Hardware Support for Bulk Data Movement in Server Platforms

Li Zhao[†], Ravi Iyer[‡] Srihari Makineni[‡], Laxmi Bhuyan[†] and Don Newell[‡]

[†]Department of Computer Science and Engineering, University of California, Riverside, CA 92521
Email: {zhao, bhuyan}@cs.ucr.edu

[‡]Communications Technology Lab, Intel Corporation, Hillsboro, OR 97124
Email: {ravishankar.iyer, srihari.makineni, donald.newell}@intel.com

Abstract

Bulk data movement occurs commonly in server workloads and their performance is rather poor on today's microprocessors. We propose the use of small dedicated copy engines, and present a detailed analysis of a bulk data copy engine architecture. We describe the hardware support required to implement the copy engine and to tightly integrate it into server platforms. Our evaluation is based on an execution driven simulator that was extended with detailed models of bulk data movement engines. The simulation results show that dedicated engines are quite effective in eliminating the data movement overhead and are an attractive choice for handling bulk data in future high performance server platforms.

1 Introduction

Bulk data movement includes data initialization and data copies. Initialization of data is a very common operation that occurs frequently during memory management. Memory copy is another important overhead that affects I/O processing. Most networking stacks undergo at least one memory-to-memory copy when moving data between the application and kernel space, especially when receiving data. Recent TCP/IP performance results on today's microprocessors have shown that bulk data movement is a significant performance bottleneck [1]. Alleviating this memory copy overhead is important in order to allow 10Gbps processing and beyond required by many network-intensive server workloads [2].

Bulk data movement incurs a large overhead on general-purpose microprocessors because of the following reasons: (1) microprocessors move data at register (small) granularity, (2) several long-latency memory accesses are involved because source and/or destination are typically in memory (not in cache), (3) the memory accesses clog up all the processor resources (load/store queues, cache line fill buffers and re-order buffer entries), and end up stalling the CPU for a long time and (4) latency hiding techniques such as out-of-order execution [3], multi-threading techniques [4] [5], prefetching [6] [7] can tolerate a few simultaneous memory accesses, but are not very effective to address the bulk data movement overhead.

Our main contribution in this paper is the detailed exploration of design and implementation choices for bulk data copy engines. Focusing on a bus-based centralized-memory (UMA) system, we cover trade-offs and point out issues along the following dimensions: (1) proximity to memory, (2) access to cache, (3) interconnect design modifications, (4) coherence protocol changes and (5) adherence to consistency models. For the operation modes, we have considered synchronous and asynchronous execution of copies by the copy engine. We discuss instruction-based triggering of copy execution and evaluate whether the copy execution in the copy engine should be synchronous or asynchronous with respect to the CPU.

Based on our analysis, we focus on the implementation options and describe the changes required in the platform to integrate the copy engine solution. We model the implementation in an execution-driven simulator and evaluate the performance benefits. Based on a detailed case study of the TCP/IP processing, our evaluation shows that use of a copy engine can speed up TCP/IP processing by 15 to 50% depending on the packet sizes.

The rest of this paper is organized as follows. Section 2 describes motivation for a copy engine to solve the bulk data movement problem. In Section 3, we describe the architectural design and implementation for integrating copy engines in server platforms. Section 4 presents the evaluation methodology and analyzes the performance benefits of copy engines. Section 5 describes related work and Section 6 summarizes the paper and presents future research directions.

2 Motivation for Copy Engine for Bulk Data Movement in Servers

Figure 1 illustrates the basic characteristics of copy execution on CPUs versus copy engines (CEs). Performance improvement can be realized by employing copy engines due to the following benefits.

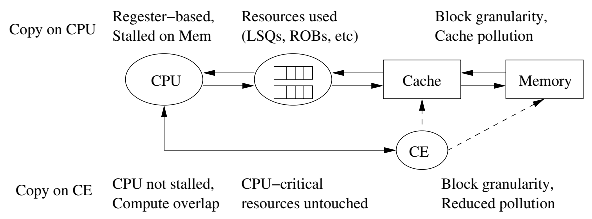

Copy on CPU
Regester–based, Stalled on Mem
Resources used (LSQs, ROBs, etc)
Block granularity, Cache pollution
CPU
Cache
Memory
CE
Copy on CE
CPU not stalled, Compute overlap
CPU–critical resources untouched
Block granularity, Reduced pollution

