A Hardware Cache *memcpy* Accelerator

Stephan Wong, Filipa Duarte, and Stamatis Vassiliadis

Computer Engineering, Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
{J.S.S.M.Wong, F.Duarte, S.Vassiliadis}@ewi.tudelft.nl

***Abstract*— In this paper, we present a hardware solution to perform the commonly used *memcpy* operation with the goal to reduce the time to perform the actual memory copies. This is accomplished by taking advantage of the presence of a cache that is found next to many current-day (embedded) processors. Additionally, the currently presented solution assumes that the to be copied data is already in the cache and is aligned by the cache-line size. We present the concept and implementation details of the proposed hardware module and the system used to experiment both our hardware and an optimized software implementation of the *memcpy* function. Experimental results show that the proposed hardware solution is at least 79% faster than an optimized hand-coded software solution.**

I. Introduction

Currently, main memory accesses remain a performance bottleneck in any computing system. Many solutions to overcome this bottleneck have been proposed in literature by simply reducing the amount of main memory accesses (by introducing small and fast caches, changing data structures, modifying algorithms, etc.). Such solutions are the result of many profiling investigations of the targeted applications in question that determine the number and type of main memory accesses. Profiling information has been gathered by authors providing valuable insight into how a program behaves, e.g., with regard to the main memory, by identifying the most compute-intensive or data-intensive parts of the program.

In [1], the authors presented benchmarking and profiling results of the Linux kernel for 32-bit and 64-bit PowerPCs (PPCs). The authors conclude that optimizations are necessary to decrease the overhead of operations related to (1) maintenance of cache consistency with the main memory, (2) main memory copying, and (3) page table entries. In [2], the authors present profiling results of the Bluetooth protocol working under the Linux operating system. In this work, it is concluded that the main memory copies are the most time-consuming operations. Both [1] and [2] provide profiling information identifying an operating system function *memcpy* as having a major impact on the performance of the system.

The *memcpy* function is responsible for copying data of size *size* from memory address *src* to memory address *dst*. The C code is presented below:

```
/**
 * Copy one area of memory to another
 * @dst: Where to copy to (copy)
 * @src: Where to copy from (original data)
 * @size: Size of the area to copy
**/
void *memcpy(void *dest,void *src,
             size_t size)
{
  char *tmp=(char *)dest, *s=(char *)src;

  while (size--)
     *tmp++ = *s++;

  return dest;
}
```

It is worth noting that there has been extensive research on how to optimize this function in software. The most utilized solution is to hand-write this function in assembly and link it to the operating system instead of compiling the C code. This will result in more efficient code, however the optimizations are only valid for the selected system.

In this paper, we present a hardware unit that performs the *memcpy* operation using an additional indexing table in an existing cache organization. Our proposed solution has the following advantages:

- it performs a *memcpy* of one cache-line in 30 clock cycles and performs *memcpy*'s of cache-lines sizes varying from 32 bytes to 256 bytes (which are common values for cache-lines), 79% to 93% faster than an optimized software implementation, respectively.
- it avoids duplicating data in caches, because the copy (of the original data) is simply represented by inserting an additional pointer to the original data that is already present in the cache. This pointer allows the 'copied' data to be accessed from the cache.
- it offloads the processor as it is no longer required to perform the copies word by word (or the largest data unit the utilized architecture supports).

The proposed hardware solution is different from a Direct Memory Access (DMA) controller that simply offloads the processor from performing the necessary memory operations. I.e., the memory operations are still performed, but just not by the processor that can now do other useful work.

The paper is organized as follows. In Section II, we present related work. In Section III, we introduce the cache organization that is used for our evaluation and in Section IV we present the concept of the proposed hardware solution. In Section V, we discuss the platform used for the implementation of the memory copy hardware. The details of both the cache, the *memcpy* hardware and software are also presented. In Section VI, we present the experimental results for both the software and hardware implementations of the *memcpy* utilizing the same processor and compare the results. Finally, in Section VII, we draw some conclusions.

