This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

Cache-based Memory Copy Hardware Accelerator for Multi-core Systems

Filipa Duarte, *Student, IEEE,* and Stephan Wong, *Member, IEEE*

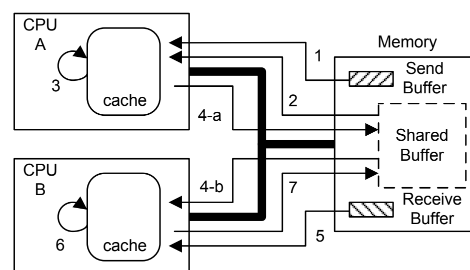
Abstract—In this paper, we present a new architecture of the cache-based memory copy hardware accelerator in a multi-core system supporting message passing. The accelerator is able to accelerate memory data movements, in particular memory copies. We perform an analytical analysis based on open-queuing theory to study the utilization of our accelerator in a multi-core system. In order to correctly model the system, we gather the necessary information by utilizing a full-system simulator. We present both the simulation results and the analytical analysis. We demonstrate the advantages of our solution based on a full-system simulator utilizing several applications: the STREAM benchmark and the receiver-side of the TCP/IP stack. Our accelerator provides speedups from 2.96 to 4.61 for the receiver-side of the TCP/IP stack, reduces the number of instructions from 26% to 44% and achieves a higher cache hit rate. Utilizing the analytical analysis, our accelerator reduces in the average number of cycles executed per instruction up to 50% for one of the CPUs in the multi-core system.

Index Terms—Hardware accelerator, cache, multi-core, TCP/IP, open-queuing theory

1 Introduction

The message passing protocol of a multiprocessor system is implemented through sending and receiving messages. One of the main bottlenecks identified of utilizing a message passing protocol concerns the memory copies as the messages to be sent or received are copied between the caches of the multiprocessor. The message passing protocol is based on two main operations, namely `send` and `receive` (or its variations, i.e., various forms of blocking and non-blocking operations). Only the structures (header queues and payload buffers) involved in the message passing `send`/`receive` operations are allocated in a shared address space during the initialization process. The header queues hold information about the messages (e.g., type, size, tag), and the payload for short messages. The payload buffers contain the payload for large data messages. There are several data exchange mechanisms and which one is used in a particular instance is determined by the implementation. The exchange mechanism depends on the size of the exchanged data and its performance depends on the underlying platform. The referred mechanisms mainly rely on copying the data to and from the header queues and the payload buffers and mainly the software `glibc memcpy` function is used to accomplish the copy.

A typical implementation of the mentioned mechanisms to `send` and `receive` data over a message passing protocol is depicted in Figure 1. The sending process copies the message along with other information required for message matching, to the shared memory area (Shared Buffer in Figure 1). The receiver process can subsequently match the tags of the posted received message and, accordingly, copy the correct message to its own buffer (Receive Buffer in Figure 1). Although this approach involves minimal setup overhead, it requires the use of at least two copy operations that will keep the CPU[1] performing the memory copy busy for the duration of the copy. As the trend is to increase the number of CPUs in a single chip, the impact of memory copies is expected to increase for multi-cores.



1 - Bring the Send Buffer into the cache A, if not there;
2 - Bring the Shared Buffer into the cache A, if not there;
3 - Write the Send Buffer into the Shared Buffer in cache A;
4-a - Update the content of the Shared Buffer in the memory;
4-b - Bring the Shared Buffer into the cache B, if not there;
5 - Bring the Receive Buffer into the cache B, if not there;
6 - Write the Receive Buffer into the Shared Buffer in cache B;
7 - Update the content of the Shared Buffer in the memory.

Fig. 1. A typical message passing protocol

It is important to notice that nowadays, such CPUs typically are tied with caches with varying sizes. As caches store the most recently used data, the majority of the data to be moved by a memory copy is already present within the cache. This is especially true when considering that such data must often be processed first.

Work developed at the Computer Engineering Laboratory of Delft University of Technology, The Netherlands. Filipa Duarte is currently with the Ultra Low Power DSP Group at Holst Centre/IMEC, The Netherlands.
E-mails: Filipa.Duarte@imec-nl.nl, J.S.S.M.Wong@tudelft.nl
Manuscript submitted April 7, 2009. Resubmitted on October 17, 2009. Revised on January 6, 2010. Accepted on January 11, 2010.

1. In this paper, a CPU - Central Processing Unit - can either be a core in a multiprocessor or an uniprocessor.

Digital Object Indentifier 10.1109/TC.2010.41 0018-9340/10/$26.00 © 2010 IEEE
Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on April 20,2010 at 12:02:58 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

The memory copy operation performed in the traditional way in software involves the utilization of many loads and stores by the CPU itself to accomplish the operation and results in cache pollution (overwriting the to-be-copied data before being copied) and/or in data duplication (the same data is present twice in the cache). Moreover, Direct Memory Access (DMA)-based approaches - typically used to reduce the cost of memory data movements and, therefore, a very utilized solution also for memory copies - provide a limited solution mainly due to 3 reasons:

- DMA controllers (DMAC) are peripheral devices and therefore there is a significant overhead on the communication between the CPU and the DMA controller, as its the initialization has to be performed explicitly;
- The notification of a DMA transfer completion is performed either through polling or an interrupt, both being expensive;
- DMACs deal mainly with physical addresses and therefore user-level application cannot take advantage of them.

In [1], [2] and [3], we proposed a solution that exploits the presence of a cache to perform memory copies. The architecture of our accelerator incorporates an indexing table to be inserted within an existing direct-mapped cache that points to the original data present in the cache. The following section describes the concept behind the accelerator and its architecture.

1.1 Cache-based Memory Copy Hardware Accelerator: Concept and Architecture of the Indexing Table connected to a Direct-mapped Cache

Starting by assuming that the original data is already present in the cache (an initial assumption to not complicate the concept), the memory copy operation can be performed by utilizing an indexing table that is indexed by the copied data address and that stores the pointers to the cache locations containing the original data. Consequently, we can summarize that a memory copy operation is reduced to a single action of entering the copied data address into the indexing table and creating a pointer to the original data stored in the cache. When the copied data needs to be accessed, this can be easily performed by extracting the corresponding pointer from the indexing table and subsequently accessing the cache using this pointer. It must be noted that the actual storing of the copied data to the main memory is deferred to a later time in the proposed method. In particular, when either the original or copied data locations are being overwritten or when the original data location is evicted from the cache, this must be detected and the appropriate measures (explained in more detail in [2]) must be taken to store the copied data to the main memory. Furthermore, the pointer in the indexing table must be invalidated. An illustration of the proposed method is depicted in Figure 2 and a brief explanation of the system follows (for more details an interested reader is directed to [1], [2] and [3]).

