Analysis of a User-space Device-driver for the *memcpy* Hardware

Filipa Duarte and Stephan Wong
Computer Engineering
Delft University of Technology
F.Duarte@ce.et.tudelft.nl, J.S.S.M.Wong@ewi.tudelft.nl

Abstract— **In this paper, we analyze the utilization of a previously presented *memcpy* hardware when accessed through a user-space device-driver in a computer system. As the *memcpy* hardware only performs the actual data movement when it has less impact on the system, it is faster and does not incur on cache pollution. We also present the details of the chosen implementation and the results of executing two benchmarks suites, LMbench and STREAM. We demonstrate that the *memcpy* hardware can reach up to 121 times average execution time speedup for copies of 32kB when utilizing a 32kB cache. We analyze the impact of changing the processor frequency, the memory latency and the cache-line size and we concluded that our device is access bounded and not copy size bounded.**

I. Introduction

The oncoming deployment of gigabit Ethernet means another jump in performance requirement of network equipment. This will also put further strain on machines and processors in these machines that are running the TCP/IP stack in software that is usually part of the operating system (OS). In particular, the time arrival of packets is in the same order of magnitude as the time it takes to process a single packet. Therefore, the TCP/IP stack processing is becoming a bottleneck in gigabit networks.

Researchers analyzed the stack processing and have identified the main time-consuming parts of the TCP/IP code [2], [4], [12]: OS integration, checksums and memory copies. Focusing on resolving the memory copy bottleneck, the work presented in [16] introduced the concept of a dedicated *memcpy* hardware unit that works in conjunction with any existing cache to alleviate this bottleneck. Subsequent publications in [3] and [15] show implementations of the *memcpy* hardware prototyped in a Xilinx Virtex-4 Field Programmable Gate Array (FPGA) family. The limitations imposed by the chosen platform at that time meant that OS support was not possible and therefore only synthetic benchmarks were utilized to show the performance benefits of the *memcpy* hardware.

In this paper, the *memcpy* hardware was modelled on the Simics full-system simulator [6], with a standard Pentium 4 processor running the standard Linux 2.4 kernel. We compare different memory copy algorithms, throughput and execution time by running two benchmark suites: LMbench [8] and STREAM [7]. We also identify the influence of changing the memory latency, the cache-line size and the processor frequency.

We implemented the *memcpy* hardware accessed through a user-space device-driver, in a computer system. Although having a device accessed through a user-space device-driver implies performance degradation, this is the easier and more straightforward way of including a new hardware in a computer system running an OS. Accessing a device through a user-space device-driver [14] implies performance degradation due to the overhead introduced by system calls and additional memory copies to communicate with the kernel. In particular, acquiring and releasing the device, looking-up a particular address in /dev/mem file and read/write to it. However, we expect our hardware to still bring benefits compared with the software implementation of a memory copy. We also expect that the impact of changing some system parameters (memory latency, cache-line size processor frequency) may not be visible due to the performance degradation of accessing our hardware through a user-space device-driver.

The main contributions of this paper are:

- The investigation of accessing the *memcpy* hardware through a user-space device-driver in a computer system.
- The performance benefits of the *memcpy* hardware over the software implementation when accessed using a user-space device-driver.
- The study of the impact of changing the memory latency, the cache-line size and the processor frequency on the performance of the *memcpy* hardware.

This paper is organized as follows. In Section II, we analyze the related work and in Section III, we explain the concept behind the *memcpy* hardware. In Section

IV, we present the details of our implementation and in Section V, we describe the benchmarks used to experiment our *memcpy* hardware. In Section VI, we depict and analyze the results of executing the previously described benchmarks suits and the impact of changing some system parameters. Finally, in Section VII, we conclude our work and present some future directions.

II. Related Work

Several solutions have been proposed to overcome the bottleneck that memory copies present to today's processors. We present first the related work on software solutions and after on hardware solutions.

