users are demanding a greater amount of bandwidth and network resources for services such as media streaming, network files systems, interactive content, and cluster computing. This network traffic is consuming an increasingly large fraction of the computational capacity of the computers involved as communication capacity and demand continues to increase. The copy from application to kernel memory required by the basic networking algorithm can constitute a significant fraction of total execution time. This chapter discusses various methods proposed to reduce the time spent processing for networking applications.

The basic TCP segmentation algorithm is a simple and repetitive computational task that takes the data that will be sent over the network and divides it into pieces the size of the maximum transmission unit (MTU), typically 1460 bytes for TCP/IP over Ethernet.

Figure 7.1 shows this process in a standard networking setup. The application creates the data that is going to be sent over the network and calls the operating system to handle the sending at point (1). The operating system copies this data into the kernel address space using the copyin() function at (2). The operating system network stack segments the data into MTU-sized pieces and appends a TCP/IP header onto each packet in step (3), then

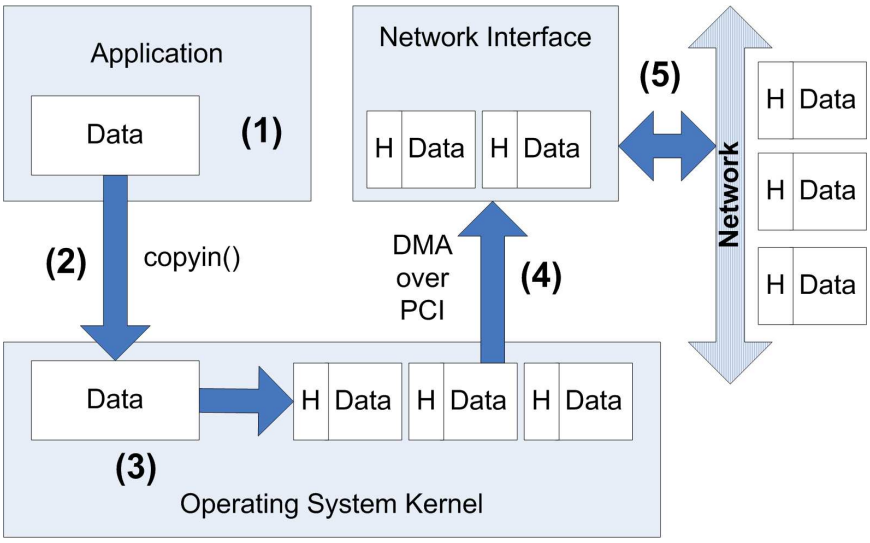


Figure 7.1 : Block Memory Operations in TCP/IP Networking

signals the network interface card (NIC) to retrieve the packets. In step (4), the NIC uses the direct memory access protocol (DMA) to fetch the packets from the kernel over the PCI bus. Finally, the NIC calculates the checksums for each packet for TCP/IP and sends the data over the network in step (5). This process is extremely memory intensive, with the kernel spending 71–73% of its time performing block memory operations and associated memory management during TCP/IP send.

7.1 TCP Segmentation Offload

TCP Segmentation Offload (TSO) reduces the load on the host CPU by requesting TCP/IP packets of a much larger size, typically up to 64KB. Figure 7.2 shows how the TSO algorithm differs from the standard TCP/IP networking approach. The application creates the packet and it is copied into the kernel in steps (1) and (2) similar to Figure 7.1. The operat-