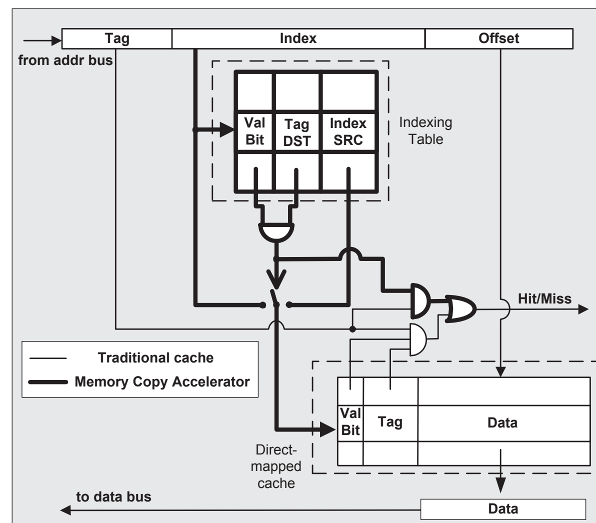


Fig. 2. The accelerator concept

A direct-mapped cache (thinner lines, gates and table in Figure 2) 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). 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 directory (on a write) or is used to compare with a tag already in the cache directory (on a read). Finally, the word to be supplied to the processor is determined by the offset.

The memory copy operation performs a copy of size `size` from a source address `src` to a destination address `dst`. The manner a copy is performed using the indexing table (**ticker lines**, gates and table in Figure 2) connected to a cache is to access the indexing table with the index part of the `dst` address and to write the index part of the `src` address, the tag part of the `dst` address and a valid bit in the entry accessed. The execution of a memory copy subsequently becomes the filling of the indexing table.

If there is a read hit in the indexing table (calculated based on: 1 - the tag part of the requested address; 2 - the tag part of the `dst` address stored in the indexing table; and 3 - the valid bit also stored in the indexing table), the index part of the `src` address stored in the indexing table is provided to the cache (this is the pointer to the cache entry). It is worth mentioning that, if there is a read miss on the indexing table, there will be no penalty in the performance of the system, as the indexing table and the cache are accessed in parallel. This implies that on a read miss in the indexing table, the cache is already being accessed and returns the data in the same amount of time. On a read hit in the indexing table, we

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on April 20,2010 at 12:02:58 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

require one more clock cycle in order to retrieve from the indexing table the correct address to provide to the cache. Without delving into too many details (the more interested reader can find more details in [1], [2] and [3]), we also developed a custom load/store unit that differs from a standard one in the following cases: (1) on the execution of a memory copy, the custom load/store unit is responsible for loading all the cache lines part of the `src` addresses (and write-back any previous data that these loads may evict from the cache); (2) on a miss that loads data from the main memory that evicts `src` addresses from the cache. The previously described solution was implemented in a Field-Programmable Gate Array (FPGA) as a proof-of-concept.

1.2 Contributions and Organization

In this paper, we demonstrate the benefits of our accelerator concept in a multi-core system. Moreover, we introduce a new architecture of the cache-based memory copy hardware accelerator tightly coupled with a set-associative cache. We demonstrate our solution based on open-queuing theory, where the values for the parameters involved in the analysis are gathered using Simics [4] full-system simulator by executing the receiver-side of the TCP/IP stack. The main contributions of the work described in this paper are:

- An accelerator able to support the message passing protocol on a multi-core system and achieving a reduction on the average number of cycles executed per instruction up to 50% for one of the CPUs in the system.
- Comparing the execution time of the receiver-side of the TCP/IP stack with and without our accelerator, we get the following benefits: 1) speedups ranging from 2.96 up to 4.61 times; 2) reduction of the number of instructions ranging from 26% up to 44%; and 3) higher cache hit rate. Moreover, the simulation of the receiver-side of the TCP/IP stack executed with our accelerator also provided the necessary values to correctly perform the open-queuing theory analysis.

The remainder of the paper is organized as follows. In Section 2, we describe the related work and in Section 3, we extend the previously presented cache-based memory copy hardware accelerator for a multi-core system. In Section 4, we present the analytical analysis performed for both a single CPU and a multi-core system. In Section 5, we first describe the setup utilized in a full-system simulator. Afterwards, we analyze the results of executing synthetic benchmarks and the receiver-side of the TCP/IP stack, in order to gather the necessary information to accurately model the multi-core system. Moreover, in this section, the results of utilizing our solution in a multi-core system supporting message passing protocol are also presented and discussed. The paper ends with conclusions in Section 6.

2 Related Work

Several works have been presented in analyzing the message passing protocol for hardware cache coherent multiprocessors. In this section, we highlight the more recent ones.

The authors of [5] and [6] compared the performance of the shared address space and message passing interface on a hardware cache coherent multiprocessor. Both studies concluded that removing the extra copy and using lock-free management queues in the message passing models can improve performance. The following situations impacted the performance of the systems analyzed: (i) remote accesses of cache line granularity on the remote data that does not have good spacial locality; (ii) the use of explicit transfers that either put data in the cache or in the main memory of the destination; (iii) cache conflict and cache coherence protocol behavior; and (iv) the usage of barriers. Moreover, the authors also concluded that the message passing model benefits from having a hardware cache coherent multiprocessor as it enhances locality. The authors of [7] presented 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.

One of the first machines supporting message passing protocols were the Cray T3D [8] and the Stanford FLASH [9]. The designers of both machines identified the need to alleviate expensive operations in the path of sending and receiving messages, in order to provide performance. For that, the solution relied on avoiding message copying through direct transfer of data between processes, and overlap computation with communication. The solution implemented in the Cray T3D was the use of a system level block transfer engine, which use DMA to transfer large blocks of contiguous or strided data to or from remote memories. On the Stanford FLASH [10], the designers implemented a custom programmable node controller containing an embedded processor that can be programmed to implement both cache coherence and message passing protocol.

However, the need for a data transfer engine is still a matter of debate. In [11], the authors analyzed the performance of integrating a data transfer engine in a system based closely in the Stanford FLASH architecture. According to the authors, the benefits of block transfer were not substantial for hardware cache coherent multiprocessors. The reasons given were (i) the relative modest fraction of time applications spend in communication; (ii) the difficulty of finding enough independent computation to overlap with communication latency; and (iii) the cache lines often capture many of the benefits of block transfer. However, in a more recent work presented by [12], the authors analyzed the performance of transferring large data in symmetric multiprocessors. The authors analyzed five different mechanisms (shared memory buffers, message queues, Ptrace system calls,

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on April 20,2010 at 12:02:58 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

kernel module copies and network cards) in terms of latency, bandwidth and cache usage. The main conclusion was that, as soon as the proper mechanism was chosen, these mechanisms did provide performance benefits contradicting the conclusions reached by [11].

We believe that a mechanism that uses the cache efficiently can correctly capture the block transfers and enhance performance. Therefore, we have coupled our accelerator with the second level cache in order to efficiently support the message passing protocol by reducing the time spend on performing the copies between the caches in the system.

3 Cache-based Memory Copy Hardware Accelerator in a Multi-core System

In this section, we start by illustrating the application of our accelerator concept to a multi-core system with 3 CPUs supporting a message passing protocol. Subsequently, we introduce the architecture of our accelerator to be tightly coupled with a set-associative cache. We conclude this section by presenting the hardware costs involved when implementing the proposed architecture.

