A LOAD/STORE UNIT FOR A *MEMCPY* HARDWARE ACCELERATOR

Stamatis Vassiliadis [1]*, Filipa Duarte* [2]*, and Stephan Wong* [1]

Computer Engineering
Delft University of Technology
emails: [1] {S.Vassiliadis, J.S.S.M.Wong}@ewi.tudelft.nl [2] F.Duarte@ce.et.tudelft.nl

ABSTRACT

Recently, a dedicated hardware accelerator was proposed that works in conjunction with caches found next to modern-day microprocessors, to speedup the commonly utilized *memcpy* operation. The main assumption of the proposal was that the to-be-*memcpy*-ed data has to reside inside the cache, which is not always valid. In this paper, we present a dedicated load/store unit and its implementation which cooperates with the previously proposed *memcpy* hardware accelerator and cache to ensure that data becomes available in the cache. Experimental results, using synthetic benchmarks, show that the load/store unit in conjunction with the *memcpy* hardware accelerator is capable of reducing the *memcpy* latencies by 85% (when the data is not present in the cache) compared to a highly optimized, hand-coded in assembly software solution.

1. INTRODUCTION

Profiling investigations of network related applications show *memcpy* as one of the most time-consuming function/operation in certain applications [1], [2]. The *memcpy* function is responsible for copying data of size *size* from memory address *src* to memory address *dst*. This function has undergone extensive research in order to find a suitable optimization in software. The most utilized solution is to hand-code this function in assembly, and exploit machine-depended features, and link it to the program instead of compiling the straightforward C code. This will result in more efficient code, however the optimizations are only valid for the selected machine.

In [3], a dedicated hardware accelerator was introduced to efficiently perform the *memcpy* function working in tandem with a cache that is nowadays commonly found next to many microprocessors (both general-purpose and embedded ones). This work assumed that the to-be-copied data was already operated upon before being *memcpy*'ed, implying that it is already present in the cache. However, this is not always true. Therefore, a solution must be sought after to address this. Moreover, as the mentioned hardware accelerator is tightly coupled with a cache, a write to either the source or destination address must result in the invalidation of the data in the cache and the data must be correctly stored back into the main memory.

We created a synthetic benchmark (its behavior derived from the Bluetooth protocol) which we compared the performance of utilizing an optimized hand-coded version of the *memcpy* (optimized for our utilized platform) with the performance of our complete hardware *memcpy* solution. The complete solution (the custom cache, the *memcpy* hardware accelerator, and the load/store unit) has the following advantages:

- the load/store unit (in conjunction with the custom cache) determines whether there is a need to load all the requested data by the *memcpy* from the main memory. When it is necessary, the unit only 'suffers' once from the read main memory latency after which each subsequent clock cycle yields a newly loaded word (only restricted by the available bandwidth).

- the *memcpy* hardware accelerator avoids duplicating data in cache, 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 real copies or to deal with the *memcpy* loads and stores.

- the synthetic benchmark achieves a speedup of approximately 7 in hardware than an optimized software implementation.

The paper is organized as follows. In Section 2, we present the related work and in Section 3, we present the concepts behind both the *memcpy* hardware accelerator and the load/store unit. In Section 4, we discuss the platform used for the implementation of the proposed solution and show the implementation details of each module of the system. In Section 5, 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 6, we draw some conclusions.

2. RELATED WORK

In this section, we first present some of the solutions proposed to reduce the impact of the *memcpy*, either being hardware or software, and secondly, an overview of still ongoing research on load/store units and their hardware implementations. As several memory controllers and load/store units can be found in literature, this related work section focus on the hardware design of such units.

Hardware optimizations for memory copies, besides DMA support, include the use of vector processors. Specifically for the PPC, the Velocity Engine (also known as AltiVec) 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 [4], [5]. Another solution was presented by [6], that introduced a new portable communication library, providing one-sided communication capabilities for distributed array libraries. Moreover, support was added for remote memory copy, accumulate, and synchronization operations optimized for non-contiguous data transfers.

Load/store units are still a research topic mainly on power- or energy-aware [7] area and on speculative loads [8], [9]. Some work has been performed on improving the load/store queues [10]. However, on hardware design of a load/store unit little literature can be found. In [11], the authors present a hardware load/store unit for the SCIMA memory architecture that the same authors presented. Another work involves a hardware design for a load/store unit [12]. This general-purpose load/store unit was developed for a Virtex II Pro FPGA, while the FPGA we used in our custom *memcpy* load/store unit is a Virtex 4. A complete memory system, for a multiprocessor platform using the Virtex II Pro FPGA was presented in [13]. This work implements a cache and a shared memory for the two processors on the Virtex II Pro.