II. Related Work

In many network related applications, *memcpy* (among others) is considered to be an time-consuming operation. Several solutions (both software and hardware) have been proposed to reduce the impact of *memcpy*. In this section, we describe the related work presenting, first, profiling information of some of those network related applications and, second, we present some of the solutions proposed to reduce the impact of the *memcpy*. We also highlight the difference between our approach and existing solutions.

In [3] and [4], the authors present profiling results of the TCP/IP and UDP/IP protocols. They conclude that the main performance bottlenecks in network-related functionalities are the checksum calculations and data movements. In particular, in [4] the authors present that approximately 70% of the all processing time of the TCP/IP or UDP/IP protocol is due to data movement operations. In this paper, the authors also present network traffic traces. They show that 86% of the UDP traffic and 99% of the TCP traffic in a LAN (Local Area Network) is less than 200 bytes and that 99.7% of the traffic in a WAN (Wide Area Network) is less than 500 bytes.

In [5], the authors present a survey of the solutions proposed in literature to alleviate the network subsystem bottlenecks. Those solutions include (1) checksum optimizations (merge copying and checksum, checksum offloading and checksum elimination), (2) using a DMA instead of Programmable Input Output (PIO), and (3) avoiding cross domain data transfers through 'zero-copying' schemes.

Hardware optimizations, besides DMA support, include the use of vector processors. Specifically for the PPC, the Velocity Engine (also known as AltiVec [6]) expands the current PPC architecture through addition of a 128-bit vector execution unit. This unit operates concurrently with existing integer and floating-point units. This approach expands the processors capabilities to concurrently address high-bandwidth data processing (such as streaming video) and the algorithmic intensive computations.

Several software solutions have been proposed, basically variations on the 'zero-coping' scheme. In [7], the authors 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 is presented by the same authors in [8]. 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 [9]. The authors designed and implemented new protocols of transmission targeted to parallel computing that squeeze the most out of the high speed Myrinet network, without wasting time in system calls or memory copies, giving all the speed to the applications. The authors of [10] 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.

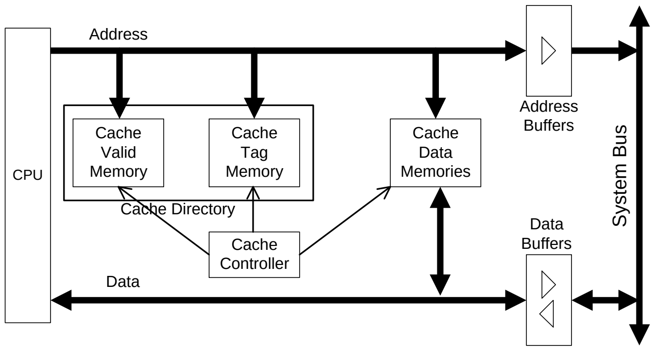

Fig. 1. Typical cache organization.

A solution not considered until now (to our best knowledge) is to develop a dedicated hardware unit to handle data movement by exploiting the presence of a cache that is closely attached to many current-day processors. Our solution does not require the usage of a DMA controller nor a vector unit, so it is more flexible than the solutions proposed by [5] and [6], and it can be applied to a majority of the computing systems. Assumptions made in the currently proposed solution are guided by the the profiling information provided in [4]. In order to implement such a hardware unit, we chose the Virtex II Pro FPGA (Field Programmable Gate Array) as it allows for fast prototyping. The platform Xilinx University Program (XUP) [11] design provides the necessary functionality to prove the concept.