3.1 Message Passing Protocol utilizing the Cache-based Memory Copy Hardware Accelerator

Consider a multi-core system consisting of a data Level 1 cache (L1$) tightly-coupled with each CPU in the system, a shared unified L2$ for all CPUs, and a bus-based interconnect network that provides the necessary functionality for snoop-based hardware cache coherence protocol. The targeted application is the `send`/`receive` message passing protocol. Figure 3 depicts the system and the application.

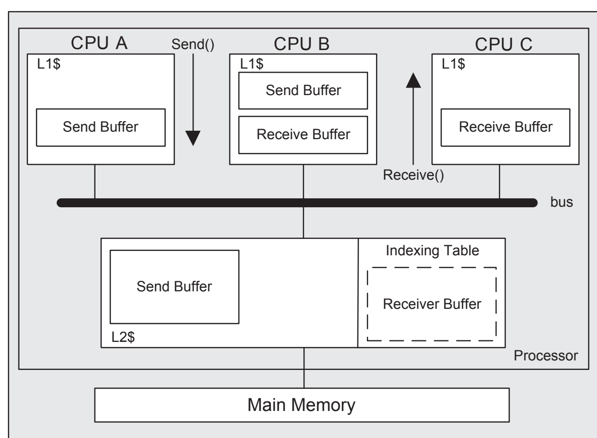


Fig. 3. Multi-core system with 3 CPUs supporting a message passing protocol

When executing the message passing protocol the cache of each CPU, i.e., the L1$, can have cache lines associated with either the Send or Receive Buffers involved in the `send` and `receive` operations of the message passing protocol. When a `send`/`receive` operation is issued by one CPU, all the CPUs in the system need to write-back any cache line in its L1$ that belongs to the Send/Receive Buffer and invalidate these cache lines. This can be achieved by changing the state of the cache line (part of the coherence protocol) in order to trigger a write-back of the cache line to the L2$. Moreover, the L2$ now has the necessary information on the Send/Receive Buffer addresses, therefore it can load the necessary data, if it is not there yet, and fill the indexing table. When the `receive` operation is issued by a particular CPU (Receive Buffer will miss the L1$ of all CPUs), it will trigger a request of the data on the bus that will be answered by the L2$. The requested cache line is loaded from L2$, using the indexing table, into the L1$ that requested it. This implies a change in the cache line state to Shared in the L1$ and to Modified in L2$. When a cache line in the L1$ part of the Receive Buffer, changes status from Modified to any other state, the L2$ needs to write it back to main memory. If another CPU requests a cache line that is in the indexing table, the L2$ answers the request and provides the cache line, changing the state of the cache line to the appropriated one.

The benefit of the presence of our accelerator is two-fold: (1) it reduces the amount of memory copies involved in the `send`/`receive` message passing protocol; and (2) it reduces the amount of loads and stores needed to bring the Send/Receive Buffers to the caches. Moreover, it increases the cache hit rate.

3.2 Architecture of the Indexing Table connected to a Set-associative Cache

In a typical multi-core system, as the one introduced in the previous section, the L2$ is normally a set-associative cache. However, the architecture of our accelerator presented in Subsection 1.1 is tightly coupled with a direct-mapped cache. Therefore, in this section we extend the previously presented architecture in order to support set-associative caches. The indexing table coupled with a 2-way associative cache with hardware cache coherence protocol is depicted in Figure 4.

As presented in Subsection 1.1, the traditional memory copy operation requires the following parameters: the size `size`; the source address `src`; and the destination address `dst`. Therefore, the indexing table is accessed by the index part of the `dst` address and contains the tag and index parts of the `src` address, the tag part of the `dst` address and a bit stating if it is a valid entry.

In a direct-mapped cache the number of entries of a cache is the same as the number of indexes, however in a set-associative cache this is not the case. In a set-associative cache, the index part of the address will only give access to a cache congruence class; the tag part of the address is the one that is used to distinguish which cache line is actually being accessed.

However, each entry of our indexing table points to a cache line, therefore, the number of entries of the indexing table has to be the same as the number of

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on April 20,2010 at 12:02:58 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

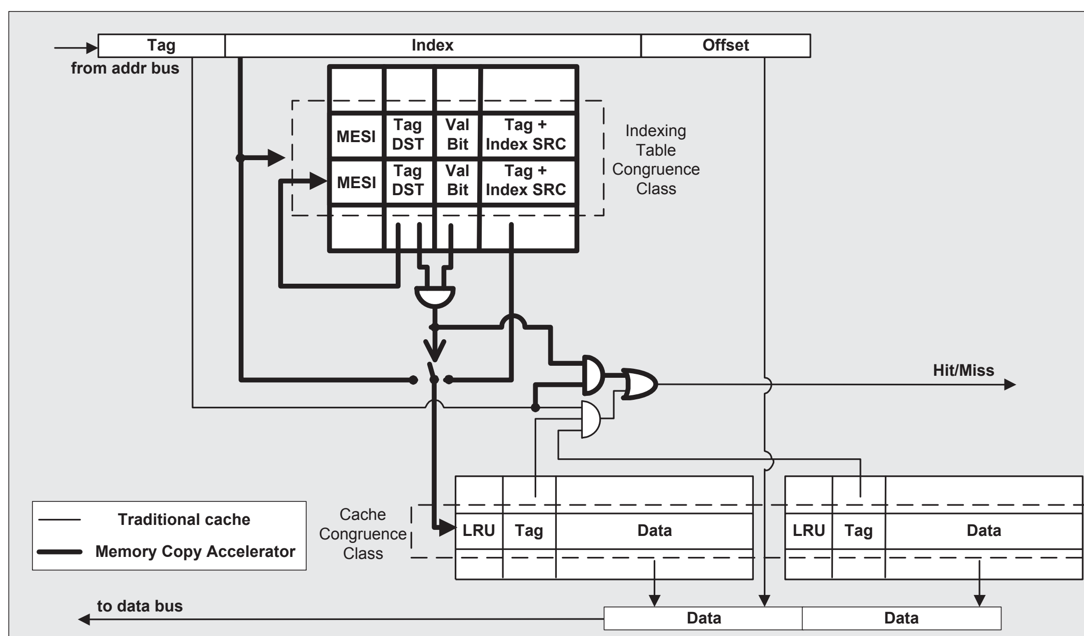


Fig. 4. Indexing table connected with a 2-way associative cache with support for cache coherence protocol

cache lines the cache has. Therefore, we introduced the same concept as the cache congruence class, referred to as indexing table congruence class in Figure 4. The index part of the address is used to select an indexing table congruence class which contains as many entries as the number of sets the cache has.

When filling in the indexing table (i.e., when performing the copy), the index part of address selects the indexing table congruence class and its first entry is used, only if that entry is taken then the following entry is used. This has a performance impact on performing a copy as the time to fill the indexing table now dependents on the number of sets the cache has. However, the bigger the size of the copy the more time the CPU spends executing a memory copy in software. Therefore, the increase in time to fill the indexing table is still much smaller than the software version.

A read hit in the cache does not incur in additional delay because the indexing table and the cache are accessed in parallel. However, if there is a read hit in the indexing table, the delay of accessing the indexing table plus the cache has to be considered.