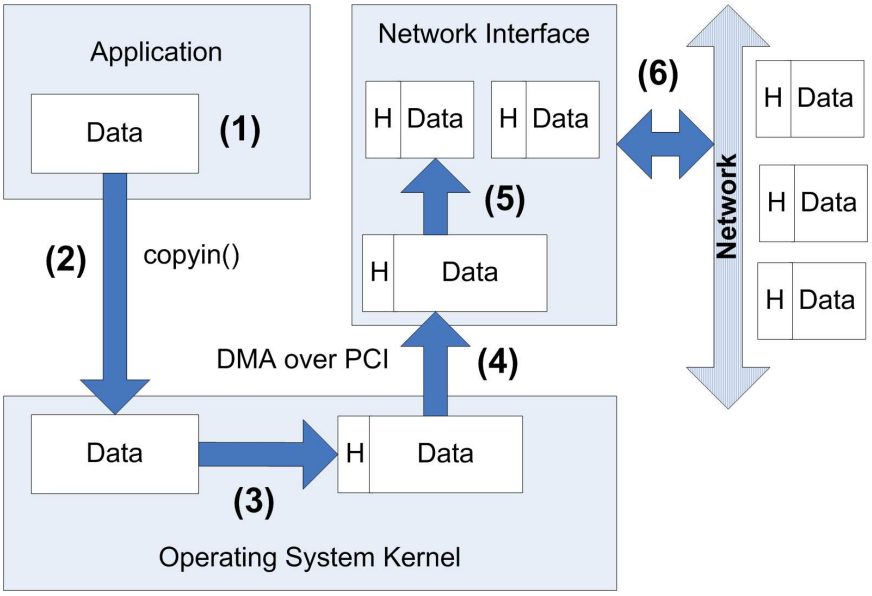


Figure 7.2 : TCP Segmentation Offload

ing system then creates a header for each 64KB packet instead of each 1460 byte packet in step (3). These much larger packets are transmitted over the PCI bus to the NIC, where they are stored in NIC memory in step (4). In step (5), the NIC divides the 64KB data segment into MTU-sized pieces and creates headers for each of the smaller segments based on the header for the 64KB packet. After calculating the appropriate checksums for each packet, the packets are sent out onto the network as normal in (6). The receive system needs to have no knowledge of the TSO behavior of the NIC and it returns acknowledgments for each packet using the standard TCP algorithm. TSO does require special functionality in the Network Interface, driver, and operating system. Currently, the Linux 2.6 kernel supports the necessary features to operate the Intel Gigabit Server NICs that are TSO capable.

Table 7.1 : Netperf and TSO.

		Execution Time	Memory Operations	TCP/IP Send	TCP/IP Recv	Other
1 KB Messages	BASE	1.000	0.734	0.164	0.040	0.062
	TSO	0.701	0.512	0.102	0.042	0.044
8KB Messages	BASE	0.851	0.610	0.114	0.054	0.073
	TSO	0.555	0.423	0.047	0.043	0.044
32KB Messages	BASE	0.758	0.554	0.106	0.041	0.058
	TSO	0.431	0.314	0.038	0.040	0.038

7.1.1 Netperf Results

Table 7.1 shows the relative performance of TSO compared to the baseline implementation for three different message sizes when measured using the Netperf Benchmark. Netperf initiates a TCP connection between two systems and attempts to saturate the network link in order to help evaluate the potential bottlenecks in the system. The data for each column is normalized to the runtime of BASE for 1KB messages. The Memory Operations column contains the execution time for both block memory operations and the associated memory and buffer management routines of the kernel. The TCP/IP Send and Receive columns correspond to the computation required for TCP/IP header creation and packet processing for sending of the data and receiving of acknowledgments. Overall, TCP Segmentation Offload resulted in a 30–43% reduction in total execution time. These gains came primarily from the reduction in memory operations by 30–43% and TCP/IP Send processing by 38–64%.

Figure 7.3 demonstrates the effect of TSO on bandwidth and CPU utilization. In this graph, “TSO Off” refers to the baseline system, and the x-axis is logarithmic in order to