There are several software solutions based on the so called 'zero-copy' scheme. Examples of this scheme are published by [1] and [5]. These works present kernel buffer management systems and their integration with the OS for uniprocessor systems. In multiprocessing environment, the authors of [13] propose a 'zero-copy' message transfer with a pin-down cache technique, which avoid memory copies between the user specified memory area and a communication buffer area. Another 'zero-copy' technique for multiprocessor systems is presented by the same authors in [10]. In this paper, the authors design an implementation of the message passing interface (MPI) using a 'zero-copy' message transfer primitive supported by a lower communication layer to realize a high performance communication library. Software solutions for optimizing memory copies have also been presented in [11]. The authors designed and implemented new protocols of transmission targeted to parallel computing of the high speed Myrinet network. The authors of [9] introduced a new portable communication library, that provides one-sided communication capabilities for distributed array libraries, and supports remote memory copy, accumulate, and synchronization operations optimized for non-contiguous data transfers.

Only recently hardware solutions started to appear to solve the memory copies bottleneck, mainly the ones due to I/O adapters. The traditional DMA solution has been used intensively to transfer data between network cards and the memory and hence reduce the time spent on writing the received/transmited data from/to network cards to/from memory. Non-DMA based solution was presented in [17] where the authors present a hardware support for memory copies. This work presents a copy engine that is able to duplicate the data in the main memory by adding new features to the traditional memory controller. This provides reduction of cache pollution, however it will result in a unnecessary overhead if the data is touched by processor. Our solution is different from this one because we assume the copied data is touched by the processor and so we do not change the memory controller but the cache. Besides, we delay the actual data movement to when these movements have smaller impact to the system. In [15] we presented a hardware accelerator to perform *memcpy*'s in conjunction with the processor's cache. However the system used to prototype the accelerator cannot support OS. As such, in this paper we present the simulation of a complete system with the proposed *memcpy* hardware.

III. *memcpy* Hardware Concept

Our hardware solution stems from the simple observation that in many cases the data to be copied is already present inside the cache. Performing the *memcpy* operation in a traditional manner (utilizing loads and stores) would pollute the cache by either inserting data already present in the cache or overwriting data that may be needed later on. Our solution has the advantage of not performing the actual data movements at the moment of the copy (resulting in the mentioned disadvantages), but it performs a *memcpy* by filling an new indexing table inside the cache. Each valid entry of the indexing table represents a copy of one cache-line and it is a pointer to a valid cache entry. Figure 1 depicts the conceptual design of the *memcpy* hardware.

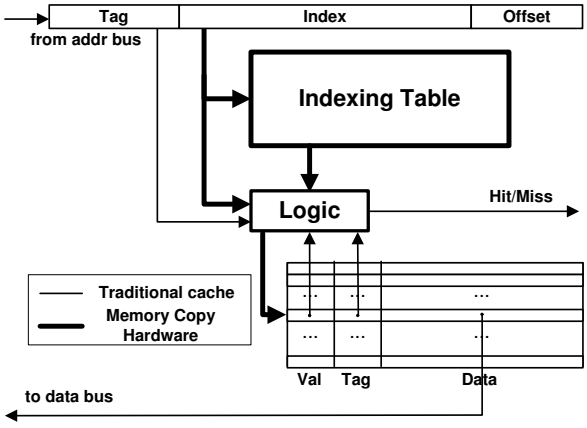

Fig. 1. The *memcpy* hardware solution

In order to maintain consistency in the memory, a write operation to any data locations (stored over multiple cache-lines) will result in the invalidation of

the corresponding cache-line and writing the cache-line back to the memory (this is when the actual data movement is performed).

IV. *memcpy* Hardware Implementation

The traditional *memcpy* operation performs a copy of size *size* from a source address *src* to a destination address *dst*. The indexing table is accessed by the index part of the *dst* address and contains the index parts of the *src* address, the tag part of the *dst* address and a bit stating that it is a valid entry. If there is a read hit in the indexing table (calculated based on the tag part of the *dst* address and the valid bit), the index part of the *src* address is provided to the cache (this is the pointer to the cache entry). On a write to the *dst* address, the corresponding entry on the indexing table is invalidated and the data pointed in the cache is written to the memory. After the copied data has been written, the new cache-line is fetched from the memory to the cache, where it is updated with the new data. On a write to a *src* address, the indexing table is looked up to find the corresponded entry. This entry is then invalidated and the the system repeats the same behavior as on the case of writing to the *dst* address. Figure 2 depicts the implementation of the *memcpy* hardware.