The index part of the address is used to identity an indexing table congruence class. As the number of entries inside of the indexing table congruence class might be large (because it depends on the number of sets of the cache), performing a sequential search (i.e., check each entry of an indexing table congruence class and compare the stored tag with the one requested to identify if this is the requested address) is not the most efficient way. A solution is to use a Content Addressable Memory (CAM) core for the “Tag DST” field (in Figure 4), which allows to search the contents of this field in one clock cycle in order to find a match between the tag stored in the indexing table and the tag of the requested address. This means that there will be as many CAM cores as the indexes with each CAM core having as many entries as the sets of the cache. Searching the “Tag DST” field, the first step to determine whether it is a read hit in the indexing table, can be performed efficiently in one clock cycle. The content of the “Tag DST” field, accessed based on the index part of the address, is afterwards (second step to determine a read hit in the indexing table) feed in parallel to the “Val Bit” and the “Tag + Index SRC” fields (see Figure 4) to determine the cache congruence class of the `src` address in the cache (and its tag part of the address). A read hit in the indexing table will subsequently have a latency of two clock cycles (one clock cycle for the CAM core of the “Tag DST” field plus one clock cycle for both “Val Bit” and “Tag + Index SRC” fields).

Another scenario that the indexing table has to handle is the case where a `src` address is evicted from the cache. In this case, the entry on the indexing table that contains the corresponding `dst` address has to be identified in order to actually perform the copy in the main memory (this is performed with a custom load/store unit presented in [1]). For this search in the indexing table to be efficient, the “Tag + Index SRC” field is implemented as a CAM core while all other fields of the indexing table can be Random Access Memory (RAM) cores. A write request to either a `src` or `dst` address will always trigger a write of the copied data to the main memory, performed by the custom load/store unit.

In addition, the indexing table is now required to support a snoop-based hardware cache coherence protocol. Therefore, we included in the indexing table a new field, referred to as “MESI” in Figure 4, consisting of two bits

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on April 20,2010 at 12:02:58 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

where the state of each cache line is kept, in order to support the modified, exclusive, share and invalidate (MESI) states of the protocol. This field is accessed the same way as the "Val Bit" and "Tag + Index SRC" fields are and is implemented as a RAM core.

3.3 Hardware Costs

In order to estimate the size of the indexing table for different caches organizations, a detailed study was performed. As presented earlier, the indexing table stores the tag and index parts of the addresses used in the memory copy. The number of bits stored has an impact on the size of the memories used to implement the indexing table. The bigger the number of bits stored the more hardware resources are necessary.

Assuming a cache size of 1 MB (a typical size for caches nowadays) and varying the cache line size (from 16 to 128 bytes) and associativity (from 4 to 32), the hardware resources are estimated for both the indexing table and the cache. In the interest of space, the figures depicting the increase of the hardware resources used in the indexing table compared with the ones used only in the cache are not presented, however the main conclusions reached are:

- The increase on associativity does not have a big impact on the percentage of hardware resources utilized on the indexing table compared with a cache (an increase of associativity from 4 to 32 implies an increase of 17% of the hardware resources utilized due to the presence of the indexing table). The reason for this is that in order to support associativity, the cache itself increases the number of bits stored in this way reducing the impact of the bigger number of bits also stored in the indexing table.
- The increase in the cache line size reduces the percentage of hardware resources utilized on the indexing table compared with the cache (from 20% with a cache line size of 16 bytes down to 2.5% with a cache line size of 128 bytes). This is expected as one cache line now contains more data, therefore one entry of the indexing table points to more data.

The increase in the percentage of hardware resources utilized on the indexing table compared with a cache for the different situations is mitigated due to the effective cache size increases for a program executing a large number of copies. Moreover, due to the effective increase of the cache size, it is expected that the hit rate of the cache increases.

4 Analytical Analysis of both Single CPU and Multi-core

In order to analyze the benefits that our accelerator can have in a multi-core system, we use the open-queuing theory to perform an analytical analysis of our system. Compared with data gathered by real system measurements, the accuracy of the open-queuing model was measured to be approximately 3%. The technique described was also compared with a closed-queuing model, where the results showed a 3% to 10% maximum difference (these results are demonstrated on [13] and [14]).

In the simplest form, any processing system can be considered to be composed of a number of servers with a specific service time. These servers can be any shared resources with a queue, such as a bus or a memory. The queuing theory correctly models the performance of such servers in a complex system and, therefore, can be utilized to model a processing system. The request rate to each server in the system is assumed to follow a Poisson distribution, which allows the use of relatively simple equations to depict queue delays. Studies by [15] and [16] present traces which closely approximate a Poisson distribution, justifying the choice for this model. For the cases where the traces deviate from a Poisson distribution, there is no simple method to evaluate the impact of the memory hierarchy. The service time of each server is assumed to be either constant or exponential (both analysis are performed). Therefore, the modeled system falls in Kendall's notation [17] M/D/1 (Poisson input, deterministic/constant service time, one server) or M/M/1 (Poisson input, exponential/Poisson service time, one server).

The ideal performance of a processing system is measured in cycles per instruction (CPI) when executed with an infinite cache ($CPI[\infty]$), i.e., when the L1$ acts as if there were no cache misses and therefore no fetch penalties. However, the actual performance of a system ($CPI[system]$) with a memory hierarchy is considerably less due to the stalls and delays caused by the cache misses and thus the fetching actions. In reality, a CPU and its L1$ are the sources that generate the outstanding misses to the memory hierarchy, and therefore also responsible by the fetching actions. This additional delay, measured as cycles per instructions, is typically known as the finite cache penalty (FCP). These two parameters are added together to obtain the actual processing system performance:

$$CPI[system] = CPI[\infty] + FCP \quad (1)$$

A problem in this type of analytical queuing analysis is to determine the queue delays at each server. In particular, the open-queuing theory requires that the number of requests at each server cannot be fixed at any point in the system. In fact, all queues must theoretically be capable of unlimited length. However, in a real multi-core system, each CPU typically permits only a fixed number of outstanding misses to exist within the memory hierarchy at any time. The reason is that the outstanding misses stall the CPU which cannot continue processing until the fetches (due to the misses) are fulfilled. This means that the maximum number of requests for fetching is fixed for all the queues within a memory hierarchy at any instant

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on April 20,2010 at 12:02:58 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

in time. Of course, the number of requests could be fewer than the maximum at any instant, only the maximum is fixed. Consequently, there is a negative feedback process in the memory hierarchy which in practice, guaranties small and self-limited queues. Consequently, there is no need to complicate the model using closed-queuing theory to model the system.

Another way of viewing the mentioned negative feedback is the following: the finite cache penalty delays depend on the request rates for the memory hierarchy, which are inversely proportional to the cycles per instruction - the smaller the cycles per instruction is, the faster the instructions are processed, thereby generating more memory hierarchy requests. The increase in memory hierarchy requests create larger queue delays, which subsequently increase the finite cache penalty value and the cycles per instruction value, reducing the queue delays, and so on and so forth. Consequently, the analysis require an iterative calculation, but it will converge in a maximum of 10 interactions.