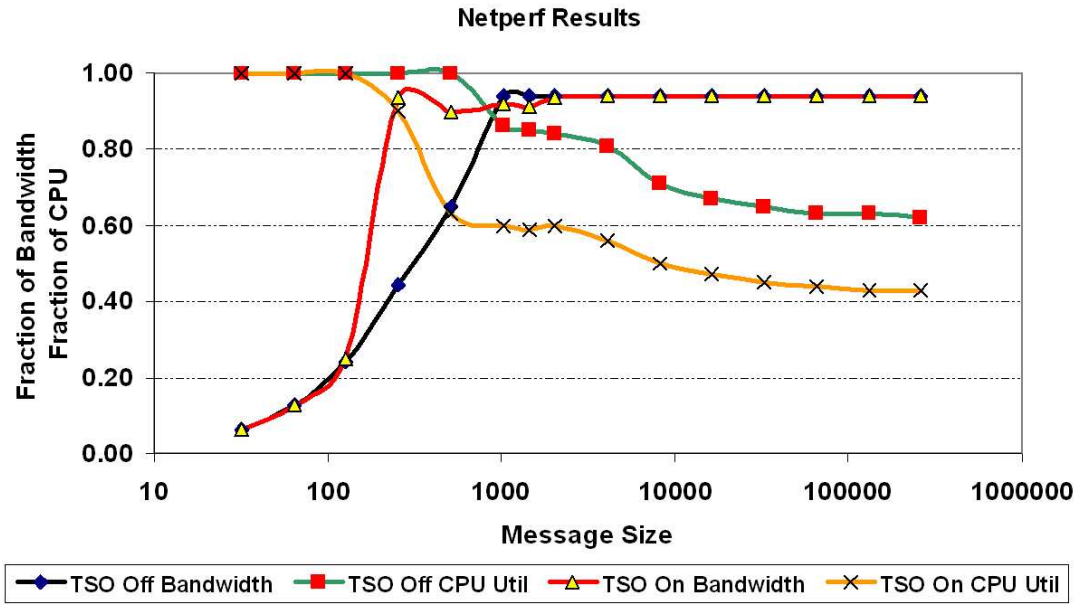


Figure 7.3 : TCP Segmentation Offload and Netperf

show the behavior across more possible message sizes. When the message size is small, both BASE and TSO saturate the CPU and are unable to achieve line-rate transmission. At message sizes between 256B and the MTU, the network driver attempts to aggregate packets together in order to maximize bandwidth. This creates packets of larger size for the network interface to segment and send on the network using the standard TSO algorithm. Though this may increase latency for small messages as the driver waits for more data to aggregate, it significantly improves performance and allows the TSO-enabled system to achieve line-rate for much smaller message sizes than the baseline system. At a message sizes of 1KB and 1460B (the max size allowed in one packet), both the baseline system and TSO are able to achieve the line-rate of the network. However, the TSO-enabled system

is able to achieve this bandwidth while using a significantly smaller fraction (27% less than base) of the CPU's execution time. As the message size increases above the MTU from 2KB and up, the benefits of TSO increase further as the algorithm is able to segment additional packets out of each message.

Initially, it was hypothesized that the performance gains from TSO would come from the decreased computational requirements in the operating system network stack [26]. Additionally, they believed that TCP/IP send was a "largely compute-bound" operation. However, this research shows that a large majority of the time required for TCP/IP send is spent performing memory operations, and that the majority of the performance gains came from the significantly increased efficiency of the routines dedicated to memory allocation/deallocation, buffer management, and page alignment. Since the operating system is dealing with far fewer data segments of much larger size, the overhead of memory management is greatly reduced. This motivates the need to group block memory operations together in order to maximize the size of the operations. Figure 4.2 shows the increased performance of block memory operations as their size grows.

The execution time spent on TCP/IP Send processing decreases due to the fewer headers required to transmit the same amount of data. As the message size increases, successively more headers must be calculated in order to send the data over the network in the baseline system. However, the TSO enabled NIC requires only one header for each data field of up to 64KB, so the benefits of TSO improve as the message size increases. These reductions in computational requirements were significantly less than the improvements in performance

of the memory system.

Both the baseline TCP/IP algorithm and TSO exhibit different block memory operation behavior than the examples in Figures 3.1 and 3.2. The kernel moves the data from the application space to the kernel memory space, where the destination is unlikely to be cached. Since the NIC calculates the checksum for the TCP/IP packet, the kernel network stack does not have to read the data into the cache to perform its portion of the processing. This means that a temporal copy of the data into the cache will likely result in the data not being accessed before eviction. This corresponds to Figure 4.2 C and D, where non-temporal stores maximized performance.

7.1.2 Apache Results

Table 7.2 shows the relative performance of TSO when applied to a Linux web-server that uses the Apache application to transfer files over the Internet using the HTTP protocol. When the files being transferred are small, as in the case with the first row of the table, the TSO algorithm is unable to improve performance. This result is to be expected for small files that approximate the HTML portion of web-pages, where the file size is not significantly larger than the MTU. When the file size is larger than the MTU, as is the case with images and graphics, TSO shows a significant benefit over the baseline system in execution time while achieving the same bandwidth. The last two rows show the effect of a mix of file sizes (between 1.5KB and 392KB) typical for a web-site, with multiple simultaneous connections simulating more than one user. In both cases, TSO shows an improvement

Table 7.2 : ApacheBench and TSO.