Figure 1. Copy execution on CPU vs. CEs

(1) Potential for faster copies and reducing CPU resource occupancy: The memory copy function is usually implemented as a series of load and store operations (memory to register and vice-versa). Even though the CPU reads data into cache at cache line granularity (64 bytes or higher in most modern processors) it performs copy by reading data into registers which are either 32 or 64 bit long. Copy engines can be used to speed up this copy operation since it can perform copies at higher (cache line) granularity. In addition, since the series of load/store instructions end up occupying load/store queues, re-order buffer and cache line fill-buffers, even if the CPU were able to look far ahead in the instruction window and execute other instructions, it would not be able to do so due to lack of resources. Copy engine can improve this by freeing up CPU resources so that other instructions can be executed.

(2) Copies can be done in parallel with CPU computation: Just like individual memory accesses are overlapped by computation, memory copies can be performed in parallel with CPU computation as well. If asynchronous memory-to-memory copy operations can be enabled using CEs, CPU is free to perform other operations. This is similar to a Direct Memory Access (DMA) operation where data is transferred between the memory and the device directly.

(3) Potential to avoid cache pollution and reduce interconnect traffic: Memory copy is a streaming workload from a caching point of view. Unless the source or the destination is needed by the application after the copy, allocating this data in the cache can result in pollution as it may kick out other valuable data from the cache. For many workloads like TCP/IP processing, the source of the memory copy is rarely touched by the workload after the copy, and the destination is touched by the application since it is the recipient of the incoming network data. However, most of the applications like a web server employ multiple processing threads, which may not touch the data immediately or even on the same processor. Thus, allocating the destination may also pollute the cache. Use of a copy engine allows for better control of this pollution, i.e. the copy engine can be designed to be configurable so as to allow for various options by the applications running on the server. Similarly, the copy engine can also reduce the interconnect traffic in the platform. For instance, in a system with centralized memory, embedding the copy engine next to the memory controller can potentially reduce the traffic that is placed on the interconnect (like a shared bus). This has the potential to reduce the queuing delays on the bus and thereby provide additional improvement to the application performance.

Researchers in the past have attempted to use DMA engines (exposed as peripheral devices) to accelerate bulk data movement. However, DMA engines have not entirely succeeded due to the following shortcomings:

(1) Descriptor setup entails setting up the [src, dest, length] parameters into shared memory descriptors and adding them to a list accessible to DMA engine. This requires at least one memory access which costs 300 to 500 clock cycles.

(2) Uncacheable triggers that trigger the DMA engine (also referred to as a doorbell) require the use of an uncacheable write to a DMA engine register. Such an uncacheable access typically is a very long latency operation (< 500 clock cycles).

(3) Notification of the copy completion is either through polling or through interrupts; both are expensive with interrupts being far worse.

(4) DMA engines operate in physical address space to prevent the use of the DMA engine by user-level stacks and applications. An alternative is to lock down pages (containing source and destination buffers) in memory, which is prohibitive specifically for use in application space.

Our goal in this study is to find a solution that avoids all of the above overheads and thereby achieve an efficient low-cost asynchronous copy. The copy engine in a server platform is designed to meet the following requirements: (1) low-overhead communication between the host processor and the engine, (2) hardware support for allowing the engines to operate asynchronously with respect to processors, (3) hardware support for sharing the virtual address space between the processor and the engine, and (4) low-overhead signaling of completion. The detailed design is presented in the next section.

3 Architecture and Design of Server Platforms with Integrated Copy Engines

In this section, we discuss the hardware support to integrate copy engines into a server platform. We describe the details of our design and implementation.

3.1 Server Architectures & CE Placement

We focus on a dominant architecture for server platforms: a bus-based centralized architecture with external memory controllers and uniform memory access (UMA/EMC [8]). Figure 2 illustrates basic components of a UMA architecture and also points out the potential choices of CE placement.