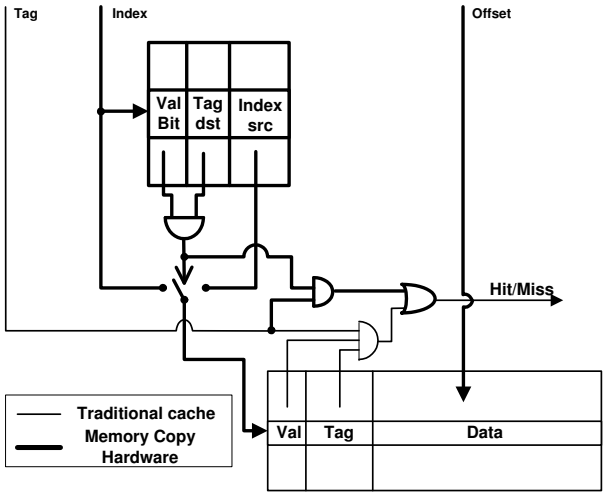

Fig. 2. The *memcpy* hardware implementation

The cache and cache-line sizes, the associativeness and the replacement policy of the cache have no influence in the design of our system. However, the write miss policy has to be write-allocate, in order to fetch from memory the correct cache-line when the copy is performed. The other limitation is that the maximum size of a copy is depended on the number of cache-lines that the cache has, as our *memcpy* hardware can only perform copies of entire cache-lines.

We implemented our *memcpy* hardware on the Simics full-system simulator [6], with a standard Pentium 4 processor at 3 GHz running the standard Linux 2.4 kernel. Our baseline cache design has a separate instruction and data cache. The instruction cache is the standard Intel Pentium 4 trace write-back write-allocate 4-way associative cache with 1536 cache-lines each with 64 bytes. The data cache is a write-back write-allocate 4-way associative cache, with 512 cache-lines each one with 64 bytes. The total size of the cache is then 32kB (corresponding to the maximum size of a copy). The cache hit penalty (both read and write for both instruction and data caches) is 2 clock cycles and the replacement policy is Least Recently Used (LRU).

The *memcpy* hardware is connected to the cache in the way depicted in Figure 2. The device is accessed through a user-space device-driver and the application writes to a specific address stored in a memory location in the /dev/mem file, where the device registers are. In order to find the correct location to write in the /dev/mem file, this file is lookup. The penalty of filling the indexing table is modelled to be 2 clock cycles, and the penalty of a write to either the *src* or *dst* addresses is also 2 clock cycles (according to the data gathered by [15]).

The memory is modelled with an average latency of 240 clock cycles which corresponds to an average access time of 80 ns of a DDR2 400MHz.

V. Benchmarks

We executed two benchmark suites, the LMbench [8] and STREAM [7]. For the LMbench we only used the application that measures the memory bandwidth (bw_mem), that provides the average throughput and execution time at which a processor can move data. The comparison is done between the *glibc bcopy* and our *memcpy* hardware and we used this benchmark to measure the impact of changing the memory latency, the cache-line size and the processor frequency on the performance of the *memcpy* hardware and on the *glibc bcopy*. The STREAM benchmark is used to compare the benefits of our *memcpy* hardware with several copy kernels: *copy_8*, *copy_32*, *copy_64*, *copy_32x2*, *copy_32x4*, *memcopy* and *glibc memcpy*. The first three kernels implement the copy using a loop and the block size of the copy doubles for each kernel. The following two kernels parallelize the copy in two or four blocks and the *memcopy* kernel parallelize in

eight blocks. Finally, the *glibc memcpy* uses the standard *glibc* algorithm implemented in the Linux kernel. The STREAM benchmark also provide the average throughput and execution time. Both benchmarks do not re-use data, which means the execution time and throughput measured includes the penalty of fetching the necessary data from memory into the cache and filling the indexing table each time a copy is executed.