3. CONCEPTS

In this section, we will present a short description of the previously published *memcpy* hardware accelerator and discuss the concepts behind the proposed load/store unit. In [3], we proposed a hardware solution to perform *memcpy* operations of entire cache-lines. Our solution stemmed from the simple observation that in some cases the data to be copied (of size *size*) from a source address (*src*) to a destination address (*dst*) was 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 even 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.

The *memcpy* hardware accelerator 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 indexing 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. For details an interested reader is directed to [3].

The *memcpy* hardware accelerator previously presented assumed that the data was already present in cache. This may be true in some cases, however, it is not for all cases. As such, we present in this paper a solution to load and store the needed data in/from main memory. The load/store unit is responsible for loading/storing data from/to the main memory, respectively. It is situated between the custom cache and the main memory and communicates control signals to direct the behavior of the latter referred modules. In addition, it controls the *memcpy* hardware accelerator. The load/store unit is mainly an finite-state machine (FSM) with two different paths for the read and write operations. Furthermore, certain states were specifically introduced to handle specific situations particular to the *memcpy* hardware accelerator, e.g., the presence of data in the cache or in the indexing table. Extra-read and -write states are introduced for the controller to wait for the hit or miss information from the cache or from the *memcpy* hardware accelerator.

On a read operation, if the address requested by the processor exists in cache or in the indexing table the data is immediately provided to the processor. If it is not in cache nor in the indexing table, the controller is responsible to request the data (a cache-line) from the main memory. However, if a *memcpy* is being executed, the size requested can vary from no request to main memory (because all data for the

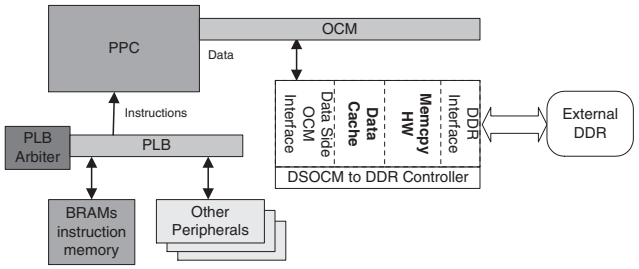

Fig. 1. System used to experiment the *memcpy* hardware and the DSOCM2DDR controller

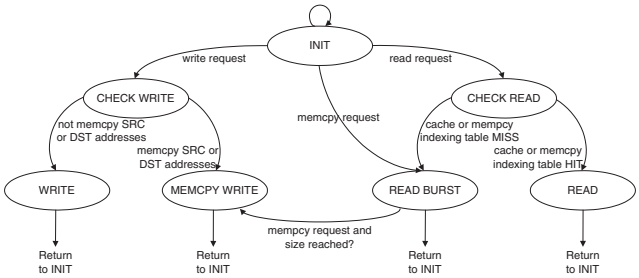

Fig. 2. States and states transitions of the load/store unit, including the *memcpy* states

memcpy is already in cache) up to the *size* of the *memcpy*. Finally, the controller enters the final state, in order to finish the execution of the *memcpy*.

On a write operation, the controller is responsible of storing the data to the main memory. If a *memcpy* is being executed, the controller is responsible to write to the main memory the data pointed to by the current access to the indexing table (if any). If the write address is a *src* or *dst* address, the controller also has to write to the main memory the data pointed to by the current access to the indexing table and update the new value in the main memory.

4. IMPLEMENTATION ENVIRONMENT

In this section, we describe the system used to experiment both our custom cache implementation, the *memcpy* hardware accelerator, and the load/store unit (from now on referred to as DSOCM2DDR controller - its name derive from the fact that we utilize the Data-Side OCM bus of the Virtex 4 to connect to a specially designed DDR controller).

We implemented the DSOCM2DDR controller on an the ML410 board containing a Virtex 4-FX FPGA with two PowerPCs (PPCs) 405 cores, although only one is used. The PPC is running at 100 MHz and, as we implemented our own custom cache in order to have control/access over it, we disabled the PPC internal caches (both the instruction and data caches). Figure 1 depicts the described system.

We implemented a standard 32 KBytes direct-mapped write-through cache with 32 bytes cache-lines, and included a custom part in order to support the *memcpy* hardware accelerator. The main differences between our custom cache and a standard one are, first, the use of a dual-ported memory for the tag memory and, second, some changes in the controller states in order to handle the *memcpy* hardware accelerator. The first change allows for checking whether the data is in the cache while loading the data to a different address. The second change involves supporting loads of more than one cache-line (until the *size* of the *memcpy* is reached). The number of cache-lines and the number of entries of the indexing table have to be the same in order for the system to work properly.

On a *memcpy*, the cache controller operates as follows depending on whether the *src* address hits or misses. When the *src* address hits in the cache:

- Check the consecutive address;
- Check if the *dst* address is in use on the *memcpy* indexing table and if it is, provide its content to the memory controller.
- Repeat the previous steps until the *size* is reached.