	Network	Connections	Execution Time	Bandwidth (MB/s)
1.5 KB File Size	BASE	1	1.00	4.3
	TSO	1	0.99	4.3
392 KB File Size	BASE	1	1.00	105
	TSO	1	0.48	104
Mix of Small and Large Files	BASE	3	1.00	92.4
	TSO	3	0.87	91.6
Mix of Small and Large Files	BASE	5	1.00	98.1
	TSO	5	0.90	98.0

of over 10% versus the baseline system, and each time maintains approximately the same bandwidth.

7.2 Remote Direct Memory Access (RDMA)

The performance of cluster computing environments has become a recent topic of interest as solutions for high-performance parallel computing move from highly-specialized, 'big-iron' shared-memory computers towards a collection of commodity low-end systems. Much of the cost savings of using commodity components for a compute cluster comes from the use of a less-expensive interconnect network. The nodes can communicate over this network to share information and synchronize computation. By using the Ethernet medium for the interconnect network, cluster computing systems are significantly cheaper than shared-memory systems. However, in order to take advantage of the cluster effectively due to the higher overhead of Ethernet traffic through the host's operating system, most programming models for parallel computing on a cluster use message passing as

the means of communication. This requires the programmer to explicitly send messages between nodes to achieve communication, instead of communication by loads and stores as in a shared memory system. Remote Direct Memory Access (RDMA) is a technique, specifically designed for cluster computing, that attempts to reduce the overhead of communicating information across the network. Research has shown that the memory-bound networking sub-system of a compute node can consume a large fraction of processor resources performing TCP/IP Send. RDMA, like TSO, improves the performance of sending data on the network by reducing the memory operation-intensive portions of the network stack.

This section compares three different methods of passing the data between nodes and evaluates their performance. The baseline method moves data through the host operating system and through the network using the current implementations of the operating system and network interface card. The second, TCP Segmentation Offload, uses a specially adapted driver and NIC to reduce the load on the CPU during TCP send. Finally, this section studies RDMA, a means to remotely initiate a data transfer between remote systems' memory without host intervention.

Message passing systems lend themselves to the RDMA algorithm because, unlike many networking applications, the receive system must be aware of which messages it requires from other nodes due to synchronization constraints. RDMA is a method to reduce the per-byte overhead of transferring data between two systems to nearly zero. To move data between two systems, the operating system of the send side is involved in tak-

ing data from the application, segmenting it into MTU pieces, and moving it out onto the network. The receive side must be involved in removing the data from the network, processing it, and returning it to the appropriate application. RDMA-enabled NICs provide a method of moving data between two systems without involvement of the operating system. The receive system directly requests data from the sender over Ethernet. The NIC intercepts the request from the network and executes DMA requests on the sender side to move the requested data into the NIC, where the appropriate segmentation and header creation occur. The data is then sent over the network to the receive-side RDMA NIC, where it is stored until it is moved using the DMA protocol into memory - often saving a copy between the operating system and application by placing the data directly in the application's destination buffer. This sequence of transfers occurs without intervention by either system's CPU, though it does require the driver to notify the operating system before and after the transfer to allocate buffers, pin pages, and handle the result of the RDMA. The RDMA-modified MPICH library adapts basic MPI calls into RDMA transfers. This decreases the overhead of communication for MPI traffic and should result in an increase in performance. However, like TSO, RDMA transfers may have higher latency than the operating system supported network transactions because the computation occurs on the much slower (when compared to the host system) NIC microprocessor and memory.

Table 7.3 : NAS 2.0 Benchmarks - MPICH MPI Library - RDMA and TSO

	bt	**is**	**lu**	**mg**	**sp**
MPICH-BASE	1.00	1.00	1.00	1.00	1.00
MPICH-TSO	1.06	.982	1.02	1.38	.988
MPICH-RDMA	.982	.293	.848	.831	.877

7.2.1 MPICH Results

Table 7.3 shows the performance of the three interconnect systems for five benchmarks from the NAS 2.0 parallel benchmark suite using the MPICH 1.2.5 Message Passing Interface (MPI) Library. Each benchmark performs a series of calculations combined with communication to solve a problem in parallel. For each of the five benchmarks, MPICH-RDMA performed better than MPICH-BASE, with improvement ranging from 18-71% and an average improvement of 23%. MPICH-TSO did not perform better on average when compared to MPICH-BASE, and only showed a performance improvement of 12–18% in the two applications in which it bested MPICH-BASE. For three of the five applications, MPICH-TSO was a slowdown when compared to MPICH-BASE, from a 2% slowdown for the bt benchmark and up to 38% slower in the mg benchmark. As discussed in Section 7.1, the TSO-enabled network driver attempts to maximize bandwidth by attempting aggregate intermediate sized messages into larger packets. This can increase the end-to-end latency of messages seen by the cluster, and MPI applications are sensitive to latency as well as bandwidth. This tradeoff for increased bandwidth and reduced execution time at the expense of latency works against TSO to degrade performance in certain cases.