The execution time of each benchmark is measured in a different way. For the STREAM benchmark, the default number of executions is 10 000. For copies of 32kB this would imply several days of simulation. We evaluated the impact of reducing this number on the accuracy of the results and noted that reducing the number of executions by 10 would still return the accurate numbers. As such, we used 1000 executions of the algorithm and then averaged them. This process is repeated 10 times and the best time is displayed.

The LMbench, on the other hand, estimates the necessary number of iterations that provide an accuracy of 95% of the execution time [8]. Looking at the number of iterations, the number is higher on LMbench than on STREAM, which leads to the conclusion that LMbench is more accurate then STREAM. This is the reason why we choose STREAM to compare our *memcpy* hardware to other copy algorithms and the LMbench to further study the impact of changing the memory latency, the cache-line size, and the processor frequency at the end of next section.

VI. Results

In this section we present and discuss the simulation results of both benchmarks. All results are depicted in graphs with a logarithmic scale in order to improve readability, except when stated otherwise. As presented early our cache design has 64 bytes cache-lines (which correspond to the minimum copy size) and has 512 cache-lines (which implies a maximum copy size of 32kB).

The first analysis (see Figure 3) presents a comparison of the average throughput of both benchmarks for our *memcpy* hardware for different copy sizes. The explained earlier, LMbench is more accurate than STREAM thus it is not surprising the average throughput to be slightly higher than the one presented by STREAM.

The subsequent analysis is comparing our *memcpy* hardware with other copy algorithms. We used the STREAM kernels and included our hardware. Figure 4 depicts the average execution time and Figure 5 depicts the average throughput comparisons. It is

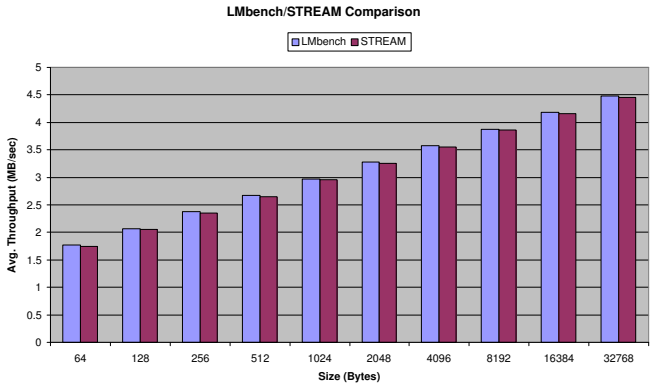

Fig. 3. Comparison of the average throughput of LMbench and STREAM benchmarks (log scale)

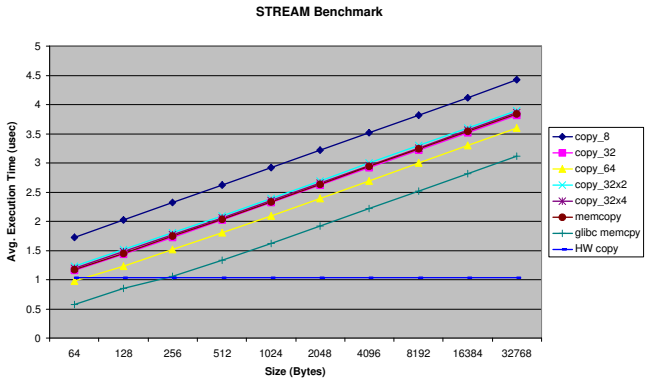

Fig. 4. Comparison of the average execution time for STREAM benchmark (log scale)

clear from the figures that for copies smaller than 512 bytes (4 cache-lines) our *memcpy* hardware presents a penalty compared with the *glibc memcpy* algorithm. However, as soon as the sizes of the copies increase the benefit of using our hardware becomes evident, and can reach, for copies of 32kB, an average execution time speedup of approximately 121. In [16], we presented 82% reduction of the execution time (comparing the *memcpy* hardware and the *glibc memcpy*) for copies of 256 cache-lines, which corresponded, on that system, to a copy of 8kB. For the same size of a copy, we can reach now a reduction of the execution time of 94.5%. The reason for this increase is due to the size of a cache-line. In [15] a cache-line was 32 bytes, while in this paper a cache-line is 64 bytes.