4.1 Single CPU with and without our Accelerator, M/D/1 Model

In a system with a multi-level cache hierarchy, each main memory access can incur a different delay determined by which level of the hierarchy contains the desired data at that moment. The finite cache penalty delay is the weighted sum of the hits at each level multiplied by the delay per hit at each level. The finite cache penalty (FCP) equation for a single CPU with tandem caches (3 level hierarchy) is:

$$\begin{aligned} FCP &= (mr_1 - mr_2)T_2 + (mr_2 - mr_{main})T_{main} \\ &\quad + mr_{main}T_{main} \\ &= mr_1[(1 - \frac{mr_2}{mr_1})T_2 + \frac{mr_2}{mr_1}T_{main}] \end{aligned} \quad (2)$$

where the mr_k are the miss rates and T_k are the fetching times of each cache level, $k = \{1, 2, main\}$.

In a memory hierarchy in which misses can occur at various levels, the probability of a miss reaching any given cache level will vary as determined by the given miss rates for each cache. This is expressed as the visitation probability (V), which is simply the hit probability per L1$ miss rate. From Equation (2), the visitation probability (V) at each level can be determined to be simply:

$$\begin{aligned} V_2 &= 1 - \frac{mr_2}{mr_1} \\ V_{main} &= \frac{mr_2}{mr_1} \end{aligned} \quad (3)$$

To determine the fetching time of each level (i.e., the value of T_k, with $k = \{1, 2, main\}$), it is assumed that each cache is simply an individual server. In a simple open-queuing model, the queue length at an individual server is determined by the utilization (U), which itself depends on the request rate (R) and the service time (S) of the server. For a memory hierarchy, the visitation probability (V) has to be included in the utilization (U) of each server. For a single CPU with tandem caches (3 level hierarchy), the utilization (U) of each server is:

$$\begin{aligned} U_2 &= V_2 \times S_2 \times R \\ U_{main} &= V_{main} \times S_{main} \times R \end{aligned} \quad (4)$$

where the request rate (R) is given by $R = mr_1/(CPI[system] \times T_{CPU})^2$.

Assuming a constant service time (S), the fetching delay (T) for the single CPU with tandem caches (3 level hierarchy) case is:

$$\begin{aligned} T_2 &= V_2 \times S_2 \times (1 + \frac{0.5U_2}{1 - U_2}) \\ T_{main} &= V_{main} \times S_{main} \times (1 + \frac{0.5U_{main}}{1 - U_{main}}) \end{aligned} \quad (5)$$

The previously mentioned negative feedback can be monitored utilizing the queue size, as the queues sizes became a constant size when the system converges:

$$\begin{aligned} Q_2 &= \frac{U_2 - 0.5U_2^2}{1 - U_2} \\ Q_{main} &= \frac{U_{main} - 0.5U_{main}^2}{1 - U_{main}} \end{aligned} \quad (6)$$

Assuming a write-back L2$, when a cache miss requires the replacement of a cache line, this cache line must be written to a higher cache level. The use of our accelerator influences the rate that this happens. If the `src` address is not present in the cache at the moment of filling the indexing table, it has to be fetched from the main memory, which implies writing the evicted cache line back to the main memory. The same situation happens if a miss in the L1$ also misses in the L2$. This situation triggers a load of the missed cache lines and the replacement of the cache lines present in the cache. Both situations will probably evict some cache lines from the L2$. This is referred to as cast-out (CO) and typical values range from 20% to 40% (as reported by [13] and [14]). The utilization of our accelerator influences this parameter.

Moreover, due to the increase of the cast-out (CO) of the L2$, there will be an increase in the visitation probability (V) of the main memory. This is modeled through a factor of eviction (FE). As it will be demonstrated in Section 5, the values for the factor of eviction (FE) are also dependent of the application besides the use of our accelerator. Therefore, the visitation probabilities (V) presented in Equation (3), become:

$$\begin{aligned} V_2 &= 1 - \frac{mr_2}{mr_1} \\ V_{main} &= \frac{mr_2 + FE \times mr_2}{mr_1}, \text{ where} \\ &\quad FE = 0, \text{ if the accelerator is not used} \end{aligned} \quad (7)$$

2. Note that the request rate is inversely proportional to cycles per instruction (CPI) providing the negative feedback process in the memory hierarchy and the reason for an iterative calculation, as introduced early.

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on April 20,2010 at 12:02:58 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

A memory hierarchy that provides the critical word first is assumed. If the CPU requires another word in the cache line that is being transferred at the moment, the CPU must wait until it is available. This is referred to as trailing-edge (TE) delay, which typically ranges from 10% to 30% (as reported by [13] and [14]). With the use of our accelerator, it is necessary to also consider the case of a first read to a copied address that will always miss in all L1$. These accesses have go to the L2$ and if another word is requested while the cache line is being transferred, there will be an increase of the trailing-edge (TE) delay.

With the utilization of our accelerator, the copied data is accessible in the L2$ as well as the data that was already in the cache before the copy was performed. This means that both the `src` and the `dst` addresses are available through either the indexing table or the cache itself, without evicting any cache lines. This implies a decrease in the original L2$ miss rate (mr_2).

The total delay (T_{total}) is given by the summation of Equation (5), with the utilization (U_k) given by Equation (4) and the visitation probability (V_k) given by Equation (7), with $k = \{1, 2, main\}$. It must be noted that our accelerator is not modeled as another server in the system. However, its impact is modeled in the values attributed to the L2$ miss rate (mr_2), the visitation probability of the main memory (V_{main}), the cast-outs (CO) and the trailing-edge (TE) delay.

4.2 Multi-core with and without our Accelerator, M/D/1 Model

In a multi-core system like the targeted one (Figure 3), it is necessary to include in the previous analysis the impact of shared resources in the system, namely the bus. The utilization (U) of the bus depends of several situations:

- It depends on the number of misses generated by the CPUs and their L1$, which affects the amount of data transferred on the bus;
- It depends on the cache coherence protocol, i.e., a read miss in one L1$ that can hit in any other L1$ and a write that either forces the invalidation or the update of the data in the others L1$.

The use of a shared bus changes the previously described visitation probabilities (V) in Equation (7). The visitation probability (V) of the bus is 1, because all misses in any L1$ imply utilizing the bus. The probability of visiting the L2$ now not only depends on the miss rate of each L1$ ($OwnL1$), but also on the miss rate of other L1$ ($OtherL1$) that share the bus. Besides, the number of hits/misses of a particular cache line in the $OtherL1$ is also dependent on the application. [18] presents typical values for the hit rate in $OtherL1$, which are in the range of 10% to 50%. This parameter will be referred to as the hit rate in $OtherL1$ ($HRO1$). Therefore, the new visitation probability (V_k), with $k = \{1, 2, main\}$, for each CPU in the system is:

$$\begin{aligned} V_{bus} &= 1 \\ V_2 &= 1 - HRO1 - \frac{mr_2}{mr_1} \\ V_{main} &= \frac{mr_2 + FE \times mr_2}{mr_1}, \text{ where} \\ & FE = 0, \text{ if the accelerator is not used} \end{aligned} \tag{8}$$