7.2.2 RDMA, TSO, and Block Memory Operations

The RDMA algorithm works for three reasons. First, the data movement from the application to the network interface bypasses the kernel, saving an expensive system call and copy into kernel memory space. Though the TSO algorithm reduces the time spent in the kernel, it still must call into the kernel network stack to send data. As this research has shown, kernel block memory operations constitute a significant fraction of execution time for networking. The RDMA algorithm can bypass these copies entirely, which minimizes the necessity of involving the kernel at all.

Secondly, the semantics of the kernel TCP/IP stack on both ends, such as header creation, segmentation, queuing, reordering, and acknowledgments, are completely handled by the network interface card, significantly decreasing the latency for messages traveling from the application space of one system to the other. In this sense, it functions like a TCP Offload Engine by reducing the work required of the operating system to send data over the network. Mogul's work on TCP Offloading [31] argues that the resulting reduction in block memory operations is responsible for the majority of the gains seen by TCP Offload Engines. Latency is particularly important for cluster computing applications. As Section 7.1 demonstrated, TSO reduces the computational load on the host CPU, but does not optimize latency. This is the likely cause of the performance degradation shown for certain benchmarks using MPICH-TSO.

Finally, the decreased time spent in the kernel provides a greater fraction of compute cycles available for the application. This greater ratio of computation to communication

improves performance whenever there is enough computational work to be done, i.e. when the application thread is not blocked waiting for synchronization with other nodes.

By decreasing the computational load, lowering the overhead of communication, and improving end-to-end latency, RDMA-enabled NICs show great promise in MPI applications. However, the TSO algorithm performed poorly in these applications and is best suited for web-serving and other networking where latency is not an issue.

Chapter 8

Future Work

The optimizations for block memory operations as proposed in this thesis motivate several areas of further research into the design and use of such copy acceleration mechanisms.

1. Hardware Probe: The probe mechanism shows promise for accelerating block memory operations. Due to the high-overhead of the software probe, an evaluation of the hardware probe requires a cycle-accurate simulator that incorporates both the application code and operating system code in order to get a reasonable estimate of the performance gains possible with the hardware probe. This will verify that the applications that experienced slight slow-downs with the software-based probe mechanism and its high overhead would be accelerated on real hardware with a low-overhead probe mechanism. The response time of the hardware probe in cycles could be varied, and a tradeoff analysis could be done to determine the maximum overhead allowable for performance improvement.

2. Call-Site Profiling: Each type of kernel block memory operation (copyin(), copyout(), memcpy(), bzero(), etc) shows different behavior depending on the calling application and system state. However, there is not currently a way to determine the caller at runtime. This information can be used to selectively choose the optimal

copy algorithm, especially aiding in prediction of data reuse patterns by application. Offline application profiles can provide the kernel with accurate hints as to the reuse pattern of a given application.

3. Dynamic Call-Site Profiling: A dynamic algorithm that keeps a table of recent callers, cache state, and usage patterns would also be very valuable for prediction and optimization purposes. A Block Memory Operation Caller Buffer (BMOCB) could contain a cache of the most recent block memory operations, their callers, cache state, and a performance hint based on previous behavior. When combined with the hardware probe mechanism, these could be a powerful tool for ensuring the optimal block memory algorithm is selected each time.

4. Kernel Block Memory Operation Handler: Current systems do not attempt to centralize the handling of block memory operations in the kernel. Often times, they are broken into a series of smaller operations that are handled separately. Applications currently do not have a means of informing the kernel of expected data use patterns, forcing the kernel to determine this information itself or use a statically selected algorithm. As shown in Section 4.1, the performance of these block memory operations is highly dependent on the size of the transactions and cache state. A centralized handler will best be able to take advantage of the information available to the kernel about current system state, will maximize the performance of block memory operations by aggregating operations to execute larger block memory operations when possible, and may incorporate the information provided by a call-site profiling

algorithm to make the optimal decision about which copy algorithms to use. Additionally, a centralized handler will allow the application to request the type of block memory operation best suited for the current situation without requiring the kernel to probe cache state or consult the BMOCB, i.e. a call to copyin_PF_NT() (copy data into the kernel memory space, prefetch source into cache, and non-temporal store destination).

