A *memcpy* Hardware Accelerator Solution for Non Cache-line Aligned Copies

Filipa Duarte and Stephan Wong
Computer Engineering Laboratory
Delft University of Technology

Abstract

In this paper, we present a hardware solution to perform non cache-line aligned memory copies allowing the commonly used memcpy *function to cope with word copies. The main purpose is to reduce the latency in executing memory copies aligned on word boundaries. The proposed solution exploits the presence of a cache and assumes that the to-be-copied words are already in the cache. We extend an earlier proposed solution that exploited the cache-line alignment of* memcpy *function when 'moving' large amounts of data. 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 66% faster than an optimized hand-coded software solution.*

1 Introduction and Motivation

Nowadays, main memory accesses remain a performance bottleneck in many 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 [4], 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 copying data from or to the user process, copying pages of memory, and copying other data structures within the kernel. Separate procedures are used to perform these three operations: *__copy_tofrom_user*, *copy_page* and *memcpy*, respectively. 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 with the *memcpy* function topping the list. This function is mainly used within the Bluetooth protocol to re-assemble the received frames.

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 *dst,void *src,
             size_t size)
{
  char *tmp=(char *)dst, *s=(char *)src;

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

  return dst;
}
```

The work presented in [11] proposed a hardware solution to perform copies of cache-lines stemming from the observation that memory copies handle large amounts of data (often spanning several cache-lines) leaving non cache-line aligned data (at the beginning/end) to be handled by software. In this paper, we present a hardware unit that performs the *memcpy* operation, using an additional indexing table in an existing cache organization, that is able to support non cache-line aligned copies. The proposed solution extends the existing hardware support of the *memcpy* operation by further reducing the latencies of copying data in the main memory. However, we still assume that the to-be-copied data is already present in the cache. Our proposed solution has the following advantages:

- it performs a *memcpy* of one word 66% faster than an optimized software implementation.
- 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 actual copies.

The paper is organized as follows. In Section 2, we present related work. In Section 3, we introduce the cache organization that is used for our evaluation and in Section 4 we present the concept of the proposed hardware solution. In Section 5, we discuss the platform used for the implementation of the memory copy hardware. The details of the cache, the *memcpy* hardware and software are also presented. In Section 6, we present the experimental results for both the software and hardware implementations of the *memcpy* utilizing the same processor and compare the results. Finally, we draw our conclusions in Section 7.

2 Related Work

In many network related applications, *memcpy* (among others) is considered to be a 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 by presenting, first, profiling information of some of those network related applications and, second, some of the solutions proposed to reduce the impact of the *memcpy*. Additionally, we highlight the difference between our approach and existing solutions.

In [1] and [3], 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 [3] 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 [9], 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.

Several software solutions have been proposed, basically variations on the 'zero-copying' scheme. In [8], 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 [6]. In [6], 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 [7]. 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 [5] 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.

Hardware optimizations, besides DMA support, include the use of vector processors. Specifically for the PPC, the Velocity Engine (also known as AltiVec [10]) 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. The work proposed by [11] presents a dedicated hardware unit to handle data movement by exploiting the presence of a cache that is closely attached to many current-day processors. This solution does not require the utilization of a DMA controller nor a vector unit, since it is assumed that the data is already present in the cache, and it can be applied to a majority of the computing systems. However, a minor drawback to this solution is that non cache-line alignment data movements need to be handled by slower 'software'.

In this paper, we present a hardware unit that performs the *memcpy* operation, using an additional indexing table in an existing cache organization, that is able to support non cache-line aligned copies and word copies and that performs 66% faster than the hand-coded software *memcpy*.

3 Cache Organization

A cache can be logically divided into two 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 comprises two different memories: a cache tag-memory and a cache valid-memory.

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.

Next to this traditional cache organization, the authors in [11] proposed an indexing table. The subsequent section explains the basic concepts of this indexing table and the novelty to the work presented in this paper.

4 *memcpy* Hardware Organization

In [11], the authors proposed a hardware solution to perform *memcpy* operations of entire cache-lines. Given large size of memory copies, it is likely that they cover many cache-lines. The solution assumed that the data to be copied (of size *size*) from a source address (*src*) to a destination address (*dst*) is already present inside the cache. The proposed solution performed a *memcpy* utilizing an additional indexing table inside the cache to simply index the *dst* address and then 'point to' the *src* data within the regular cache. More specifically, 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.

The main problem with supporting non cache-line aligned data movements is twofold: the pointed data word can be located anywhere on a cache-line and the pointer (extracted from the *dst* address) does not have to be cache-line aligned either. Consequently, our solution performs non cache-line aligned copies by additionally including, in the indexing table, the difference between the offsets of the *dst* and *src* addresses and using the index and offset part of the address to access the indexing table. Each indexing table entry is, then, a pointer to a word (defined by the difference between the offsets of the *dst* and *src* addresses). The *memcpy* operation (independent of the alignment of the *src* and *dst*) can now be simply replaced by introducing a new indexing table to the cache data-memories. Figure 1 depicts the read process, for both the copy and the original data, highlighting the indexing table and the necessary control logic (for details on the indexing table, please refer to [11]).

On a read of a copied value, the indexing table provides to the cache, a new address which points to the requested data. This new address is based on the values stored on the indexing table (the tag and index parts of the *src* address, the tag part of the *dst* address and a valid bit). In the previous solution, to perform *memcpy* operations of entire cache-lines, each cache-line copied had only one corresponding address in the indexing table. However, for the solution presented here, the index part of the *src* address stored in the in-

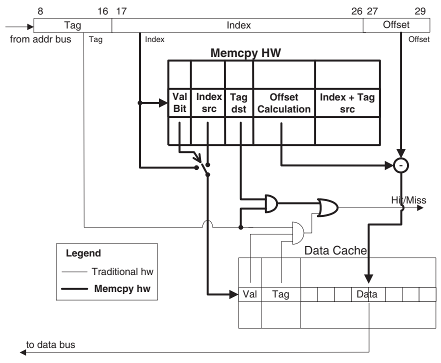

Figure 1. Read request to both the indexing table and cache of the *memcpy* hardware unit

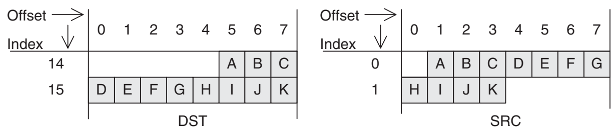

Figure 2. Example 1 of an offset calculation

dexing table can, in reality, refer to a previous or subsequent cache-line depending on the word requested. The new field of the indexing table (referred to as *offset_calculation*) is the difference between the offsets of the *dst* and *src* addresses and allows to calculate the correct address to be determined to access the cache. The processor requests a *req_index* and *req_offset* which, if it is a copied value, corresponds to an *index_src* and *offset_calculation* from the indexing table. The algorithm to provide the correct address to the cache, *index_cache* and *offset_cache*, is the following:

