Memory Copies in Multi-Level Memory Systems

Pepijn de Langen and Ben Juurlink
Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Abstract

Data movement operations, such as the C-style memcpy function, are often used to duplicate or communicate data. This type of function typically produces a significant amount of off-chip traffic. For current microprocessors, communication with off-chip memory is an increasing limitation to attain higher performance as well as a significant source of energy consumption. To decrease the amount of communication between a CPU and the off-chip memory system, we propose a system that implements a hardware memcpy in the memory level where the source data is located.

1 Introduction

The increasing disparity between the speeds of processors and memories is a well known problem in the area of computer architecture. For operating systems, being very memory and I/O intensive, this problem is even more severe than for user applications [10]. Accesses to off-chip memory are not only posing an impediment to higher performance, they also contribute significantly to the power budget [7, 14].

Block copy operations, such as the C-style memcpy function, are often used to duplicate or communicate data in operating systems, message passing systems, webservers and database applications. Desktop and embedded applications also often use of this function, but mostly for smaller blocks of data. Although a memory copy is computationally one of the least intensive functions, current implementations still require all data to be loaded into registers and to be written back to main memory afterwards. This is wasteful, since it requires significant memory bandwidth and hardly any processing power. As a result, this can render the CPU idle, waiting for data from the congested memory system. Furthermore, current software implementations store the source data and often also destination data in cache. Especially with large copy operations, this data will be evicted from the cache before the next use. This is often referred to as *single-usage cache pollution*. Due to the increasing disparity between the speeds of processors and memories, applications are expected to spend an increasingly large fraction of their time in memory-bound functions like memcpy.

Our goal in this work is to reduce the amount of traffic between processors and off-chip memory for functions like memcpy. One of the most common ways to reduce off-chip traffic is to try to keep the required data close to the CPU. This is commonly done with caches, and by adapting their designs, parameters and/or policies to specific application behavior. One of the key problems with this approach is that there is no or very limited possibility for programmers to influence the dynamic behavior. Another approach is to employ scratchpad memories [2], in which transfers between on- and off-chip memories are programmed explicitly [6]. The downside of this, however, is that all memory transfers have to be programmed and that it is often not known in advance which data is needed and if it will fit in the on-chip memory. DMA is not suitable for use in memcpy due to the significant time required to initiate a transfer, as is explained in [15]. Our approach is different in that we perform block memory operations close to the actual data. More precisely, we propose to copy the data in the highest memory level where it is found.

Several other researchers focussed on optimizing the memcpy function. Calhoun et al. [3] showed that operating systems spend a significant amount of time copying data between buffers, and proposed to dynamically select the proper memcpy algorithm based on the size of the data as well as on predicted reuse, and to use this in a system with write combining buffers and non-temporal stores. While being able to reduce cache pollution and hence improve performance, this can only reduce the amount of off-chip traffic by not fetching the cache line associated with the destination addresses. Piquet et al. [12] proposed a general method to reduce single-usage cache pollution by bypassing the L2 cache. Duarte et al. [4] proposed to store pointers to copied data in a separate table to avoid having several copies of the same data in the cache. Future read accesses to the

1-4244-1898-5/08/$20.00 ©2008 IEEE

copy are dynamically diverted to the original source. While this method improves the performance of some kernels, the actual copy is only delayed, and hence there is no reduction in traffic. In all these works, copied data still has to travel from the main memory to the CPU and back. Zhao et al. [15] recently showed the performance benefits of having data movement functions implemented in a hardware copy engine. They presented a thorough overview of the possible implementation choices. Unlike our approach, these authors propose to implement the copy engine *either* close to the on-chip *or* close to the off-chip memory. In our approach, the copy engine is placed next to the on-chip *as well as* next to the off-chip memory. If the source is located in on-chip memory, the system will decide to perform the duplication there. If the data is not found on-chip, the operation is diverted to off-chip logic. In this way, the amount of off-chip traffic can be minimized by only sending the corresponding addresses to the off-chip logic, instead of sending all data to the CPU and back.