III. Cache Organization

A cache is divided in two main parts: a cache directory and cache data-memories. The cache directory can be seen as a list of the main memory addresses of the data stored in the corresponding location of the cache data-memory (which is the one that contains the data). In our implementation, the cache directory is constituted by two different memories: a cache tag-memory and a cache valid-memory. Figure 1 depicts the referred cache organization. The address provided by the processor is divided into 3 parts: the index, the tag, and the offset. The *index* is used to access the cache directory and the cache data-memories. The *tag* is written to the cache tag-memory (on a write) or is used to compare with a tag already in the cache tag-memory (on a read). If the tag supplied by the cache tag-memory is the same as the tag of the address provided by the processor and the valid bit supplied by the cache valid-memory is set, a cache read hit is registered. On a cache read hit, the data supplied by the cache data-memories (the cache-line) is accessed and, based on the *offset*, the correct word is accessed. On a write hit, besides the tag, the index and the offset, also a byte write signal is used. This signal identifies which byte, within the selected word, is to be written.

In the remainder of this section, we present some implementation details of the cache organization pertaining the chosen experimentation platform (discussed in Section V-A).

On a read request, if the data is in cache (read hit), the processor will have the data available on the bus on the next clock cycle. If the data is not in cache (read miss) the processor has to stall until data is provided by the main memory (the time to have the data provided by the main memory is depended on the memory technology and implementation).

On a write request, both the cache and the main memory will be updated, if the data is in the cache (write hit); or only update the main memory, if the data is not in the cache (write miss). In our design, this will not trigger a load of a cache-line. The cache-line load will only be triggered when a read miss happens. This implementation is a standard implementation of a write-through cache.

If a read request is issued after a write and the requested data is not in the cache, the processor stalls until the write instruction is finished and the read data is available to be provided to the cache and to the processor.

IV. *memcpy* Hardware Organization

Our hardware solution of the *memcpy* operation stems from the simple observation that in many cases the data to be copied (of size *size*) from a source address (*src*) to a destination address (*dst*) 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. The proposed solution has the advantage of not performing the actual data movements (resulting in the mentioned disadvantages) and of being independent of the cache organization. Instead, the proposed solution performs a *memcpy* utilizing an additional indexing table inside the cache. The table is accessed by the index part of the *dst* address and contains the tag and the index parts of the *src* address, the tag part of the *dst* address and a bit stating that it is a valid entry. Each indexing table entry is a pointer to an entire cache-line. Summarizing, the *memcpy* operation can now be simply replaced by introducing a new index table to the cache data-memories by assuming that the data to be copied is already present in the cache and that the data is already aligned to a cache-line size. Finally, in order to maintain consistency in the main memory, any write operation to the data at either the source or destination locations (stored over multiple cache-lines) will result in the invalidation of the corresponding cache-line and writing the cache-line back to the main memory. Of course, the cache-line is updated with the new data of the write operation before writing it back to the main memory.

For the moment, our solution assumes copies of only entire cache-lines and data is resident in the cache. However, according to [2], the size of a *memcpy* operation in the Bluetooth standard under Linux is typically 339 bytes and according to [1], 98% of the *memcpy* calls are less than 128 bytes. As

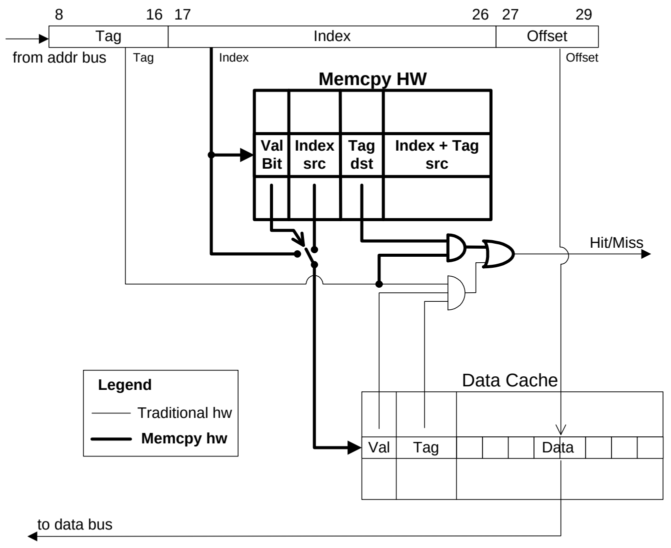

Fig. 2. Read request to both the indexing table and cache of the *memcpy* hardware unit

memcpy is extensively used on network processing and 86% to 99% of the packets are less than 200 bytes [4], we can safely assume, from [1], [2], and [4], that a typical value of a *memcpy* is between 128 to 200 bytes. This implies 4 to 6 cache-lines when assuming an cache-line size of 32 bytes.