```
cal = req_offset - offset_calculation;
if cal > 7 then
    index_cache = index_src + 1;
    offset_cache = cal - 8;
elsif cal < 0 then
    index_cache = index_src - 1;
    offset_cache = cal + 8;
else
    index_cache = index_src;
    offset_cache = cal;
end if;
```

Lets consider two examples to further explain the algorithm. The first example has a positive *offset_calculation*.
Example 1: In this case, the *offset_calculation* is 4 (offset *dst* - offset *src* = 5 - 1). The processor requests *req_index* 15, *req_offset* 2 (this means the copy of the word *F*), the indexing table will return *index_src* 1 and *offset_calculation* 4.

```
cal = 2 - 4 = -2 < 0 =>
index_cache = 1 - 1 = 0; offset_cache = -2 + 8 = 6
```

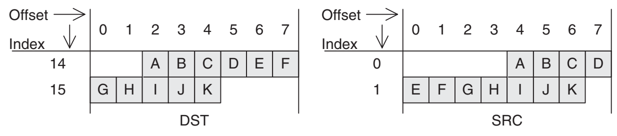

Figure 3. Example 2 of an offset calculation

This means the address provided to the cache is: *index_cache* = 0; *offset_cache* = 6, corresponding to word *F*, as expected.

Example 2: The second example has a negative *offset_calculation*. In this case, the *offset_calculation* is -2 (offset *dst* - offset *src* = 2 - 4). The processor requests *req_index* 14, *req_offset* 7 (this means the copy of the word *F*), the indexing table will return *index_src* 0, and *offset_calculation* -2.