Our proposed idea shows some similarities with *Processing-In-Memory* (PIM) techniques, such as IRAM [11], Active Pages [9], and FlexRAM [5]. However, in those approaches, the processing is performed in the lowest level memory. In our case, it is dynamically decided where to perform the copy, and this can happen anywhere in the memory system.

The main contribution of this work is that we propose to perform memory copies in hardware copy engines and to have these copy engines implemented in several levels of the memory hierarchy. Furthermore, we show how this technique reduces the amount of traffic between the CPU and the off-chip memory system.

This paper is organized as follows. The technique to perform memory copies by using existing cache mechanisms is explained in Section 2. The same section also explains how these copy engines can be used improve performance by implementing them in several memory levels. In Section 3, the experimental results are presented and analyzed. Section 4 concludes the paper and presents some directions for future research.

2 Copying in Multi-Level Memory Systems

A common operation in any application is to copy data from one place in memory to another. For larger data, this copying is often performed by calling the *memcpy* function. Software implementations of memcpy generally use an unrolled loop containing a series of loads followed by a series of stores. To maximally exploit data parallelism, some systems use optimized implementations of memcpy, using multi-word registers and corresponding loads and stores. However, most systems incur a significant penalty if the source or destination of a copy are not properly aligned to the granularity of memory accesses in the system. Alvarez et al. [1] proposed an extension to the Altivec SIMD instruction set to support unaligned memory operations.

We propose a novel principle for duplicating or moving data efficiently in a multi-level memory system. The method is based on the observation that copied data is not always required in the near future, and that it can be harmful to have several copies of the same data in small memories, especially if the data is large. Our proposal is to perform memory copies in hardware copy engines, one for each memory level.

First we discuss the organization of the copy engine, the instruction that uses the engine, and the performance improvements due to a single copy engine placed either next to the L1 cache or to the L2 cache. Thereafter, the proposed *Dynamic Copy Engine* is discussed, where a copy engine is placed in every cache level.

2.1 Copying Using Copy Engines

In this paper we propose to use copy engines that can copy a block of data of the same size as a cache line. The central part of this copy engine is a shift register twice the size of a cache line. The copy engine is placed next to the cache, as it reuses existing logic for reading and writing data from and to the cache.

The copy engine can be instructed to copy data by issuing a special *movblk* instruction. This instruction takes two operands: the starting address of the source and the starting address of the destination of the copy. While there are no restrictions on the address pointing to the source to be copied, the destination address should be aligned with a cache line (i.e.: it should point to the first byte in a cache line). The movblk instruction copies a block of data the size of a cache line and the destination of the copy is always exactly one cache line.

Figure 1 schematically depicts how a copy is performed in copy engine connected to the L1 data cache. In this example, the source is not aligned to a cache line boundary, and therefore the copy engine needs to fetch two different cache lines and combine these into one. The copy engine first loads a cache line into the left side of the buffer. The second cache line is loaded into the right part of the buffer. The buffer is then shifted a corresponding number of places to the left, and the left part is finally written back to the cache.

Performing copies of exactly one cache line has several advantages: First, by performing the copies in larger blocks, the speed of data copies can be improved significantly compared to using registers. Second, when copying in the L1 cache the destination cache line can be allocated without fetching data, since the whole line is overwritten. Third, by reusing the existing logic for reading from and writing

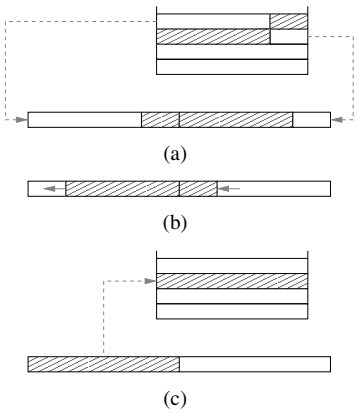

Figure 1. (a) The copy engine fetches 2 cache lines. (b) The double cache line is shifted to the left. (c) The result is written back.

to the caches, our copy engines can be implemented with little additional hardware. Fourth, by using blocks of the same size as a cache line, we make sure that the source data is located at a most two different cache lines. Using larger blocks can significantly complicate the copy process as the source data can be spread over more than two levels of the memory hierarchy. Whereas larger copies can be done by performing several copies, data smaller than a cache line cannot be copied by this copy engine.