In order to understand the impact of changing some parameters of the system we used the LMbench kernel and copy sizes of 32kB. Table I presents the average throughput and execution time of the baseline

TABLE I

AVERAGE THROUGHPUT AND EXECUTION TIME OF THE BASELINE SCENARIO COMPARED WITH CHANGING THE PROCESSOR FREQUENCY, THE MEMORY LATENCY AND THE CACHE-LINE SIZE

	Baseline Scenario		Freq = 6 GHz		Lat = 180 clk		Cache-line = 128 B	
	glibc bcopy	*memcpy* Hardware	*glibc bcopy*	*memcpy* Hardware	*glibc bcopy*	*memcpy* Hardware	*glibc bcopy*	*memcpy* Hardware
Throughput (MB/sec)	24.89	30286.2	24.87	30231.6	33.3	40362.7	24.88	30286.2
Execution Time (usec)	1316.4	10.8	1317.2	10.8	983.8	8.1	1316.4	10.8

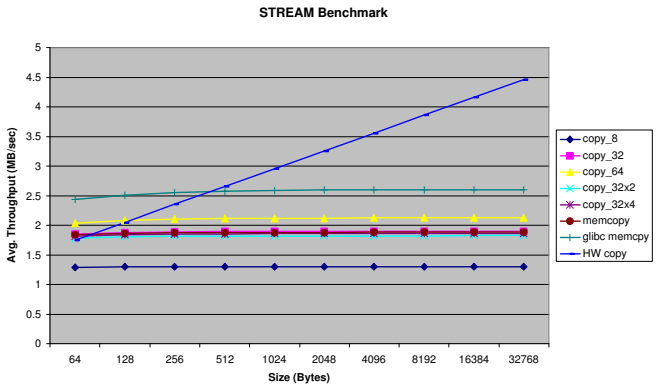

Fig. 5. Comparison of the average throughput for STREAM benchmark (log scale)

scenario, which corresponds to a memory latency of 240 clock cycles, a cache-line size of 64 bytes and a processor frequency of 3 GHz.

We increased the processor frequency to 6GHz, an increase of 100% to simulate the impact of future processors (with higher frequencies) on the copies algorithms. In order to have correct simulations, we also need to increase by the same amount the modelled latencies of the *memcpy* hardware, the cache and the memory, because the model is based on the number of cycles of the processor. As such, the penalty of filling the indexing table of the *memcpy* hardware is now of 4 clock cycles instead of the previously modelled 2 clock cycles, and the penalty of a write to either the *src* or *dst* addresses is also 4 clock cycles. For the cache a hit penalty (both read and write for both instruction and data caches) is now 4 clock cycles and the memory latency is now 480 clock cycles. Table I presents the correspondent results. The increase in frequency does not have an impact for either the *glibc bcopy* or our *memcpy* hardware, because the copy algorithms are not computing-intensive but memory-intensive.

In order to model the impact of future generations of memories, we simulated the impact of decreasing the memory latency to 180 clock cycles (60 ns), a decrease of 25%. Table I presents the correspondent results. As expected there is an average reduction of 25% on both the average throughput and execution time.

And finally, we increase the cache-line size to 128 bytes, an increase of 100%. Table I presents the correspondent results. We would expect to have a increase in the *memcpy* hardware average throughput and execution time, because we are actually using a bigger block to perform the copy. However, because the penalty of fetching data from memory is dominant, the benefit of increasing the cache-line size is not visible.

It is important to notice the impact of accessing the device has on these results. When the memory latency is decreased there is a decrease on *memcpy* hardware average throughput and execution time. However, when the cache-line size is increased, there is no change on the average throughput or execution time, when it should. The reason for this behavior is due to the way the *memcpy* hardware is accessed. As explained before the device is accessed through a user-space device-driver, so in order to access it there is the penalty of performing system calls and memory copies to communicate with the kernel. As such, reducing the memory latency will reduce the time to perform the necessary memory copies and consequently reduce the access time of the device (as depicted in Table I). However, increasing the cache-line size has no visible increase on the average throughput or execution time. The conclusion we can reach is that the device is bounded by access time and not by the copy size. This justifies the future work to implement a more

efficient access to the *memcpy* hardware.