If the addresses to perform a *memcpy* on are not cache-line aligned, the alignment can be done in software and the aligned addresses provided to the hardware. The software also has to perform the copy of the remaining words and bytes. This software part will have the same performance as the software part that performs bytes and words copy of the software implementation of *memcpy* (we will address the the software implementation in Section V-B).

Figure 2 depicts the read process, for both the copy and the original data, highlighting the indexing table and the necessary control logic. On a read request, the index part of the address provided by the processor will be used to access the indexing table:

- If the valid bit in the corresponded entry of the table is set (the requested data is a copy) and:
  - If the tag part of the address provided by the processor and the tag stored on the table are the same (a read hit on the indexing table), the output will give the index part of the corresponding *src* address, which is used to access the cache.
  - If the tag part of the address provided by the processor and the tag stored on the table are not the same (a read miss on the indexing table) then the index part of the address provided by the processor is used to access the cache (and it follows the cache accessing procedure described in Section III).
- If the valid bit in the corresponded entry of the index table is not set (the requested data is not a copy), then the index part of the address provided by the processor is used

TABLE I
HIT/MISS COMBINATION IN THE CACHE AND INDEXING TABLE

Indexing Table	Cache	Comments
hit	hit	Original data in cache and copy in the indexing table
hit	miss	Cannot occur
miss	hit	Original data in cache and no copy in the indexing table
miss	miss	No original data in cache

to access the cache (and it follows the cache accessing procedure described in Section III).

Table I summarizes the different cases of hits/misses on the indexing table and on the cache. It is worth mentioning that the case hit on the indexing table and miss on the cache cannot occur because this would imply that the requested data is not in the cache. For this to happen, data had to be changed, due to a write hit or a load overwriting the current content of the requested cache line. On such cases, the correspondent line on the indexing table is invalidated (as explained below). As the copy may be considered as a pointer to the original data, if the original data is changed (written to) the correspondent copy has to be written to memory and the table entry invalidated. The same situation happens when the copy itself is changed. For a write request, the indexing table is accessed using the index part of the address provided by the processor.

- If the valid bit is set (to write to a copy) and:
  - If the tag part of the address provided by the processor is the same as the tag stored on the entry of the indexing table (a write hit):
    1) Use the index given by the indexing table to access the cache;
    2) Write the data given by the cache to address provided by the processor in the main memory;
    3) Invalidate the indexing table entry, setting the valid bit to zero;
  - If the tag part of the address provided by the processor is not the same as the tag stored on the entry of the indexing table (a write miss):
    1) Use the address provided by the processor to access the cache and the main memory;
    2) Write data to the cache and the main memory;
    3) Search the indexing table to find if the address provided by the processor points to an original data; if it is, set the valid bit to zero;
- If the valid bit is not set, then the address provided by the processor is not on the indexing table (not a write to a copy):
  1) Use the address provided by the processor to access the cache and the main memory;
  2) Write data to the cache and the main memory;
  3) Search the indexing table to find if the address provided by the processor points to an original data; if it is, set the valid bit to zero;