An additional advantage of this copy engine, is that we do not incur a penalty when the data is unaligned. Aside from the special case where the source is located on only one cache line, the time required by the copy engine is independent of the data alignments.

To avoid hazards in an out-of-order processor, the movblk instruction is not allowed to bypass other instructions. Other instructions are allowed to bypass the movblk instruction, but the corresponding cache is locked for other accesses while the copy engine is executing.

2.2 Dynamic Copy Engines

The copy engine presented here can be integrated in any level of the memory hierarchy. When performing a copy on a small amount of data that was recently used, it makes sense to have a copy engine connected to the L1 data cache. When the copied data is not in the L1 cache or when the data is too large to fit in this cache, it makes more sense to perform the copy engine in a lower level cache.

For data residing in the corresponding cache, a copy engine can provide a speedup proportional to the number of words in a cache line. In case the data is not in the cache, however, the speedup decreases significantly or might even become negligible because of stalls in the memory system.

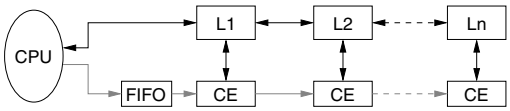

Figure 2. Implementation of copy engines in several layers of the memory hierarchy

Although other instructions might be allowed to execute in parallel, the CPU can be prevented from further execution due to these memory stalls.

In fact, a copy engine per se does not reduce the amount of traffic in the memory system. This not only poses a limitation to the attainable speedup and the scalability of the system. As noted before, memory traffic also accounts for a significant amount of energy, especially if between on- and off-chip logic. Energy consumption of is a growing concern, both for processors embedded in portable devices as for processors in high-end servers.

In order to minimize the amount of off-chip traffic, and at the same time improve performance, we propose the *Dynamic Copy Engine* (DCE). In the DCE, a copy engine is implemented in several levels of the memory hierarchy. This is schematically depicted in Figure 2. The first copy engine is employed with a FIFO buffer to allow pipelining several copy instructions. After receiving the operands, the first copy engine first checks if the source of the memory copy is present in the lowest memory level. In case the source is also aligned on a cache line boundary, this implies a singe cache access. Otherwise, two cache accesses are needed. If any source data is found in this cache, the copy is performed by the associated copy engine, which fetches the missing source data if needed. If no source data is found in this cache, the instruction is diverted to the copy engine associated with the next level memory, where the same procedure is repeated until the last copy engine is reached. Instead of moving the source data from a higher level memory to the CPU and the destination back from CPU to memory, we propose to move the *operation* down the memory hierarchy and to perform the copy close to the memory where the source is found. This way, the amount of traffic is reduced significantly by only sending addresses to the correct copy engine, instead of transferring the data from lower level memory to the CPU and back.

When the source is not aligned on a cache line boundary and the copy engines needs to perform two fetches, it may happen that the first cache line is found while the second is missing. In the method described above, the missing cache line would then be fetched from the lower memory level and stored into the cache. However, when this instruction is part of a group of several movblk instructions used to copy a larger block of data, the next movblk instructions will all

find part of their source data in cache. Since a large part of a copied block may actually not have been in the cache at the time of the first copy, this could severely reduce performance. To make sure that one movblk instruction does not influence the decision on the next one, we propose a variation on the DCE, which we call the *Dynamic Copy Engine with Non-Temporal loads* (DCE-NT). In the DCE-NT, missing source data is never stored into the cache. When a copy is performed and the corresponding cache is missing part of the source data, this data is fetched from the next level but is only sent to the copy engine.

Reducing the amount of communication between different memory levels can significantly improve performance. However, several other factors can be of influence: When the source data is not found in the a cache, it is also not allocated in this cache. This can reduce cache pollution if the copied block is large, and if the data associated with the source address is not required in the near future. Furthermore, when a data cache contains the cache line corresponding to the destination of a copy, but the copy is performed in a lower level, the cache line can be discarded even if it contains 'dirty' data. Because the data is to be overwritten, there is no need to write back the old data.