To determine the service time (S_k), with $k = \{1, 2, main\}$, of the shared bus, we need to determine the amount of data transferred, the bus cycle time and the bus width. A bus that has separate lines for data and control has two different services times:

$$\begin{aligned} S_{control} &= bus_cycle_time \times \frac{control_request_size}{bus_control_width} \\ S_{data} &= bus_cycle_time \times \frac{L1_cacheline_size}{bus_data_width} \end{aligned} \tag{9}$$

Therefore, the total utilization (U_k), with $k = \{1, 2, main\}$, of the shared bus, both control (CB) and data (DB) busses, due to misses in any of the 3 L1$ ($OwnL1$), in the other L1$ ($OtherL1$) and due to the cast-outs (CO) is:

$$\begin{aligned} UOwnL1_{DB} &= V_{bus} \times S_{data} \times R \\ UOwnL1_{CB} &= V_{bus} \times S_{control} \times R \\ UOwnL1_{CO} &= CO \times V_{bus} \times S_{data} \times R \\ UOtherL1_{DB} &= 3 \times V_{bus} \times HRO1 \times S_{data} \times R \\ UOtherL1_{CB} &= 3 \times V_{bus} \times HRO1 \times S_{control} \times R \end{aligned} \tag{10}$$

And the corresponding fetching delay (T_k), with $k = \{1, 2, main\}$, for both control (CB) and data (DB) busses, due to misses in any of the 3 L1$ ($OwnL1$), in the other L1$ ($OtherL1$), and due to the cast-outs (CO) is:

$$\begin{aligned} TOwnL1_{DB} &= V_{bus} \times S_{data} \times (1 + \frac{0.5UOwnL1_{DB}}{1 - UOwnL1_{DB}}) \\ TOwnL1_{CB} &= V_{bus} \times S_{control} \times (1 + \frac{0.5UOwnL1_{CB}}{1 - UOwnL1_{CB}}) \\ TOwnL1_{CO} &= CO \times V_{bus} \times S_{data} \\ &\quad \times (1 + \frac{0.5UOwnL1_{CO}}{1 - UOwnL1_{CO}}) \\ TOtherL1_{DB} &= 0.5 \times V_{bus} \times HRO1 \\ &\quad \times S_{data} \times (1 + \frac{0.5UOtherL1_{DB}}{1 - UOtherL1_{DB}}) \\ TOtherL1_{CB} &= 0.5 \times V_{bus} \times HRO1 \\ &\quad \times S_{control} \times (1 + \frac{0.5UOtherL1_{CB}}{1 - UOtherL1_{CB}}) \end{aligned} \tag{11}$$

The queue sizes (Q_k), with $k = \{1, 2, main\}$, for both control (CB) and data (DB) busses, due to misses in any of the 3 L1$ ($OwnL1$), in the other L1$ ($OtherL1$),

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on April 20,2010 at 12:02:58 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

and due to the cast-outs (CO) are:

$$
\begin{aligned}
QOwnL1_{DB} &= \frac{UOwnL1_{DB} - 0.5UOwnL1_{DB}^2}{1 - UOwnL1_{DB}} \\
QOwnL1_{CB} &= \frac{UOwnL1_{CB} - 0.5UOwnL1_{CB}^2}{1 - UOwnL1_{CB}} \\
QOwnL1_{CO} &= \frac{UOwnL1_{CO} - 0.5UOwnL1_{CO}^2}{1 - UOwnL1_{CO}} \\
QOtherL1_{DB} &= \frac{UOtherL1_{DB} - 0.5UOtherL1_{DB}^2}{1 - UOtherL1_{DB}} \\
QOtherL1_{CB} &= \frac{UOtherL1_{CB} - 0.5UOtherL1_{CB}^2}{1 - UOtherL1_{CB}}
\end{aligned}
\tag{12}
$$

The trailing edge (TE_k), with $k = \{1, 2, main\}$, delay due to other L1$ ($OtherL1$) and L2$ ($L2$) is given by:

$$
\begin{aligned}
TOtherL1_{TE} &= (\frac{L1_cacheline_size}{bus_data_width} - 1) \\
&\times TE \times bus_cycle_time \\
TL2_{TE} &= (\frac{L2_cache_line_size}{bus_data_width} - 1) \\
&\times TE \times bus_cycle_time;
\end{aligned}
\tag{13}
$$

4.3 M/M/1 Model

Until now it is assumed a constant service time for each server (M/D/1). As this parameter models the time each resource of the system takes, it is interesting to study the impact of changing this value. Therefore, the following assume an service time (S) of each server to be exponential (M/M/1). Subsequently, the delay given by Equation (5) and Equation (11) became:

$$
\begin{aligned}
T_2 &= V_2 \times S_2 \times (1 + \frac{U_2}{1 - U_2}) \\
T_{main} &= V_{main} \times S_{main} \times (1 + \frac{U_{main}}{1 - U_{main}}) \\
TOwnL1_{DB} &= V_{bus} \times S_{data} \times (1 + \frac{UOwnL1_{DB}}{1 - UOwnL1_{DB}}) \\
TOwnL1_{CB} &= V_{bus} \times S_{control} \times (1 + \frac{UOwnL1_{CB}}{1 - UOwnL1_{CB}}) \\
TOwnL1_{CO} &= CO \times V_{bus} \times S_{data} \\
&\times (1 + \frac{UOwnL1_{CO}}{1 - UOwnL1_{CO}}) \\
TOtherL1_{DB} &= 0.5 \times V_{bus} \times HRO1 \\
&\times S_{data} \times (1 + \frac{UOtherL1_{DB}}{1 - UOtherL1_{DB}}) \\
TOtherL1_{CB} &= 0.5 \times V_{bus} \times HRO1 \\
&\times S_{control} \times (1 + \frac{UOtherL1_{CB}}{1 - UOtherL1_{CB}})
\end{aligned}
\tag{14}
$$

And the corresponding queue sizes:

$$
\begin{aligned}
QOwnL1_{DB} &= \frac{UOwnL1_{DB}}{1 - UOwnL1_{DB}} \\
QOwnL1_{CB} &= \frac{UOwnL1_{CB}}{1 - UOwnL1_{CB}} \\
QOwnL1_{CO} &= \frac{UOwnL1_{CO}}{1 - UOwnL1_{CO}} \\
QOtherL1_{DB} &= \frac{UOtherL1_{DB}}{1 - UOtherL1_{DB}} \\
QOtherL1_{CB} &= \frac{UOtherL1_{CB}}{1 - UOtherL1_{CB}}
\end{aligned}
\tag{15}
$$

Finally, the total delay (T_{total}) for each CPU in the multi-core system is given by the summation of Equation (5), Equation (11) and Equation (13) (with the utilization (U_k) given by Equation (10) and the visitation probability (V_k) given by Equation (8), with $k = \{1, 2, main\}$), for a constant service time. If a exponential service time is used the total delay (T_{total}) is calculated by the summation of Equation (14) and Equation (13) (with the utilization (U_k) given by Equation (10) and the visitation probability (V_k) given by Equation (8), with $k = \{1, 2, main\}$). The cycles per instruction (CPI) for each CPU in the multi-core system is then given by:

$$CPI[system] = CPI[\infty] + mr_1 \times T_{total} \tag{16}$$

Table 1 depicts a summary of the equations required to calculate the total delay (T_{total}) for each case.