5. Memory controller architecture: this research suggests fully exploring the design of future memory controllers and the addition of block transfer acceleration mechanisms. The functionality of the memory controller will have to continue to improve as the gap between processors and memory widens. The design of an efficient asynchronous copy engine requires careful integration with the memory controller including efficient buffer management and DRAM scheduling.

6. Cache coherence: with this research comes the need to analyze and streamline the interactions between block memory operations, a copy engine, and cache coherence mechanisms. In order to be most effective, the cache coherence mechanisms must not interfere with the performance of the block memory algorithms or copy engine. By combining information on coherence with the hardware cache probe, the kernel can preemptively initiate coherency actions to optimize block memory transactions.

7. Communication: this paper motivates an analysis of the available asynchrony within the system and to develop efficient communication and notification mechanisms be-

tween the memory controller and the processors. Efficient communication and notification mechanisms will enable improved copy performance to translate into improved system performance. Furthermore, such mechanisms are likely to be useful for other communications between the memory controller and the processor as further enhancements are integrated into the memory controller. The overhead of asynchronous memory operations is minimized when there are efficient means of communicating the behavior of the memory copy engine to the main processor.

Chapter 9

Conclusions

Block memory operations occur frequently in the kernel when it is performing a variety of common system tasks, such as networking, interprocess communication, and I/O. The performance impacts of the memory system require the kernel to account for the location of both the source and destination regions of memory, the size of the copy, and the probable reuse pattern of the data to be able to select the optimal algorithm for these block memory operations.

The performance of block memory operations are particularly important in networking, in which 75% or more of total execution time can be spent performing block memory operations. Optimizations such as TCP Segmentation Offload and Remote DMA can improve performance by reducing the time spent performing these block memory operations.

In order to determine the optimal block memory algorithm for a given situation, 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. This unexpected result allows a simple cache probe mechanism to dynamically determine the current cache state without requiring access to each word or line in a region of memory. A simple software-based approach can perform this determination and, when combined with an algorithm selection mechanism, improve performance in certain cases.

The significant overheads associated with memory timing and serialization required by the software probe mechanism makes it 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 thesis motivate additional research into the feasibility and benefits of a hardware cache probe mechanism. The hardware probe would be trivial to implement in modern processor architectures and would provide useful dynamic information about the current system state.

The frequency of block memory operations, combined with the potentially undesirable behavior of block memory operations where the source and/or destination are uncached, motivates the addition of an asynchronous memory copy engine that can handle block memory operations without the intervention of the processor or unwanted interaction with the processor cache. The copy engine uses its knowledge of the current DRAM state to maximize the performance of block memory operations by optimally scheduling them with other system traffic. Additionally, it allows the processor to execute independently of the block memory operations, leaving more cycles available for computation.

This thesis advocates that architectural improvements such as sub-block allocation in the cache, non-temporal stores that replace instead of evict cached data, hardware cache probe instructions, and memory copy engines will improve the performance of kernel block memory operations in the future. However, these architectural improvements must be combined with the appropriate software routines in order to take advantage of any performance

benefits. Together, they are able to take into account the many considerations simultaneously affecting the performance of block memory operations and then make optimal decisions based on this information.

Bibliography

[1] David Bernstein, Doron Cohen, and Ari 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*, pages 19–26, 1995.

[2] Hrvoje Bilic, Igor Chirashnya, and Zorik Machulsky. Deferred segmentation for wire-speed transmission of large tcp frames over standard gbe networks. In *IEEE Symposium on High Performance Interconnects*, page 81, 2001.

[3] Matthias A. Blumrich, Cezary Dubnicki, Edward W. Felten, and Kai Li. Protected, user-level DMA for the SHRIMP network interface. In *HPCA*, pages 154–165, 1996.

[4] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. *SIGARCH Computer Architecture News*, 19(2):40–52, 1991.

[5] J. Bradley Chen and Brian 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*, pages 120–133. ACM Press, 1993.

[6] Tien-Fu Chen. An effective programmable prefetch engine for on-chip caches. 1995.

[7] H. K. Jerry Chu. Zero-copy TCP in solaris. In *USENIX Annual Technical Conference*, pages 253–264, 1996.