When the *src* address misses in the cache:

- Wait until the main memory provides the data and then enable the cache data-memories to write it.

Figure 2 depicted the FSM and its transitions between states of the DSOCM2DDR controller.

One particular implementation detail is that we utilized specific addresses to control the execution of the *memcpy* hardware accelerator. This implies that the main memory cannot utilize these addresses as standard addresses (these are not written or read to/from the main memory, therefore the controller stays in the INIT state). Only when other addresses are put on the bus, the controller changes state.

Subsequently, we present the detailed operations done in the WRITE, MEMCPY_WRITE, READ_BURST, and READ states. In the WRITE state, the controller:

- sets the write request signals to the main memory;
- returns to the INIT state, on a write_ack signal.

In the MEMCPY_WRITE state, the controller executes the following steps:

- If the address provide by the processor is a *src* or *dst* address of a previous *memcpy* then, the controller:
  - sets the write request signals to the main memory;

  - writes the cache-line provided by the cache in the main memory;
  - writes the changed word into memory;
  - finally, the controller returns to the INIT state.

- If the address provided by the processor is 0xA1FFFFF0 (i.e., the execution of a *memcpy*) and the *dst* address is in use in the indexing table (i.e., a previous *memcpy* exists) then:
  - sets the write request signals to the main memory;
  - writes the data provided by the cache in the main memory, until the *size* of current *memcpy* is reached;
  - returns to INIT.

In the READ_BURST state, the controller executes the following steps:

- If the address provided by the processor is not 0xA1FFFFF0, 0xA1FFFFF4, 0xA1FFFFF8, or 0xA1FFFFFC (i.e., not *memcpy* related addresses) then, the controller:
  - sets the read request signals to the main memory, until a cache-line size is reached;
  - returns to INIT.
- If the address provided by the processor is 0xA1FFFFF0 (i.e., start of a *memcpy*) then, the controller:
  - increases the *src* address and waits for a hit or miss in the cache;
  - if the address is not in cache, sets the read request signals to the main memory;
  - repeats the previous steps until *size* is reached;
  - goes to the MEMCPY_WRITE.

Finally, on the READ state, the controller:

- gives the data provided by the cache to the PPC;
- returns to the INIT state.

It is also worth noticing that the READ_BURST state benefits from the functionality of nowadays main memories, that allow bursts of data to be read. The controller generates addresses until the size is reached (either being the size of a *memcpy* or of a cache-line) which implies a delay to access the first word equal to the read latency of the main memory and the next requested word is provided every clock cycle. Another advantage of our solution is that the *size* of a *memcpy* is known in advance which enables the possibility of requesting all the necessary data and only paying once the initial main memory read latency.

5. RESULTS AND COMPARISON

We implemented the *memcpy* hardware accelerator, the custom cache, and the load/store unit in VHDL. We used ModelSim XE-III, a HDL simulation environment, that enables the verification of the HDL source code and functional and timing models of the designs. Both the software and the hardware implementation of the *memcpy* function, as well as the DSOCM2DDR controller, are analyzed using this tool.

In order to test our solution, we executed a *memcpy* of 4 cache-lines, and a sequence of reads and/or writes to either *memcpy*'ed addresses. The time required to perform the *memcpy* (in this case of 4 cache-lines or 128 bytes) in software is 415 clock cycles while in hardware it takes 169 clock cycles. The software version requires more accesses with smaller sizes to the main memory than the hardware version, thus implying a bigger total latency, due to the main memory latency of 9 clock cycles for the first word.

It is important to notice that the only difference between the software and the hardware implementation is the way the *memcpy* function is called. In the software, we use the function call *memcpy* implemented in assembly[1] and in order to program the *memcpy* hardware accelerator (i.e. pass the function parameters), the values are written to specific addresses.

In order to show the advantages of our solution (the custom cache, the *memcpy* hardware accelerator and the load/-store unit), we created a synthetic benchmark based on a real application. We based our benchmark on the data collect by [1] which describes the procedure to reassemble a frame of the Bluetooth protocol in the Linux OS. In the presented procedure reassembles a frame by calling the *memcpy* function 5 times. Therefore, we utilized 4 *memcpy* calls with a size of 352 bytes plus one of 96 bytes (corresponding of 4x11 cache-lines + 1x3 cache-lines in the hardware version). The Bluetooth protocol uses consecutive *src* addresses to reassemble one frame, which means that the data has to be loaded from memory for every *memcpy* (in the hardware case). For the *dst* addresses, the Bluetooth protocol also uses consecutive addresses, which means there is no need to write-back to memory (in the hardware case). For the hardware implementation, the complete synthe-tic benchmark takes 906 clock cycles while the software version takes 6263 clock cycles. This implies that, for this synthetic benchmark, the hardware implementation achieves a speedup of approximately 7 compared with the software version.