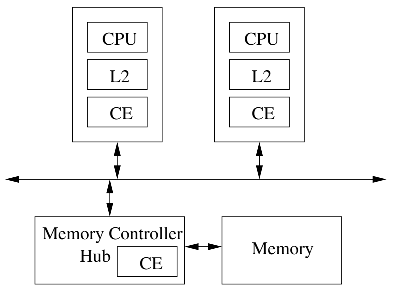


Figure 2. Copy engine placement in server platforms

Proximity to memory: Ideally, a CE should be integrated into the memory controller so that it can perform DRAM-aware sequences of reads and writes directly without occupying any other resources in the platform. In order to be close to memory controller, the CE must be integrated into the memory controller hub (MCH). However, note that as the number of cores increase on each processor socket (in CMP architectures), it may be also be desirable to provide replicated CEs on each socket.

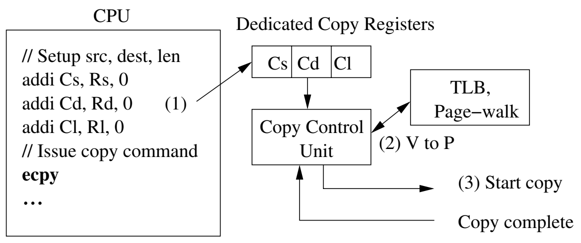
Proximity to cache: Since copy is initiated by CPU, it is possible that, in some cases, the source of the copy is already in cache. Similarly, it is also desirable in some cases that the destination should be written into the cache by the CE so that the application can avoid cache misses when touching the data subsequently. In order to support these caching benefits, CEs should have access to the cache. Since the last level of cache is typically on-die and as a result, access to cache largely means that the copy engine should be integrated on-die.

Based on the above analysis, we consider an off-die CE, which is placed in the MCH. However, this solution can be easily applied/extended for the other possibilities shown in Figure 2. We also believe these to be the most relevant solutions given how the server platform architectures are evolving.

3.2 Triggering Copy Execution on the CPU

In order to trigger copy execution, we now describe the ISA and micro-architectural support required in the CPU. There are three steps involved, as illustrated in Figure 3.

(1) Copy Initiation: A memory copy operation typically requires three operands: the source address, destination address and the length of the copy. For a CISC machine, one instruction may be enough to specify all three operands. However, for a RISC machine, more instructions may be required. We assume a RISC machine in order to describe the additional ISA support required. We propose the addition of three new registers (indicated with the *C* prefix to denote Copy). These three copy registers are first initialized with the source (in Cs), destination (in Cd) and length (in Cl) by using existing instructions (like *addi*). After all the copy parameters are available, a new instruction called *ecpy* is issued to start the process of communicating the copy parameters to the CE. At this point, the copy control unit (CCU) reads the three copy registers and buffers them.



```
// Setup src, dest, len
addi Cs, Rs, 0
addi Cd, Rd, 0
addi Cl, Rl, 0
// Issue copy command
ecpy
...
```

Figure 3. CPU support for initiating copy operations

(2) Address Translation: After receiving a copy command, the CCU translates the source and destination addresses from virtual to physical address by using the TLB (This may require a page walk if a TLB miss is detected). If the memory copy region crosses a page boundary, this copy must be split up into several operations, each of which has three operands with contiguous physical memory regions.

(3) Copy Communication: Once the translation(s) is complete, each resulting copy is individually communicated to the copy engine. Note that the communication of the three parameters to the CE should be atomic and ordered to avoid any interleaving of parameters between simultaneous memory copies issued by different processors in the platform. We discuss the interconnect support needed for this in the following section.

3.3 Communication between CPU and Copy Engine

The communication between CE and CPU needs to traverse the global interconnect (a shared bus). A typical pipelined bus [9] consists of address lines, command lines and data lines. A bus transaction goes through several phases – the phases of interest here include arbitration, request, snoop, response and data. After the arbitration phase, the CPU is allowed to place a transaction on the bus. The transaction typically consists of request type (read, write, invalidate, etc.), length of the transaction and physical memory address.

To send a copy transaction on the bus, we require two addresses (source, destination) and the length of the copy to be placed on the address/command bus. In order to do so, we encode a new request type called *copy*. By placing the copy request type on the bus during the request phase, we indicate that two different addresses will be transmitted to agents (CPU, MCH, etc) on the address lines in the subsequent clock cycles. During these clocks, the length of the request is asserted in the command lines. As a result, all nodes in the system are able to latch the request on the bus. The response and data phases are largely ignored for this transaction since it is similar to a memory write transaction where CPU does not expect any response or data. Once the MCH collects the copy addresses and length, it communicates the copy command to the CE. After some period of time, the CE will place completion status on the bus so that completion of the copy is visible to all the nodes in the system. The completion is indicated by placing the destination address on the address bus, the status of the copy on the data bus and the copy request as well as the ID of the requesting agent on the command bus.