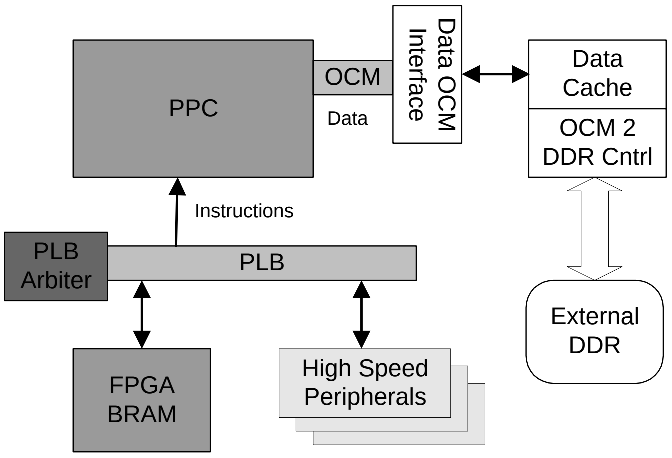

Fig. 3. System used to experiment the *memcpy* hardware

A typical problem on the software implementation of a *memcpy* function is to ensure that the *src* and the *dst* addresses do not overlap. However, this problem does not exist in our solution. If the addresses overlap, they are both present in the indexing table and in the cache, which means they are both pointing to the copy and the original data. On a write to the overlapped address, the cache will be updated (because there was a write hit) and the indexing table entry will not change (because it will still point to the new data).

V. IMPLEMENTATION ENVIRONMENT

We implemented the proposed hardware solution on an XUP platform containing a Virtex II Pro FPGA with two PowerPC (PPC) 405 cores, although only one was used. The PPC running at 100 MHz has an internal cache (a 16 Kbytes, two-way associative). As the proposed solution operate together with a cache, we implemented our own cache in order to have control/access over it, and so we disabled the PPC internal cache. Further details on the FPGA chip is given below.

Two main buses are used to connect the PPC to peripherals: the Peripheral Local Bus (PLB) bus and the On-Chip Memory (OCM) bus. The PLB is used to connect the FPGA user logic and high speed peripherals to the PPC. It has a handshaking protocol that, obviously becomes less efficient when the number of peripherals increase. The main memory is normally connected on this bus. The OCM bus is normally used to connect the FPGA Block RAMs to the PPC and it is constituted by a data side and an instruction side.

The data side of the OCM bus has an access time equivalent to a cache access [12], so we migrate the memory hierarchy to this bus. This implies disabling the internal cache of the PPC and implement a cache and a main memory controller on the data side of the OCM bus. Figure 3 depicts the described system. The interface that the data side OCM bus provides [13] has 22 bits address bus (bits 0 to 7 and 30 to 31 are reserved by the processor), 32 bits of separate read and write data bus and 4 bits of byte write bus. The byte write bus is

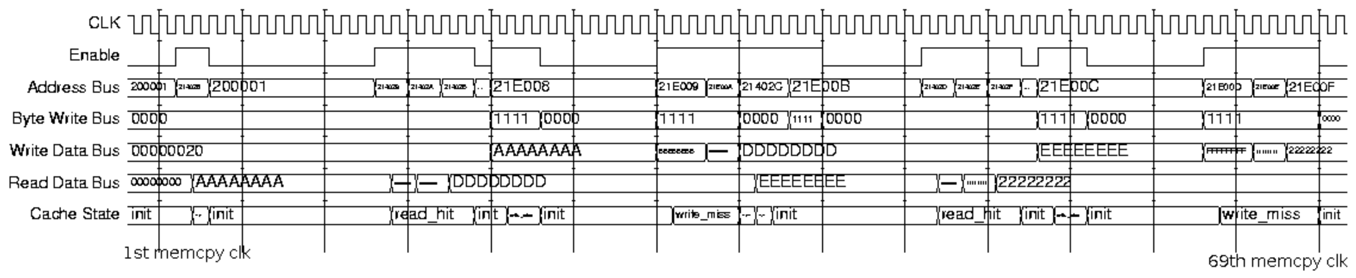

Fig. 4. Waveform of the software *memcpy* for one copy of a cache-line

TABLE II
ESTIMATION OF THE RESOURCES USED TO IMPLEMENT THE CACHE AND *memcpy* HARDWARE

	Total Available	Used in Cache	Used in *memcpy* Hardware
Slices	13696	658 (4%)	4507 (32%)
Flip-Flops	27392	352 (1%)	1157 (4%)
LUTs	27392	1162 (4%)	7126 (26%)
IOBs	556	164 (29%)	108 (19%)
BRAMs	136	34 (25%)	99 (72%)
Gclk	16	2 (12%)	2 (12%)

used to identify whether it is a read or a write, and, if it is a write, of how many bytes.