	Single CPU	**Multi-core (M/D/1)**	**Multi-core (M/M/1)**
T_{total}	Eq. 5	Eqs. 5 + 11 + 13	Eqs. 13 + 14
V	Eq. 4	Eq. 8	Eq. 8
U	Eq. 7	Eq. 10	Eq. 10
Q	Eq. 6	Eq. 12	Eq. 15

TABLE 1
Analytical analysis summary

The impact of using our accelerator in both in a single CPU system or a multi-core system with a L1$ for each CPU, a unified L2$ and a main memory is studied in the Section 5, to evaluate the model. The values of the cast-outs (CO) and the trailing edge (TE) delay are given by [13] and [14] (based on real application measurements). Typical values for the hit rate in $OtherL1$ ($HRO1$) are presented in [18], and we also evaluate (in Section 5) its impact. The factor of eviction (FE) is related with our accelerator and, because there are no measurements for it, a wide enough range of values are evaluated to determine the impact of utilizing our accelerator. Moreover, the L2$ and the main memory service times (S), the cache line sizes and the miss rates of both caches (mr_1 and mr_2) utilized in this analytical analysis are the same as the ones used in Simics simulator, when executing the receiver-side of TCP/IP stack (in Section 5).

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on April 20,2010 at 12:02:58 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

5 RESULTS

As introduced in previous sections, the results gathered by utilizing our accelerator in a full-system simulator are inputs to the analytical analysis presented in Section 4. Therefore, we start by presenting the results of our accelerator when running synthetic benchmarks and the receiver-side of the TCP/IP stack. Afterwards, we present the results for the multi-core using the values gathered previously.

5.1 Full-system Simulation Results

In order to gather the necessary numbers we performed a complete system analysis of the previously described cache-based memory copy accelerator, using Simics full-system simulator [4]. The platform modeled contains a data and instruction L1$, a unified L2$ and a DRAM DDR 2 main memory. Moreover, we created a model of our accelerator to be incorporated in the simulator with the delays derived from the actual hardware implementation (presented in [1], [2] and [3]). The simulation parameters are described in Table 2.

CPU	
Type	Pentium 4
Frequency	2GHz
CPI	1
Operating System	Linux 2.4
Caches	
L1 I/D Caches	32kB, 64B cache line, direct-mapped, write-through
L1$ Hit Time	2 clk
L2 Unified Cache	2MB, 128B cache line, 8-way, write-back
L2$ Hit Time	15 clk
Accelerator	
Fill Index Table Time	2 clk
Index Table Read Hit Time	2 clk + L1$ read hit time
Write to SRC or DST addr	2 clk
Write-back evicted data	L2$ access time
Memory	
Type	DRAM DDR2 400 MHz
Avg. Access time	240 clk

TABLE 2
Simulation Parameters

Finally, we implemented an instruction-set architecture extension by a single instruction that is used to substitute the software function calls. This instruction, besides communicating to the hardware to provide the necessary parameters (`src`, `dst` and `size`), performs the necessary checks of the boundary conditions. In our case, the boundary conditions are checked using a simple `if` statement to determine if: 1) the size of the copy is smaller or bigger than the hardware can support (if the `size` of the copy is outside of the sizes supported by the hardware, then the software implementation is utilized); and 2) the `src` and `dst` addresses are aligned (and if they are not, then perform the necessary alignment in

STREAM Memory Benchmark			
Total memory required = 16.0 MB. Number of runs = 10			
1st level cache Working on Arrays of 800 B.			
	Function	Rate (MB/s)	Avg. Time (ns)
1	copy 32	212.31	14.25
2	copy 64	294.38	13.43
3	copy 32x2	140.58	19.33
4	copy 32x4	150.15	30.89
5	glibc memcpy	600.37	2.66
6	our Acc.	722.34	2.21

TABLE 3
Stream benchmark results

STREAM Benchmark	
# of Copies	100,000
# of Read Hits in the Indexing Table	18
# of Writes to SRC addr.	180
# of Writes to DST addr.	44

TABLE 4
Memory copy statistics for the Stream Benchmark

software before calling the hardware; if a cache line alignment is not possible then the software implementation is utilized). The overhead imposed by this new instruction is simply the two new comparisons. There are no architectural changes needed to the processor. We substituted all memory copies function calls by our instruction in all the benchmarks used to verify our solution.

5.1.1 *STREAM synthetic benchmark*

In order to determine the raw throughput that our solution can provide, we used the STREAM [19] benchmark to compare the benefits of our accelerator with several copy kernels: `copy_32`, `copy_64`, `copy_32x2`, `copy_32x4`, and `glibc memcpy`. The first four kernels perform copies on loops of either integers (32 bits) or doubles (64 bits), using two or four intermediary variables. The `glibc memcpy` uses the standard `glibc` algorithm implemented in the Linux kernel, which is optimized in assembly.

The STREAM benchmark provides the average copy throughput measured over 1 000 000 executions, the process is repeated 10 times and the best time is displayed. It provides the throughput for copies of arrays of 800B (to test the L1$ throughput).

The STREAM benchmark does not use the copied data and executes several copies in order to average the results. As such, the accelerator has, for each repetition, to write-back the previously performed copy, fetch the necessary data from the main memory into the cache and fill the indexing table, which is the worst-case scenario for our accelerator. Table 3 presents the results and Table 4 presents the statistics of the hardware.

The speedup in terms of both throughput and average execution time (for the `glibc memcpy` and our accelerator) for the STREAM benchmark is 1.2 (as it can be

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on April 20,2010 at 12:02:58 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

calculated by dividing row 5 by row 6 of Table 3). As expected the number of writes to either the `src` or the `dst` addresses are small due to the write-back of the copy previously performed before each new iteration, as presented in Table 4.

5.1.2 Receiver-side of the TCP/IP stack

As the STREAM benchmark is a synthetic benchmark, we evaluated our solution with a more realistic workload, such as the receiver-side of the TCP/IP stack that is also bounded by memory copies. Consider packets arriving at very high speeds at a CPU and going through the TCP/IP protocol. The data is subsequently copied from the network interface card to the system's main memory (typically using a DMA), processed, and copied again from the operating system (OS) to the user application.

Therefore, we used the user-space implementation of this part of the stack to evaluate our accelerator. We captured the packets generated by sending a file of 10MB over a TCP connection. The file was divided into 8,418 packets of which 92.8% have 1,260 Bytes. We fed the packets captured into the receiver-side of the TCP/IP stack in user-space, measured the average number of cycles per packet and repeated the process 10 times.

The software version (using the function `glibc memcpy`) took on average 54,972 cycles/packet while with our accelerator it was reduced to 11,930 cycles/packet (Table 5 presents the results). It is worth mentioning that these numbers are given by the application itself and not measured in the simulator (the application returns an approximation due to interference of the OS). In order to have more accurate results we measured, with the simulator, the execution time and number of instructions executed. Table 5 presents the statistics gathered by the application, the simulator and by the accelerator itself.