Hazard control in dynamic copy engines is done in the same way as described above for the normal copy engine: a cache is locked for other accesses while the corresponding copy engine is executing. Due to the ordered nature of the memory system, a processor is safely allowed to execute other code using L1 while copies are being performed in a lower level cache.

3 Experimental Evaluation

We have performed several experiments to evaluate our proposed design. These experiments have been performed using the out-of-order simulator from the SimpleScalar tool set [13] that was substantially modified to support the movblk instruction. We also modified the memcpy function to check for sizes and alignments, and to use the movblk instruction to copy data whenever possible. Furthermore, the linker was instructed to redirect calls to memcpy to our new version of the memcpy function.

For these experiments, we simulate a 4-way issue out-of-order processor with a two-level cache hierarchy. The L1 data and instruction caches are 32kB each, and connect to a 1MB unified L2 cache. All caches follow the *write-back* and *write-allocate* policies. The L2 cache is assumed to be located off-chip. For the benchmarks used in this work, increasing the L1 cache size or adding more levels of on-chip cache does not improve performance. Therefore, the results presented here are comparable to a system that for example has on-chip L1 and L2 caches and an L3 cache located off-chip. We assume a latency of 10 cycles for transferring addresses or data between on- and off-chip logic. Table 1 lists the most important parameters of the memory system.

Table 1. Properties of the memory system

Memory Ports to CPU	2
L1 Data Cache	32kB 4-way associative, 32B lines, 2 cycle latency
L1 Inst. Cache	32kB 4-way associative, 32B lines, 2 cycle latency
L2 Unified Cache	1MB, 8-way set-associative, 64B lines, 15 cycle latency
Memory Latency	100 cycles
Off-Chip Latency	10 cycles

We use three different implementations of our proposed copy engine for this section: a system with only a copy engine in the L1 cache (CE-L1), a system based on the dynamic copy engine that can copy in both L1 and L2 (DCE), and the same system using the non-temporal version of the dynamic copy engine (DCE-NT). In case of the dynamic copy engines, the decision where to perform the copy depends on the availability of the source data in the L1 cache, as was explained in Section 2. We rephrase the procedure here to provide some details:

- Up to two lookups are performed in L1 to find the source data. In case of one or more hits, the copy is performed in L1. If a miss was encountered, the second cache line is fetched from L2. The DCE stores this line in L1, while the DCE-NT only stores it into the copy engine buffer. Finally, the L1 copy engine writes the result into L1. The total time required to perform this copy is the sum of the two to three cache accesses plus two cycles to realign the final data. It should be noted that only one of these cache accesses may result in a miss, since at least part of the source is in the cache and since the destination cache line can be allocated without fetching data.

- When the L1 data cache contains no source data, the L1 cache is checked again to see if it contains the destination cache line. If this is the case, this line is invalidated. The copy instruction is then propagated to the copy engine corresponding to the L2 cache, where the copy is performed. For copying data in L2, first two or three accesses to the L1 cache are required. Then, 10 cycles are accounted for sending the operands to the off-chip logic. In there, the copy requires again up to three cache accesses, out of which two may result in a cache-miss. We again assume that the realignment of the destination data is done in two cycles. In general, the time required for copying in L2 depends foremost on the availability of the source data in L2. It should be noted that after the copy instruction is sent to the L2 copy engine, the CPU may continue execution.