[8] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):733–746, 1995.

[9] Amer Diwan, David Tarditi, and Eliot 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*, pages 1–14, Portland, Oregon, 1994.

[10] Edward H. Gornish and Alex Veidenbaum. An integrated hardware/software data prefetching scheme for shared-memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, 1994.

[11] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. Informing loads: Enabling software to observe and react to memory behavior. Technical Report CSL-TR-95-673, 1995.

[12] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct cache access for high bandwidth network i/o. In *ISCA*, pages 50–59, 2005.

[13] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct cache access for high bandwidth network I/O. In *Proceedings of the International Symposium on Computer Architecture*, 2005.

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

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

[16] Doug Joseph and Dirk Grunwald. Prefetching using Markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, 1997.

[17] N. Jouppi. Cache write policies and performance. In *Proceedings of the 20th International Symposium on Computer Architecture*, 1993.

[18] Norman 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*, pages 364–373, May 1990.

[19] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2), 2004.

[20] Jeffrey Katcher. PostMark: a new file system benchmark. Technical Report TR3022, Network Appliance.

[21] Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed, and Pat Conway. The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, 23(2), 2003.

[22] Ando Ki and Alan E. Knowles. Adaptive data prefetching using cache information. In *Proceedings of the 11th International Conference on Supercomputing*, pages 204–212, 1997.

[23] Hyong-Youb Kim and Scott Rixner. Tcp offload through connection handoff. In *EuroSys 2006*, 2006.

[24] Alexander C. Klaiber and Henry M. Levy. An architecture for software-controlled data prefetching. *SIGARCH Computer Architecture News*, 19(3):43–53, 1991.

[25] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, and J. Hogberg. Simics: A full system simulation platform. *IEEE Computer*, 35(2), 2002.

[26] Srihari Makineni and Ravi Iyer. Architectural characterization of tcp/ip packet processing on the pentium m microprocessor. In *HPCA*, pages 152–163, 2004.

[27] Evangelos P. Markatos and Manolis G. H. Katevenis. User-level DMA without operating system kernel modification. In *HPCA*, 1997.

[28] Milo M. K. Martin, Mark D. Hill, and David A. Wood. Token coherence: A new framework for shared-memory multiprocessors. *IEEE Micro*, 23(6):108–115, 2003.

[29] Milo M. K. Martin, Mark D. Hill, and David A. Wood. Token coherence: Decoupling performance and correctness. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 182–193, 2003.

[30] Larry McVoy and Carl Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 USENIX Technical Conference*, pages 279–295, January 1996.

[31] Jeffery C. Mogul. Tcp offload is a dumb idea whose time has come. In *Workshop on Hot Topics in Operating Systems*, pages 227–291, 2003.

[32] Todd Mowry, Monica Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. pages 62–73, 1992.

[33] Chih-Jui Peng and Gurindar S. Sohi. Cache memory design considerations to support languages with dynamic heap allocation. Technical Report 860, Computer Sciences Department, University of Wisconsin-Madison, July 1989.

[34] Steven Przybylski. The performance impact of block sizes and fetch strategies. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 160–169, 1990.

[35] Scott Rixner. Memory controller optimizations for web servers. In *Proceedings of the International Symposium on Microarchitecture*, pages 355–366, Portland, OR, December 2004.

[36] Vatsa Santhanam, Edward H. Gornish, and Wei-Chung Hsu. Data prefetching on the HP PA-8000. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 264–273, 1997.

[37] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(2):7–21, 1978.

[38] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[39] Steven P. VanderWiel and David J. Lilja. A compiler-assisted data prefetch controller. In *Proceedings of the International Conference on Computer Design*, 1999.

[40] Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, and Charles C. Weems. Guided region prefetching: A cooperative software/hardware approach. In *ISCA*, pages 1–11, 2003.

[41] R. Paul Wilson, Michael S. Lam, and Thoma G. Moher. Caching considerations for generational garbage collection. In *Proc. 1992 ACM Conf. on Lisp and Functional Programming*, pages 32–42, San Francisco CA (USA), jun 1992.

[42] Yeh, Chao, Mannem, Gervais, and Booth. Introduction to tcp offload engine. Technical Report TR3022, 10 Gigabit Ethernet Alliance, 2002.

[43] Benjamin Zorn. The effect of garbage collection on cache performance. Technical Report CU–CS–528–91, University of Colorado, Department of Computer Science, 1991.