3.4 Operation Modes for Copy Engines

We consider two execution modes: synchronous and asynchronous. Figure 4 illustrates CE's flow of execution starting after the copy command is communicated to the CE and ending with the CE notifying the CPU of copy completion.

Synchronous Copy Engine: The simplest mode is to execute the copy synchronously with respect to CPU. The CE notifies CPU only after the copy is completed. As a result, the *ecpy* instruction will not retire until the copy is completed. Since the CE is in the cache coherent domain so that it performs coherent reads and writes by sending out the necessary snoops to all the processors, and by performing speculative memory reads/writes. As shown in Figure 4(a), by doing these in parallel, the latencies are overlapped and the overall execution time can be reduced. Once the copy is completed, the CE sends a notification to the requesting processor, which then retires the *ecpy* instruction.

Asynchronous Copy Engines: Synchronous copy engines allow the CPU to overlap the execution of as many instructions as can be held in the re-order buffer. However, the re-order buffer is typically small (around 128 entries), and a copy operation can take a much longer period of time to complete. Since there is room for additional overlap, we consider asynchronous copy execution where CE notifies CPU of copy completion earlier than actual completion. To

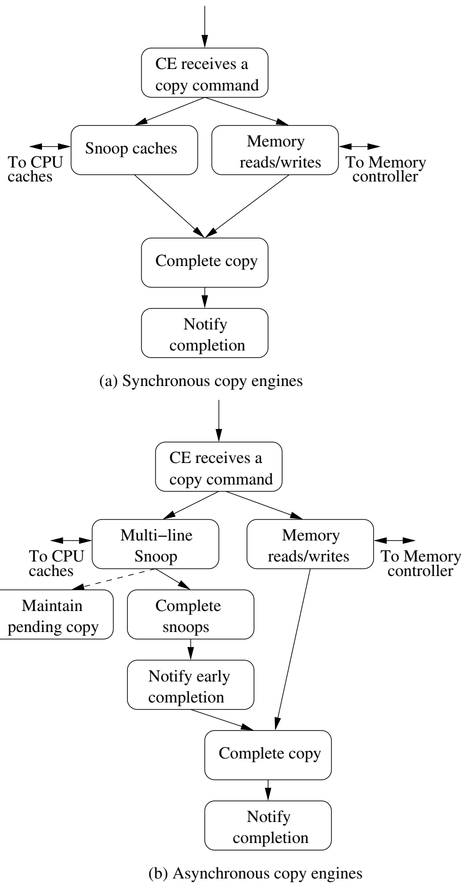


Figure 4. Copy execution flow

enable this, the CE needs to make the outstanding copy globally observable by informing all processors that it is using the memory regions. Typically, this is made possible by broadcasting individual snoop operations. However, as broadcasting one snoop per cache line and receiving individual responses for each may incur additional overhead, we propose to use a *multi-line snoop* operation, which sends base address and length of the buffer to processor caches. Upon receiving the multi-line snoop for source and destination of the copy, the CCU enters this information in a pending copy table to track the pending copies and sends back an acknowledgement to the CE. As shown in Figure 4(b), the CE then sends an early completion notification to the requesting processor so that it can retire the *ecpy* instruction. All subsequent loads and stores from any processor are locally compared against the entries in the pending copy table and are stalled and retried until the copy is actually completed. Once the CE completes copy, it broadcasts the completion notification to the processors in the platform so that the local CCU can delete the copy entry from the pending copy table. This releases any loads and stores that are pending in the processor.

3.5 Copy Engine Structure and Memory Transactions Scheduling

CE consists of a queue of control registers, which store the source, destination addresses and the length of outstanding copy commands. It also contains a stream buffer to store copied data. Since data transfer size requested from the memory is typically one cache line (e.g., 64 bytes), more than one data transfers are required if the copy length is larger than a cache line size. Assuming there is one memory queue *Mem_Q* in the memory controller, CE performs the following steps: (1) enqueues a sequence of read requests to *Mem_Q*, (2) reads data from the memory into a stream buffer, (3) enqueues a sequence of write requests to *Mem_Q*, (4) writes data from the stream buffer to the memory. By issuing a stream of reads and writes, we can exploit the row locality in the memory. It is known that modern DRAM chips latch an entire row (row buffer) on the first access to that row. Subsequent accesses to the same row have much lower latency because these accesses will not incur the precharge and row access latencies.