A. Cache and memcpy *implementation*

We implemented a 32 Kbytes direct-mapped write-through cache with 32 bytes cache-line. This implies that, from the 22 bits of the address, 9 bits are for the tag part of the address, 10 bits for the index and 3 bits for the offset. The cache and the *memcpy* hardware used 37 RAM (Random Addressable Memory) memory arrays [14] and 1 CAM (Content Addressable Memory) memory array [15], both developed by LogiCORE. Table II presents the estimation of the percentage of FPGA resources needed to implement the cache and the *memcpy* hardware. The RAM memory arrays read or write data in one clock cycle, while the CAM memory array provides data in one clock cycle for a read and it takes two clock cycles for a write. As the CAM is on the critical path of the *memcpy* hardware, the performance of a copy of a cache-line is bounded to the write time of this memory. Thus, our solution can perform a copy of one cache-line in two clock cycles.

The PPC is expecting to have the requested data available on the read data bus on the second clock cycle. When the PPC reads the data that was previously *memcpy*, it needs to wait one clock cycle for the indexing table and the normal one clock cycle for the cache to provide the data. However, PPC only expects data on the OCM bus after two clock cycles, so even if the data is available before, it is not used. The latency that a indexing table for the *memcpy* takes is, then, hidden on the latency of the PPC itself.

In order to perform a *memcpy* in hardware, the *memcpy* function in the program has to substituted by:

```
src = (int *)0xa1400004;
*src = // tag and index part of the src address

dst = (int *)0xa1400008;
*dst = // tag and index part of the dst address

size = (int *)0xa140000c;
*size = // number of cache-lines to copy

start = (int *)0xa1400000;
*start = 0x1; // start the hardware memcpy
```

B. memcpy *in software*

The XUP platform provides a PPC compiler, however the *memcpy* function implemented by this compiler does not provide any optimization for a cache-line copy. Comparing a hardware implementation of *memcpy* with such a software implementation is unfair.

One optimized implementation of this function is found in the Linux kernel for PPC. This optimized *memcpy* function is hand-written, in assembly, and it includes cache management instructions. As we implemented our own cache, it was necessary to change this optimized implementation in order to suite our system. All the optimizations for word or cache-line copies were kept.

The optimized software implementation of the *memcpy* is 32% faster than the default implementation. The software implementation of *memcpy* referred to in this paper is the modified Linux implementation.

VI. RESULTS AND COMPARISON

We implemented the hardware *memcpy* indexing table in VHDL. We used the ModelSim XE-III [16], a HDL simulation environment, that enables to verify the HDL source code and functional and timing models of the designs. Both the software implementation of *memcpy* and the hardware unit are analyzed using this tool.

For the software implementation of *memcpy*, there is a period to calculate if the addresses overlap and if there are bytes or words to perform a *memcpy* on. This time is 74 clock cycles and the total time to perform a *memcpy* of one cache-line in software is 143 clock cycles (Figure 4 depicts the waveform of the software version of *memcpy* for a copy of one cache-line). For a *memcpy* of five cache-lines (typical sizes for *memcpy*, according to [1] and [4], are between 128 and 200 bytes, or 4 to 6 cache-lines, in our case) the software implementation takes 419 clock cycles. For the hardware