```
cal = 7 - (-2) = 9 > 7 =>
index_cache = 0 + 1 = 1; offset_cache = 9 - 8 = 1
```

This means the address provided to the cache is: *index_cache* = 1; *offset_cache* = 1, corresponding to word *F*, as expected.

Since the copy is considered to be a pointer to the original data, special care must be taken when the original data is changed (written to) or if the cache is evicted. More specifically, the corresponding copy has to be written to memory and the table entry must be invalidated. The same situation occurs when the copy itself is changed. For a write request, the indexing table is accessed using the index and offset 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):
    - Access the cache based on the previously presented algorithm;
    - Write the data given by the cache to address provided by the processor in the main memory;
    - 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):
    - Use the address provided by the processor to access the cache and the main memory;
    - Write data to the cache and the main memory;
    - 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):
  - Use the address provided by the processor to access the cache and the main memory;
  - Write data to the cache and the main memory;
  - 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;

A typical problem in the software implementation of a *memcpy* function is to ensure that the memory regions pointed by the *src* and the *dst* addresses do not overlap. If they do, the software uses another function, the *memmove* function to perform this kind of copies. However, this problem does not exist in our solution. If the memory regions 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 memory region, 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). This implies our solution can perform also a *memmove* function.

If the addresses to perform a *memcpy* on are not word aligned, the alignment can be performed in software and the aligned addresses provided to the hardware. Consequently, the software has to perform the copy of the remaining bytes. This software part will have the same performance as the software part that performs bytes copy of the software implementation of *memcpy*.

5 Implementation Environment

We implemented the proposed hardware solution on an Xilinx University Program board. The XUP board contains a Virtex II Pro FPGA with two PowerPC (PPC) 405 cores with data and instruction caches. Two main buses are used to connect the PPCs to peripherals: the Peripheral Local Bus (PLB) and the On-Chip Memory (OCM). The PLB is used to connect the FPGA user logic and high speed peripherals to the PPC. The OCM is normally used to connect the FPGA Block RAMs to the PPC and it is constituted by a data side and an instruction side. In our implementation, we only used one PPC running at 100 MHz. As the proposed solution operates in conjunction with a cache, we disabled the PPC internal caches. The data side of the OCM bus has an access time equivalent to a cache access, therefore, we designed a cache attached to this bus that is physically addressed. As we only utilize the data side OCM bus for the

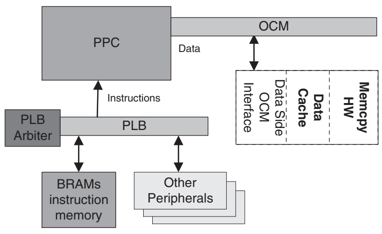

Figure 4. System used to experiment the *memcpy* **hardware**

	Total Available	Used in Cache	Used in *memcpy* Hardware
Slices	13696	658 (4%)	4533 (33%)
Flip-Flops	27392	352 (1%)	1157 (4%)
LUTs	27392	1162 (4%)	7181 (26%)
IOBs	556	164 (29%)	114 (20%)
BRAMs	136	34 (25%)	100 (73%)
Gclk	16	2 (12%)	3 (18%)

Table 1. Estimation of the resources used to implement the cache and *memcpy* **hardware**

cache and *memcpy* hardware, we utilized the BRAMs on the PLB to store the C program. A linker script was used to assign the data to the OCM and the instructions to the PLB. Figure 4 depicts the described system.

The cache implementation is a 32 KBytes direct-mapped write-through cache with 32 bytes cache-lines. This implies that, from the 22 bits of the address (the interface of the data side OCM has 22 bits : bits 0 to 7 and 30 to 31 are reserved by the PPC), 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 use 38 RAM (Random Addressable Memory) arrays and 1 CAM (Content Addressable Memory) array, both developed by LogiCORE. Table 1 presents the estimation of the percentage of FPGA resources needed to implement the cache and the *memcpy* hardware. The RAM arrays read or write data in one clock cycle, while the CAM 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 is bounded to the write time of this memory. Therefore, our solution can perform a copy of a word in two clock cycles.

On a read of a copied value, the PPC waits one clock cycle for the indexing table and the normal one clock cycle for the cache to provide the data. However, on a read of a normal (non-*memcpy*) value, the PPC has the data on the OCM one clock cycle after it was requested. Therefore, there is no effect on any other load/store of the program, except if it is a load/store to a copied value, then, there is a penalty of one clock cycle on a read.

In order to perform a *memcpy* in hardware, the *memcpy* function needs to pass the parameters *src*, *dst* and *size*. For that, the *memcpy* call in C the program has to be 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 words to copy
start = (int *)0xa1400000;
*start = 0x1; // start the hardware memcpy
```

The XUP platform provides a PPC compiler, however, the *memcpy* function implemented by this compiler does not provide any optimization for a word copy. Comparing a hardware implementation of *memcpy* with such a software implementation is unfair. An 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 suit our system. All the optimizations for word copies were kept. We used the system presented in Figure 4 to test the default implementation of *memcpy* and we compared it with the optimized version. 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.