Since both CPU and CE issue memory requests to *Mem_Q*, we consider scheduling of these two types of requests. This depends on CE's operation mode. FIFO can be used in synchronous mode. However, CE may issue several read/write requests into *Mem_Q* so that CPU requests suffer a longer delays. This limits the extent to which CPU can overlap other computation with the copy operation. To let the CPU make maximum forward progress while CE is busy, we design another scheduling approach called *CPU request bypass*, where CPU requests are placed in front of CE generated requests in *Mem_Q*. To avoid CE starvation, we have implemented a fairness algorithm which guarantees a pre-determined amount of memory bandwidth for the CE. In the rest of the paper, we use SYNC_WLB to indicate CPU request bypass and SYNC_NLB to indicate no bypass during synchronous mode. ASYNC indicates asynchronous copy mode which assumes CPU request bypass.

3.6 Copy Retirement and Dependency Check

Copy retirement and dependency check are the last two steps to ensure that copies are completed appropriately and all dependency checks are maintained appropriately.

Copy retirement: As mentioned earlier, CCU receives copy notification from CE(s) in the platform. It has to deal with three types of copy notification: (1) an early copy completion notification for a copy that it generated, (2) a final copy completion notification for a copy that was either initiated by itself or by another unit, (3) a multi-line snoop notification for a copy that was initiated elsewhere. When receiving an early notification or a final notification, CCU ensures that *ecpy* instruction is retired if it initiated the copy. Upon receiving final notification, it also deletes the copy entry from the pending copy table. Upon receiving a multi-line snoop, CCU enters the addresses in the pending copy table so that it can enable dependency checks for asynchronous copy execution.

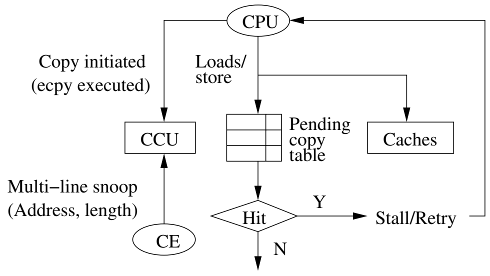

CPU
Copy initiated (ecpy executed)
Loads/ store
CCU
Pending copy table
Caches
Multi−line snoop (Address, length)
CE
Hit
Y
N
Stall/Retry

Figure 5. Dependency check for asynchronous copies

Dependency check: Another issue that needs to be addressed is dependency check. For instance, once the copy command is initiated, CPU continues to execute other instructions, which could be dependent on the copy source or destination data. In order to detect these data dependencies, CCU maintains the outstanding copy source and destination addresses in the pending copy table. Figure 5 shows how dependency is detected and handled. Just as the subsequent loads and stores are compared to pending memory transactions in load/store queues, they should also be compared against the copy commands in the pending copy table. This is especially critical when copies are executed asynchronously. When receiving a multi-line snoop, CCU enters the addresses and length into the pending copy table. This allows the CPU to perform dependency checks globally against copies initiated by all CPUs.

4 Performance Evaluation

4.1 Execution-Driven Simulation

Our copy engine simulation results are based on an execution-driven simulator SimpleScalar [10]. Since SimpleScalar had a simplistic memory subsystem model based on a fixed latency calculation, our first task is to modify the cache/memory subsystem to a detailed event-driven model, which implements (1) MSHRs to limit the number of outstanding memory transactions, (2) a pipelined bus model to accurately model latency and queuing effect, (3) a memory subsystem model that takes into account DRAM cycle times (row access, column access, precharge) based on the page conflicts, number of memory channels and DRAM technology. Our second major task is to add hardware support required to enable synchronous and asynchronous CE models.

This is achieved by adding instruction support to the simulator, (2) modeling of the communication between CPU and CE via the interconnect and (3) modeling of CE to generate coherent reads and writes to memory subsystem. The final task is to calibrate the model with appropriate parameter values so as to simulate the delays observed in a realistic server platform. While we do not have multi-processor support in SimpleScalar, the delays we model take into account the time taken for snooping all the processors in a 4-way platform.