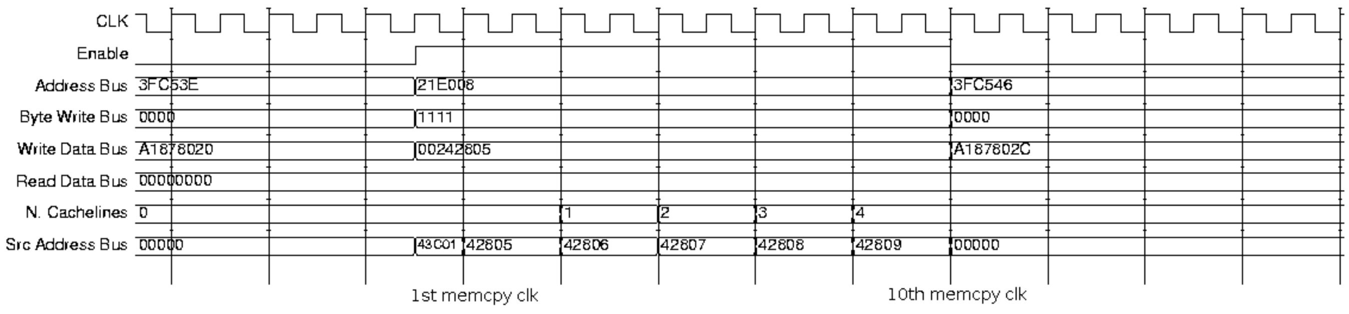

Fig. 5. Waveform of the hardware *memcpy* for a copy of five cache-lines

TABLE III
PERFORMANCE OF THE HARDWARE AND SOFTWARE *memcpy*

	1 Cache-line Copy	**5 Cache-lines Copies**	**1024 Cache-lines Copies**
SW *memcpy*	143 clk	419 clk	70730 clk
HW *memcpy*	30 clk (79%)	38 clk (91%)	2076 clk (97%)

implementation, there is also a setup period of transferring the *src* and *dst* addresses and the *size* to the hardware unit of 28 clock cycles. On a copy of one cache-line the unit takes 2 clock cycles and it takes 10 clock cycles to perform the copy of the five cache-lines (Figure 5 depicts the waveform of the hardware unit performing a *memcpy* of five cache-lines). Consequently, on a copy of one cache-line, our solution performs 79% better than the software implementation. On the copy of five cache-lines the benefit is of 91%. Table III presents the number of clock cycles and percentage that a copy of 1, 5 and 1024 cache-lines take in software and in hardware. Generalizing, a *memcpy* performed in hardware takes 28 clock cycles of setup time plus 2 clock cycles per cache-line.

On a read of both the copy or the original data, the PPC will have the data available in 2 clock cycles. On a write, the data will be updated in the cache (for a write to the original data in one clock cycle and, for a write to a copy, it will take one clock cycle to invalidate the line, plus the time needed to copy the data to the main memory). Figure 6 depicts the performance of the software and hardware implementation of *memcpy* for different number of cache-lines, represented in bytes instead of cache-lines. As expected, the benefit of our solution increase with the size of the copy. We also compared our solution with AltiVec [6] solution of *memcpy*. In [6], the authors present values for a copy of 160 bytes around 586 MB/sec (the picture presented does not allow to determine exact values, and no values are referred in the text). As the PPC runs at 750 MHz in their implementation, this corresponds to around 210 clock cycles. Our PPC is running at 100 MHz and we can perform a *memcpy* in 38 clock cycles, for the same 160 bytes (5 cache-lines). We can perform approximately 80% better than [6], in terms of clock cycles, although the throughput we can achieve is only 421 MB/sec (due to the difference in the PPC clock).