6 Results and Comparison

We implemented the hardware *memcpy* indexing table in VHDL. We used the ModelSim XE-III, 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 to perform a *memcpy* on. This time is 74 clock cycles and the total time to perform a *memcpy* of 1 word in software is 89 clock cycles. For a *memcpy* of 8 words, the software implementation takes 143 clock cycles.

For the hardware implementation, there is 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 1 word the unit takes 2 clock cycles and it takes 16 clock cycles to perform the copy of the 8 words. Consequently, on a copy of 1 word, our solution performs 66% better than the software implementation. On the copy of 8 words the benefit is of 69%. Table 2 presents the number of clock cycles and

	1 Word	8 Words	40 Words	8192 Words
SW *memcpy*	89 clk	143 clk	419 clk	70730 clk
HW *memcpy*	30 clk (66%)	44 clk (69%)	108 clk (74%)	16412 clk (77%)

Table 2. Performance of the hardware and software *memcpy*

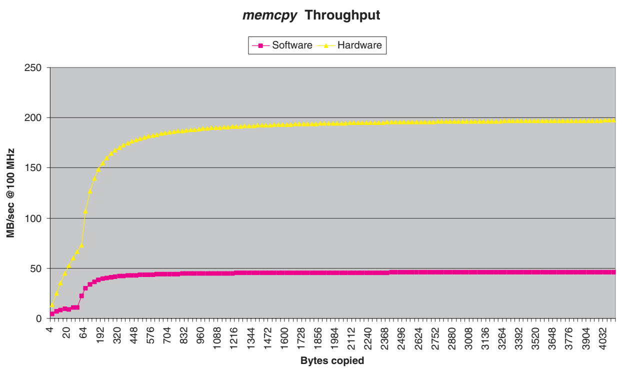

Figure 5. *memcpy* **throughput**

percentage that a copy of 1, 8 (one cache-line), 40 (5 cache-lines, typical copy size) and 8192 (maximum allowed copy size) words take in software and in hardware.

Generalizing, a *memcpy* performed in hardware takes 28 clock cycles of setup time plus 2 clock cycles per word. Figure 5 depicts the performance of the software and hardware implementation of *memcpy* for different number of words, represented in bytes instead of words. As expected, the benefit of our solution increase with the size of the copy but it stabilizes at around 200 MB/sec.

We also compared our solution with the AltiVec [10] solution of *memcpy*. In [10], the authors present values for a copy of 160 bytes at about 586 MB/sec (the picture presented does not allow to determine exact values). As the PPC runs at 750 MHz in their implementation, this corresponds to approximately 210 clock cycles. Our PPC is running at 100 MHz and we can perform a *memcpy* in 108 clock cycles, for the same 160 bytes. Concluding, we can perform approximately 48% better than [10], in terms of clock cycles, although the throughput we can achieve is only 148 MB/sec (due to the difference in the PPC clock frequencies).

7 Conclusions

In this paper, we presented a solution to perform non cache-line aligned *memcpy* operations using an additional indexing table in an existing cache organization. Our solution allows *memcpy* to be performed on word aligned data and performs a *memcpy* of one word and of eight words (one cache-line), 66% and 69% 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] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, pages 23–29, June 1989.

[2] F. Duarte and S. Wong. Profiling Bluetooth and Linux on the Xilinx Virtex-II Pro. In *Proc. IEEE Euromicro Conference on Digital System Design*, pages 229–235, 2006.

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

[4] P. Mackerras. Low-Level Optimizations in tehe PowerPC Linux Kernels. In *Proc. Linux Symposium*, pages 321–331, 2003.

[5] 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, Apr. 1999.

[6] 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 International Conference on Supercomputing*, pages 243–250, 1998.

[7] 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.

[8] 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 International Parallel Processing Symposium*, pages 308–315, 1998.

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

[10] Enhanced TCP/IP Performance with AltiVec.

[11] S. Wong, F. Duarte, and S. Vassiliadis. A Hardware Cache *memcpy* Accelerator. In *Proc. IEEE International Conference in Field Programmable Technology*, pages 141–147, 2006.