Our base system configuration is a four-way fetch/issue/commit MIPS micro-processor. The detailed configuration parameters are shown in Table 1. Variations to the parameters are explained where needed. The memory latency (80ns) also includes the delays taken to propagate the request and response via a realistic chipset. In addition to latency, the simulation takes into account the stalls experienced in the cache hierarchy due to fully occupied MSHRs and queuing delays experienced due to bus and memory traffic generated by the application.

Table 1. Architectural parameters of simulation

CPU	
CPU Clock rate	3.0 GHz
Peak issue, retire rate	4 instructions/cycle
Instruction window size	128
Functional units	2 integer arithmetic, 1 floating point
Cache	
L1 inst/data cache	32 Kbytes, 4-way, 64-byte block
L1 cache hit time	2 cycles
L2 unified cache	1 Mbytes, 8-way, 64-byte block
L2 cache hit time	15 cycles
MSHR size	8
Bus	
system bus width	64 bits
system bus clock rate	800 MHz
DRAM	
Speed	400 MHz DDR 2
Average access time	80ns
Precharge time	15 ns
Row access time	15 ns
Column access time	15 ns

The TCP/IP processing workload that we use is derived from the FreeBSD stack [11] and performs receive-side processing. To simulate different types of network traffic, we drive the TCP/IP stack with three different packet traces. Each trace contains 100,000 packets with a fixed packet payload size as 512, 1024 and 1400 bytes respectively. These packet sizes were chosen to represent the typical receive traffic in a web server, email server and database server configurations.

4.2 Summary of Copy Engine Configurations

We present and analyze simulation results for three CE configurations: (1) SYNC_NLB: synchronous CEs with no load bypass, (2) SYNC_WLB: synchronous CEs with load bypass and (3) ASYNC: asynchronous CEs which assumes load bypass. As mentioned earlier, load bypass allows memory requests generated by CPU to be interleaved with CE requests and therefore does not have to stall until all outstanding copies are completed. We compare these three CE configurations with the base system configuration that has no CE support (BASE). All three traces are run through the TCP stack in these four configurations (BASE, SYNC_WLB, SYNC_NLB and ASYNC). All data is shown in the form of percentage of execution time reduction as compared to the base case.

4.3 Performance Benefits of Copy Engines

Figure 6 shows the execution time reduction with the CE as compared to a base system in in-order (IO) and out-of-order (OOO) execution mode. While our primary focus is the system with an OOO core, we compare against an IO core to show how poorly copy may perform if the core is unable to extract parallelism by looking ahead in the instruction window. As expected, the benefits of a CE are significantly greater in an IO system versus than an OOO system. For instance, when processing 512-byte packets in an IO system, use of a CE reduces the execution time by 43% to 58%. When processing the same-sized packet in an OOO system, use of a CE reduces the execution time by 15% to 48.5%. Based on these results, we also notice that the difference between synchronous engine and asynchronous engine performance is less significant in an IO system than in an OOO system. The reason is that the major benefit of a CE in an IO system is the reduction in copy latency itself, whereas a significant portion of the benefit in an OOO system is because of computation overlap.

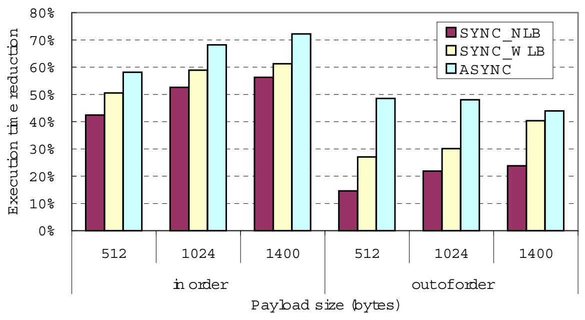


Figure 6. Performance benefits of CEs

To understand the underlying reasons for the benefits of CE, we break down the execution time reduction into two parts: (1) faster copy caused by the fact that CE moves data at a faster rate, and (2) computation overlap where some computation are executed in parallel with copy operation. As confirmed in Table 2, in an in-order execution system, most of the improvement is from faster copies. In addition, the amount of compute overlap is limited due to the in-order restriction placed on execution. Therefore even with ASYNC CEs, the number of the instructions that can be executed in parallel with copy is very limited.