VII. Conclusions

We presented in this paper the *memcpy* hardware integrated in a complete computer system. As the *memcpy* hardware does not perform the actual data movement at the moment of the copy, it is faster and does not incur on cache pollution. We also present the details of the chosen implementation and the results of executing two benchmarks suites, LMbench and STREAM. We demonstrated that the *memcpy* hardware can reach up to 121 times average execution time speedup for copies of 32kB. We analyzed the impact of changing the processor frequency, the memory latency and the cache-line size and we concluded that our device is access bounded and not copy bounded. As future work we will implement a more efficient access to the *memcpy* hardware.

References

[1] M.M. Buddhikot, X.J. Chen, W. Dakang, and G.M. Parulkar. Enhancements to 4.4 BSD UNIX for efficient networked multimedia in project MARS. In *Proc. of IEEE International Conference on Multimedia Computing and Systems*, 1998.

[2] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, pages 23–29, June 1989.

[3] F. Duarte and S. Wong. A *memcpy* Hardware Accelerator Solution for Non Cache-line Aligned Copies. In *Proc. of IEEE 18th International Conference on Application-specific Systems, Architectures and Processors*, 2007.

[4] J. Kay and J. Pasquale. Profiling and Reducing Processing Overheads in TCP/IP. *IEEE/ACM Transactions on Networking*, pages 817–828, December 1996.

[5] Yousef A. Khalidi and Moti N. Thadani. An Efficient Zero-Copy I/O Framework for UNIX. Technical report tr-95-39, Sun Microsystems, Inc., Mountain View, CA, USA, 1995.

[6] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hllberg, J. Hgberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, pages 50–58, February 2002.

[7] J. D. McCalpin. A Survey of Memory Bandwidth and Machine Balance in Current High Performance Computers. In *Newsletter of the IEEE Technical Committee on Computer Architecture*, dec 1995.

[8] L. McVoy and C. Staelin. LMbench: Portable Tools for Performance Analysis. In *Proc. of the Annual Technical Conference on USENIX 1996 Annual Technical Conference*, 1996.

[9] J. Nieplocha and B. Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems. *Lecture Notes in Computer Science*, pages 533–546, April 1999.

[10] F. O'Carroll, H. Tezuka, A. Hori, and Y. Ishikawa. The design and implementation of zero copy MPI using commodity hardware with a high performance network. In *Proc. ACM 12th International Conference on Supercomputing*, pages 243–250, 1998.

[11] L. Prylli and B. Tourancheau. BIP: A New Protocol Designed for High Performance Networking on Myrinet. In *Proc. International Parallel Processing Symposium Workshop on Personal Computer Based Networks of Workstations*, 1998.

[12] G. Reignier, S. Makineni, R. Illikkal, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP Onloading for Data Center Servers. *IEEE Computer*, pages 46–56, November 2004.

[13] H. Tezuka, F.O'Carroll, A. Hori, and Y. Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *Proc.IEEE 12th International Parallel Processing Symposium*, pages 308–315, 1998.

[14] Factorization of Device Driver Code between Kernel and User Spaces. `http://pages.cs.wisc.edu/~arinib/report.pdf`.

[15] S. Vassiliadis, F. Duarte, and S. Wong. A Load/Store Unit for a *memcpy* Hardware Accelerator. In *Proc. of IEEE 17th International Conference on Field Programmable Logic and Applications*, 2007.

[16] S. Wong, F. Duarte, and S. Vassiliadis. A Hardware Cache-Line *memcpy* Accelarator. In *Proc. of IEEE International Conference on Field-Programmable Technology*, 2006.

[17] L. Zhao, L. Bhuyan, R. Iyer, S. Makineni, and D. Newell. Hardware Support for Accelerating Data Movement in Server Platform. *IEEE Transactions on Computers*, pages 740–753, June 2007.