The current benefits and limitations of our solution to

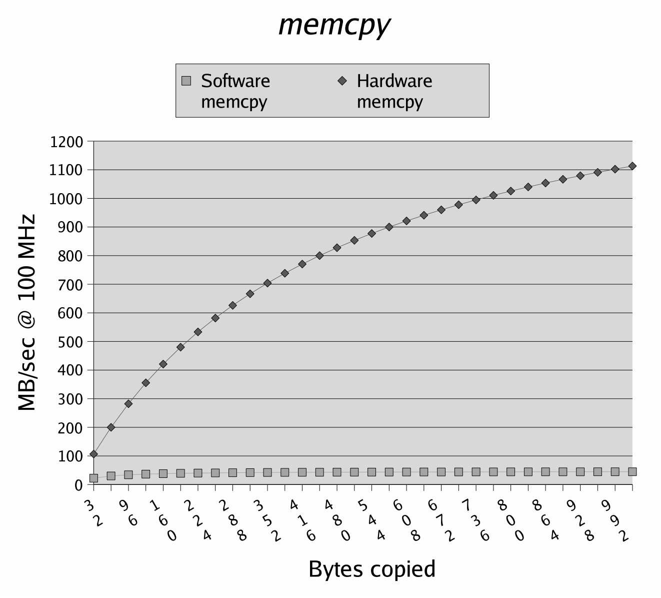

Fig. 6. *memcpy* throughput

perform memory copies are:

- Benefits:
  - No duplication of data in the cache;
  - Offloading the processor;
  - Each cache-line copy takes 2 clock cycles (excluding a setup time of 28 clock cycles for the first copy);
  - Performing copies with overlapped addresses does not hamper our solution.
- Current limitations:
  - Only support copies of entire cache-lines: the word and byte copies have to be performed in software;
  - Data has to be in cache;
  - Maximum size of a *memcpy* is 1024 bytes.

VII. CONCLUSIONS

In this paper, we presented a hardware unit that performs the *memcpy* operation using an additional indexing table in an existing cache organization.

Our proposed solution performs a *memcpy* of one cache-line in 30 clock cycles and performs *memcpy*'s of cache-lines sizes varying from 32 bytes to 256 bytes (which are common values for cache-lines), 79% to 93% faster than an optimized software implementation, respectively.

The indexing table to the cache also avoids duplicating data in caches, because the copy (of the original data) is simply represented by inserting an additional pointer to the original data that is already present in the cache. This pointer allows the 'copied' data to be accessed from the cache. Our solution also offloads the processor as it is no longer required to perform the copies word by word (or the largest data unit the utilized architecture supports).

References

[1] P. Mackerras, "Low-Level Optimizations in tehe PowerPC Linux Kernels," in *Proc. of the Linux Symposium*, 2003, pp. 321–331.

[2] F. Duarte and S. Wong, "Profiling Bluetooth and Linux on the Xilinx Virtex-II Pro," in *Proc. IEEE 9th Euromicro Conference on Digital System Design*, 2006.

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

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

[5] P. Wang and Z. Liu, "Operating System Support for High-Performance Networking, A Survey," *Journal of China Universities of Posts and Telecommunications*, pp. 32–42, Sept. 2004.

[6] "Enhanced TCP/IP Performance with AltiVec," http://www.freescale.com/AltiVec.

[7] 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*, 1998, pp. 308–315.

[8] 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*, 1998, pp. 243–250.

[9] 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.

[10] 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*, pp. 533–546, Apr. 1999.

[11] "Xilinx University Program," http://www.xilinx.com/univ/.

[12] "PLB vs. OCM Comparison Using the Packet Processor Software," http://www.xilinx.com/bvdocs/appnotes/xapp644.pdf.

[13] "Data Side OCM Bus v1.0 (v2.00a)," http://www.xilinx.com/bvdocs/ipcenter/data_sheet/ dsocm.pdf.

[14] "Single-Port Block Memory Core v6.2," http://www.xilinx.com/ipcenter/catalog/logicore/docs/ sp_block_mem.pdf.

[15] "Content Addressable Memory v5.1," http://www.xilinx.com/ipcenter/catalog/logicore/docs/ cam.pdf.

[16] "ModelSim^{TM}Xilinx Edition III," http://www.xilinx.com/ise/verification/ mxe_details.html.