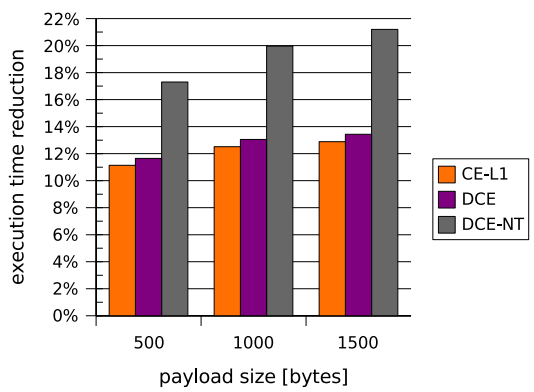

Figure 3. Execution time reduction

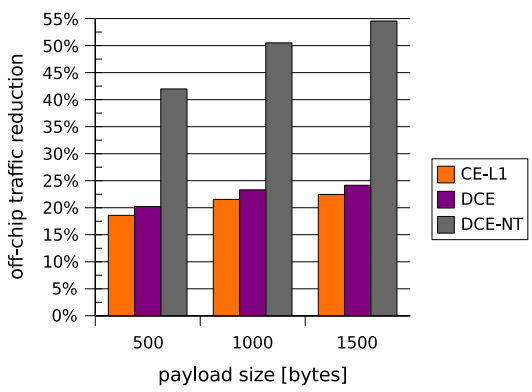

Figure 4. Off-chip traffic reduction

We conservatively assume that the two cache accesses to read the source data are performed in series, although they could be performed in parallel by using two-bank interleaved caches [1].

Since memory copies are often used in operating systems, message passing parallel systems, webservers, and databases, it would be best to evaluate our system with those applications. However, in SimpleScalar it is not possible to run these experiments. Therefore, we resort to using a TCP/IP processing workload as was also used in [15], which resembles part of the typical processing in an operation system kernel. This workload is derived from the FreeBSD stack [8]. The traces used in these experiments consist of 50,000 packets each, with fixed payload sizes of 500, 1000, and 1500 bytes.

Figures 3 depicts the execution time reduction for the different copy engines when processing TCP/IP payloads of varying sizes. All three implementations substantially improve performance, and for each implementation the improvement increases for larger payloads. This is expected, since every call to memcpy induces overhead for checking the data size and alignments. For larger copies, this overhead is relatively less. Furthermore, for larger payloads the memcpy function constitutes a larger part of the workload.

The implementation with only an L1 copy engine (CE-L1) improves performance upon the software implementation by between 11 and 13%. The most significant part of this speedup is due to the fact that the copy engine avoids fetching data corresponding to the destination cache line. This can also be seen from Figure 4, which depicts the relative savings in off-chip traffic (i.e.: the total number of bytes transferred between on-chip L1 and off-chip L2) for all three implementations. The numbers in this figure include transfers for both data and instructions, and include the corresponding addresses. By not fetching the old data corresponding to the destination cache line of a copy, this implementation reduces the amount of traffic by around 20%. We note that the same saving could be attained for the software routine on a system that has write-combining write-buffers and supports non-temporal stores. In this case, however, the system can suffer significant performance loss if the copied data is accessed afterwards.

Compared to the CE-L1 implementation, the dynamic copy engine (DCE) does not provide a significant additional improvement. In this case, the ability to perform the copies in an off-chip cache brings a small additional reduction in the amount of traffic, and hence also a small improvement of the execution time. The main impediment to higher reduction here, is that the vast majority of copies are still performed in L1. As explained before, when a larger copy is performed by a group of movblk instructions, a single movblk instruction may cause all following ones to be executed in L1 as well. Figure 5 depicts the percentage of all movblk instructions that is performed in the off-chip L2 cache. For all payload sizes, the normal DCE only performs 6% of the copies in L2. As most data may not have been located in L1 originally, this clearly limits the potential speedup and traffic savings.

When the copy engine is instructed to not store source data in the L1 cache, the percentage of copies performed off-chip increases dramatically. Figure 5 shows that the number of copies performed in L2 is more than 86% for smaller payloads, and reaches up to 94% for larger payloads. Correspondingly, the reduction in execution time attained by the DCE-NT is also much higher. For smaller payloads the DCE-NT improves performance by more than 17%, while for larger ones it improves by more than 21%. By performing the majority of copies in L2, the DCE-NT also reduces the amount of off-chip traffic significantly. Instead of fetching the source data into the processor and writing back the destination, the DCE-NT only sends 2 addresses to the off-chip logic. This way, the amount of traffic between L1 and L2 can be reduced by 42% for smaller payloads, and by more than 54% for larger ones.