Subsequently, we present results for the best and worst cases in software and compare them with the worst case in

[1]The *memcpy* assembly code used is a changed version of the linux-2.4.20 for PPC. The change of the assembly code was necessary in order to take out the cache management instructions, because we implemented our own cache. The cross-compiler used to compile the C code is the powerpc-eabi-gcc with optimizations -O2.

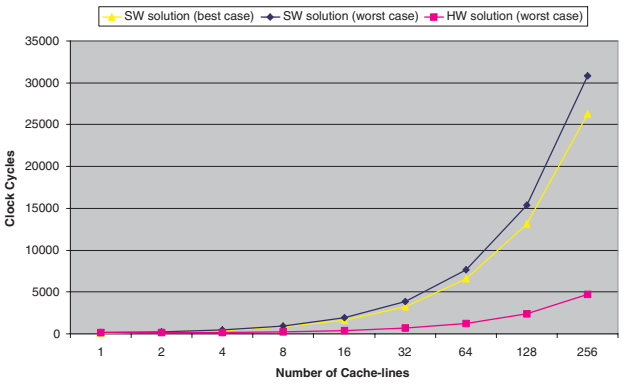

Fig. 3. Comparison between the software and the hardware solution for different values of *size*

hardware, for an increasing number of cache-lines. Figure 3 depicts the worst case scenario for the *memcpy* hardware accelerator, which is no data required for a *memcpy* is in cache (all the data has to be loaded from the main memory) and the *memcpy* is overwriting previously performed *memcpy*'s (every cache-line has to be written-back to main memory). The best case scenario for the software version is having all the required data in cache, while the worst case is not having it. As expected, the benefit of the hardware solution increases with the increase of the *size* of a *memcpy*. For a *memcpy* of only one cache-line, there is no clear benefit of the worst case hardware solution compared with either the best or worst case software solutions. However, for bigger *sizes* the hardware solution is capable of reducing the *memcpy*'s latencies by 82% (for the 256 cache-lines) compared with the best case for the software implementation of *memcpy* function.

6. CONCLUSIONS

In this paper, we proposed a new load/store unit that attached to a cache complements the *memcpy* hardware accelerator. The unit is able to autonomously perform loading of data from the main memory to support the *memcpy* operation when the data was not present in the cache. Moreover, as the size of the to-be-loaded data is known beforehand, the new unit is able to fully utilize the available bandwidth between the cache and the main memory. In addition, this approach allows the load latency to be reduced.

In order to experiment our system, we developed a synthetic benchmark (based on a the reassembly of a Bluetooth frame) and showed that the proposed hardware solution (a custom cache, the *memcpy* hardware accelerator and the load/store unit) provides a speedup of approximately 7, for the experimented benchmark, compared to an optimized (hand-coded in assembly) software solution. We also presented a comparison between the worst case in the hardware solution with the best case in the software solution, and show that the hardware solution brings increasing benefits for bigger number of cache-lines copies.

7. REFERENCES

[1] 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, pp. 229–235.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] J. Yang and R. Gupta, "Energy-Efficient Load and Store Reuse," in *Proc. IEEE/ACM International Symposium on Low Power Electronics and Design*, 2001, pp. 72–75.

[8] H. Hwang and J. Shann, "An X86 Load/Store Unit with Aggressive Scheduling of Load/Store Operations," in *Proc. IEEE International Conference on Parallel and Distributed Systems*, 1998, pp. 496–503.

[9] T. Moreshet and R. I. Bahar, "Effects of speculation on performance and issue queue design," *IEEE Trans. Very Large Scale Integration Systems*, pp. 1123–1126, Oct. 2004.

[10] C. Lemuet, W. Jalby, and S. Touati, "Improving Load/Store Queues Usage in Scientific Computing," in *Proc. IEEE International Conference on Parallel Processing*, 2004, pp. 38–45.

[11] T. Ohneda, M. Kondo, M. Imai, and H. Nakamura, "Design and evaluation of high performance microprocessor with reconfigurable on-chip memory," in *Proc. IEEE Asia-Pacific Conference on Circuits and Systems*, 2002, pp. 211–216.

[12] B. Donchev, G. Kuzmanov, and G. N. Gaydadjiev, "External Memory Controller for Virtex II Pro," in *Proc. International Symposium on System-on-Chip*, 2006, pp. 37–40.

[13] E. Vlachos, "Design and Implementation of a Coherent Memory Sub-System for Shared Memory Multiprocessors," Computer Architecture & VLSI Systems Laboratory, University of Crete, Tech. Rep., July 2006.