On the other hand, for an out-of-order system, the three copy execution modes have different contributions. With SYNC_NLB, the improvement is mostly due to faster copies. Since this approach does not allow subsequent load instructions to bypass the copy command, only a limited number of instructions can be executed in parallel with copy operation. Those instructions that have data dependency on the load instructions have to wait until the copy completes. SYNC_WLB improves upon this by allowing CPU issued load instructions to bypass the copy command, so that more instructions can be executed in parallel with the copy. For 512-byte packets, the computation overlap is increased from 2.5% (out of 14.6%) for SYNC_NLB to 19.6% (out of the 27.1%) for SYNC_WLB. However, SYNC_WLB is still limited by the fact that the number of subsequent instructions that can be executed cannot exceed the size of the instruction window and the load/store queue (128 and 64 in our case).

Table 2. Factors affecting execution time reduction payload size

Payload Size (bytes)	CE operation modes	In order (%)			Out of order (%)		
		Overall	Faster copies	Overlap	Overall	Faster copies	Overlap
512	SYNC_NLB	42.5	40.7	1.8	14.6	12.1	2.5
	SYNC_WLB	50.5	37.4	13.1	27.1	7.5	19.6
	ASYNC	68.2	50.5	17.7	48.5	7.5	41.1
1024	SYNC_NLB	52.5	51.4	1.1	21.9	20.4	1.5
	SYNC_WLB	58.9	50.5	8.4	30.1	17.4	12.7
	ASYNC	68.2	50.5	17.7	48.0	17.4	30.6
1400	SYNC_NLB	56.3	55.4	0.9	23.8	22.7	1.1
	SYNC_WLB	61.2	54.5	6.7	40.3	32.0	8.3
	ASYNC	72.1	54.5	17.6	43.9	32.0	11.9

ASYNC alleviates this limitation by allowing *ecpy* instruction to retire sooner than its actual completion. This increases computation overlap from 19.6% (SYNC_WLB) to 41.1%. As we increase packet size from 512 to 1400 bytes, we can see that the improvement because of faster copy is increased whereas that because of computation overlap is reduced. The former observation is obvious since the percentage of the copy latency in total execution time is higher with larger payload sizes. Thus faster copies contribute more to the improvement. The latter observation is because of the same reason: the percentage of the computation is smaller with larger payload sizes. In summary, we believe an asynchronous execution model is critical to provide sufficient performance benefits even with larger payload sizes.

4.4 Impact of Memory Subsystem

We look at the impact of memory system on performance with the 1KB packet trace. Figure 7 shows that the execution time reduces significantly in all four modes as we increase the memory bandwidth from 3.2 GHz to 6.4 GHz. This is expected because even with CE, the bottleneck still remains in the memory system. Therefore faster memory gives better performance. Figure 8 illustrates performance as we decrease the memory access latency from 110 ns to 80 ns. We can see that memory access time has less impact on CE than the base case. For instance, the execution time of BASE is reduced by 10% and 18% when latency is reduced to 95 ns and 80 ns respectively. However with SYNC_NLB, the execution time reduction changes by hardly 2% (from 33.7% to 36.1%). The difference is even less for SYNC_WLB and ASYNC copy engines (1% and 0.2% respectively).