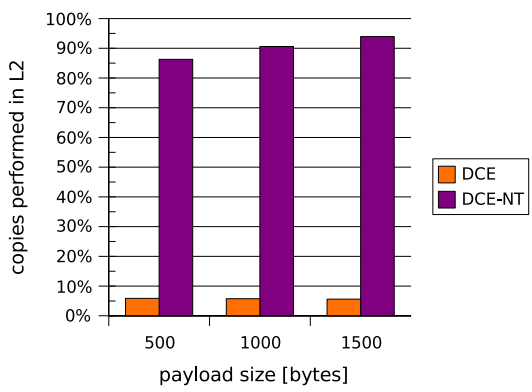

Figure 5. Percentage of memory copies performed in L2

4 Conclusions

Duplicating data is a common operation found in almost all applications. In some applications, like operating system kernels, this is even the most time consuming function. Unless effort is spent on reducing the number of memory transactions, the growing disparity between the speeds of processors and memories will disallow any further speedup. Moreover, a significant part of the total power budget is used for transactions in the memory system.

We proposed the use of copy engines built alongside the cache, that can perform data copies to exactly one complete cache line. Using this copy engine, we proposed a system called the dynamic copy engine, which implements copy engines in several levels of the memory hierarchy. By performing the copy in the highest level memory that contains part of the source data, the amount of communication between different memory levels is significantly reduced. This reduction in memory traffic not only improves performance, but also reduces a significant amount of energy involved with transferring data between different memories.

Detailed experimental evaluation of dynamic copy engines in a two-level cache system showed that this approach reduces the amount of off-chip traffic by up to 94% and improves the execution time by more than 21%.

For multiprocessor or multicore systems, reducing the load in the memory system may not only benefit the node that issues the copies, but also other nodes. Moreover, since memory copies are used extensively in shared-memory multiprocessor systems to communicate between different threads, additional benefits may be expected by using dynamic copy engines. We plan to investigate the possibilities of multicore systems with dynamic copy engines in future research.

Acknowledgments

This research was supported in part by the Netherlands Organisation for Scientific Research (NWO). We thank Li Zhao for providing the TCP/IP processing benchmark.

References

[1] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. Performance Impact of Unaligned Memory Operations in SIMD Extensions for Video Codec Applications. In *Int. Symp. on Performance Analysis of Systems & Software*, pages 62–71, 2007.

[2] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: A Design Alternative for Cache On-Chip Memory in Embedded Systems. In *Proc. Int. Symp. on Hardware/Software Codesign*, pages 73–78, 2002.

[3] M. Calhoun, S. Rixner, and A. L. Cox. Optimizing Kernel Block Memory Operations. In *Proc. 4th Workshop on Memory Performance Issues*, 2006.

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

[5] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an Advanced Intelligent Memory System. In *Int. Conf. on Computer Design*, pages 192–201, 1999.

[6] T. J. Knight et al. Compilation for Explicitly Managed Memory Hierarchies. In *Proc. Symp. on Principles and Practice of Parallel Programming*, 2007.

[7] D. Liu and C. Svensson. Power Consumption Estimation in CMOS VLSI Chips. *IEEE J. Solid-State Circuits*, 29:663–670, 1994.

[8] M. K. Mckusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996.

[9] M. Oskin, F. T. Chong, and T. Sherwood. Active pages: a computation model for intelligent memory. In *Proc. 25th Int. Symp. on Computer Architecture*, pages 192–203, 1998.

[10] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *Proc. USENIX Summer Conf.*, pages 247–256, 1990.

[11] D. Patterson et al. A Case for Intelligent RAM. *IEEE Micro*, 17(2):34–44, 1997.

[12] T. Piquet, O. Rochecouste, and A. Seznec. Minimizing Single-Usage Cache Pollution for Effective Cache Hierarchy Management. Technical Report PI-1826, IRISA, 2006.

[13] SimpleScalar LLC. http://www.simplescalar.com/.

[14] T. Simunic, L. Benini, and G. D. Micheli. Energy-Efficient Design of Battery-Powered Embedded Systems. *IEEE Trans. VLSI Syst.*, 9(1), 2001.

[15] L. Zhao, L. N. Bhuyan, R. R. Iyer, S. Makineni, and D. Newell. Hardware Support for Accelerating Data Movement in Server Platform. *IEEE Trans. Comput.*, 56(6):740–753, 2007.