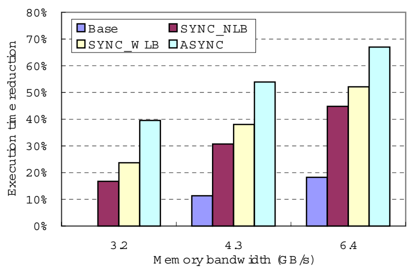


Figure 7. Impact of memory bandwidth

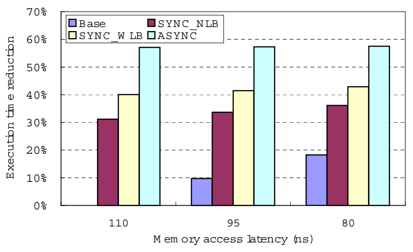


Figure 8. Impact of memory latency

5 Related Work

Researchers have attempted to address the bulk data movement solution in the past. From a networking standpoint, two major solution vectors are copy avoidance [12] [13] [14] [15] and copy acceleration [16]. However, most copy avoidance (zero-copy) techniques have not been adopted widely in commercial OS due to their limitations in scope and specific requirements. For instance, in the case of page remapping [14], when the network packet sizes are smaller than O/S page sizes, zero-copy is inefficient and requires pages to be pinned down in memory (which in turn requires TLBs to be flushed, etc). Other approaches like new APIs and kernel structures [12][13] require modifications to the application and hence have not been adopted yet. On the other hand, RDMA [15] achieves zero-copies using TCP Offload Engines (TOE), which we believe are not viable solutions from an economical as well as technology standpoint [1]. For copy acceleration, use of DMA engines for memory-to-memory copies is quite attractive. However, DMA engines are typically treated as peripheral devices which may impose a significant overhead in communication between CPU and DMA engine. Finally, since the DMA engine largely deals with physical addresses (as it has no translation support), user-level applications are not allowed to take advantage of it. Our goal in this paper is to investigate hardware support needed to enable copy engines with the requirement that they are tightly coupled into the platform and have low communication overhead. Other copy acceleration techniques include use of larger registers to move data at a large granularity [16]. While these techniques speed up the copy operation to some extent, they still stall the CPU for a long time.

6 Conclusions and Future Work

In this paper, we described the bulk data movement problem and pointed out the need for adding specialized engines in server platforms. We presented the architectural design and implementation for copy engines with synchronous as well as asynchronous execution modes. The hardware support including CPU support, interconnect support, copy engine design and coherence/synchronization requirements were also presented in detail. Finally, we modeled copy engine solutions by extending an execution-driven simulator and showed that the performance benefits of proposed copy engines are significant.

We believe that integration of copy engines in server platforms has significant potential. Similar design can be extended to other bulk data processing engines. In future work, we plan to evaluate the benefits of copy engines for wider range of applications. We also plan to extend our analysis to other bulk data processing like XML parsing and encryption. We believe that the basic framework presented in this paper can be easily extended to accommodate other frequently occurring operations.

References

[1] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, and et al., "TCP onloading for datacenter servers," *IEEE Computer Magazine*, November 2004.

[2] S. Makineni and R. Iyer, "Performance characterization of TCP/IP processing in commercial server workloads," in *6th IEEE Workshop on Workload Characterization (WWC-6)*, October 2003.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishing Company, 1999.

[4] D. Marr and et al., "Hyper-threading technology architecture and microarchitecture," *Intel Technology Journal*, February 2002.

[5] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proc. of the 22nd Annual International Symposium on Computer Architecture*, June 1995, pp. 392–403.

[6] D. Callahan and et al., "Software prefetching," in *4th Int'l conference on Architectural support for programming languages and operating systems*, April 1991.

[7] T. Chen, "An effective programmable prefetch engine for on-chip caches," in *Micro-28*, December 1995.

[8] Intel Corporation, "IA-32 intel architecture optimization reference manual." [Online]. Available: http://www.intel.com/design/Pentium4/documentation.htm

[9] ——, "Pentium(R) Pro family developer's manual, volume 1: Specifications." [Online]. Available: http:// developer.intel.com/design/ archives/processors/pro/docs/242690.htm

[10] D. Burger and T. Austin, "The SimpleScalar Tool Set, version 2.0," University of Wisconsin, Tech. Rep. CS-TR-97-1342, June 1997.

[11] M. K. McKusick and et al., *The Design and Implementation of the 4.4BSD Unix Operating System*. Addison-Wesley Publishing Company, 1996.

[12] J. Brustoloni, "Interoperation of copy avoidance in network and file I/O," in *IEEE INFOCOM*, March 1999.

[13] J. Brustoloni and P. Steenkiste, "Effects of buffering semantics on I/O performance," in *Proc. OSDI-II, USENIX*, October 1996, pp. 277–291.

[14] M. Thadani and Y. Khalidi, "An efficient zero-copy I/O framework for UNIX," Sun Microsystems Laboratories, Tech. Rep. SMLI TR-95-39, May 1995.

[15] RDMA Consortium, "Architectural specifications for RDMA over TCP/IP." [Online]. Available: http://www.rdmaconsortium.org

[16] Sun Microsystems Laboratories, "Accelerating core networking functions using the UltraSPARC VIS[tm] instruction set." [Online]. Available: http://www.sun.com