From Table 5, we can calculate a speedup of 4.61 (corresponding to a 78% reduction) in execution time, when our accelerator is used. Besides, we reduced the number of instructions executed by 44% (because we do not use loads and stores) and we have a higher cache hit rate compared with the software version (because we fetch the `src` data from the main memory to the cache while executing the memory copy).

It has to be noted that the receiver-side of the TCP/IP stack does not include the reading of the copied data by the application that is receiving this data (as it can be noticed by the zero read hits in the indexing table in Table 5). In order to correctly evaluate our proposal, we included a read of every packet received in order to mimic the normal behavior of an application. Table 5 also presents the statistics for this case. The simulator provided a speedup of 2.96 (corresponding to a 66% reduction) in execution time. The reason for this decrease is the increase in execution time of the stack, as seen by the number of instructions executed (in this case, our hardware only reduces the number of instructions compared with the software case by 26%). As such, the clear benefits of our solution are diluted over the total execution time, however we still achieve a higher cache hit rate than the software.

Standard TCP/IP stack		
App. Stats.	SW Exec. Time	54,972 cycles/pkt
	HW Execution Time	11,930 cycles/pkt
Sim. Stats.	SW # Instr.	25,256,762
	SW Exec. Time	232 msec
	SW L1$ Read Hit	92.12%
	SW L1$ Write Hit	89.53%
	HW # Instr.	13,968,064
	HW Exec. Time	51 msec
	HW L1$ Read Hit	98.05%
	HW L1$ Write Hit	96.02%
HW Stats.	# Copies	8,414
	# Read Hits Index.	0
	# Writes SRC Addr.	262
	# Writes DST Add.	481,685
TCP/IP stack with reads		
App. Stats.	SW Exec. Time	98,693 cycles/pkt
	HW Exec. Time	57,097 cycles/pkt
Sim. Stats	SW # Instr.	534,685,789
	SW Exec. Time	268 msec
	SW L1$ Read Hit	92.77%
	SW L1$ Write Hit	90.24%
	HW # Instr.	180,120,983
	HW Exec. Time	90 msec
	HW L1$ Read Hit	97.82%
	HW L1$ Write Hit	96.18%
HW Stats	# Copies	8,414
	# Read Hits Index.	4,299
	# Writes SRC addr.	22,686
	# Writes DST addr.	1,194,569

TABLE 5
Memory copy statistics for the receiver-side of the TCP/IP stack

In order to evaluate the impact of changing the number of packets processed, we repeated the experiment with a smaller file. The speedup provided by our accelerator (measured by the simulator) was 5.68 times, where the hardware kept the same average cycles per packet (the software execution time increased). This means that our accelerator keeps the same performance independent of the number of packets processed. In contrast, the software version behaves better for a bigger number of packets processed. Analyzing the code, we can find that, per session, there is a loop to process the packets of that session. Therefore, the bigger the number of packets per session, the better the software version behaves. However, in a typical web-browsing or e-mail sessions (in contrast with FTP session), the number of packets processed per session is limited.

Finally, we compare our accelerator with the solution presented by [20]. In that paper, the authors modified the standard memory controller in order to support memory copies and evaluated their solution with the receiver-side of the TCP/IP stack. The authors used SimpleScalar [21] simulator that does not support multiprocessor platforms, and in order to simulate a 4-way platform, the authors claim that they assigned the correct timing to each model as it would have in a 4 core system.

In order to be able to compare our solution with theirs, we used the same delays for our simulation with Simics.

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on April 20,2010 at 12:02:58 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

We utilized the same memory hierarchy and the same receiver-side TCP/IP stack implementation. The main differences in the works are:

- Type of configuration: the authors of [20] present results for both synchronous and asynchronous configurations of their copy engine. The asynchronous version provide computation overlap, allowing the copy instruction to retire even before the copy transaction is complete, and therefore, allowing for the higher speedups. The synchronous version can include or not load bypass, i.e., allowing the cache misses generated by the CPU to be interleaved with the copy engine requests. For the synchronous version, the highest speedup is obtained when load bypass is used, because the CPU does not have to stall until all outstanding copies are completed. As our approach is to hold the processor while the indexing table is being filled, we would expect to be a worst solution than the ones presented in [20]. However, because the time needed to perform the copy in our case (i.e., filling the indexing table) is negligible, this is not the case.
- The payload size of the packets being processed: the authors of [20] analyzed the impact of their solution for 100,000 packets of payloads of 512, 1,024 and 1,400 bytes. The packet sizes studied in our case is the 1,024 bytes payload and the traces have 8,418 packets. The different sizes would not have impact in our results as soon as the payload is bigger than a cache line size.

Taking into account the mentioned differences, we present a 78% reduction in execution time while the authors of [20] present 52.5% for the same case (i.e., synchronous copy without load bypass and payload size of 1,024 bytes). The highest speedup achieved in the work of [20] is 72,8% for an asynchronous copy, which is still below the numbers we report. Our benefits are mainly due to performing the copy faster (as it is done initially in the cache), to better utilizing the bus (when we actually perform the write-back) and obtaining a higher cache hit rate.

5.2 Analytical Analysis Results

The analytical analysis introduced in Section 4 is utilized to model the impact of applying our accelerator in a multi-core system. The finite cache penalty of a baseline scenario without the usage of our accelerator is compared with the finite cache penalty of a system utilizing our accelerator. These comparisons are performed for both a single CPU case (to validate the model) as well as the multi-core system assuming the two different service time of the servers in the system.

The values utilized to model the multi-core system are presented in Table 6. The baseline scenario depicted in the second column corresponds to the typical values used in real systems. In particular, the L2$ and the memory service times and the cache lines sizes are the same as the ones used for the Simics simulator; the hit rate in $Other L1$ ($HRO1$), the cast-outs (CO) and trailing edge (TE) delay are given by [13] and [14] (based on real application measurements) and the cache miss rates (mr_1 and mr_2) were observed when our accelerator was implemented in Simics simulator. The other parameters are typical values. The factor of eviction (FE) is related with our accelerator and, because there is no possibility of performing measurements to determine its value, a wide enough range of values is evaluated to determine the impact of utilizing our accelerator. The third column presents the parameters' range used to evaluate the system, as these are the parameters that either depend on the application (the hit rate in $Other L1$ - $HRO1$), or on the use of our accelerator (miss rate in L2$ - mr_2 and factor of eviction - FE), as introduced in Section 4. It is clear from Table 5, that the factor of eviction (FE) decreases from the case where the receiver-side of the TCP/IP stack does not perform reads to the case where it does[3]. This is due to the increase of the number of instructions executed from one case to another. For the case of the receiver-side of TCP/IP stack, the factor of eviction (FE) ranges from 3.45% (no reads) to 0.6% (with reads). As the value of FE has high influence on the final result, we made the FE value range from 0% to 40% in order to correctly evaluate the impact.

First, the single CPU case is analyzed in order to evaluate the model. Figure 5 depicts the impact of increasing the miss rate in L2$ (mr_2) and the factor of eviction (FE) values on the finite cache penalty (FCP), compared with the baseline scenario described in Table 6. In this case, the hit rate in $Other L1$ ($HRO1$) is zero, as there are no other CPUs in the system.

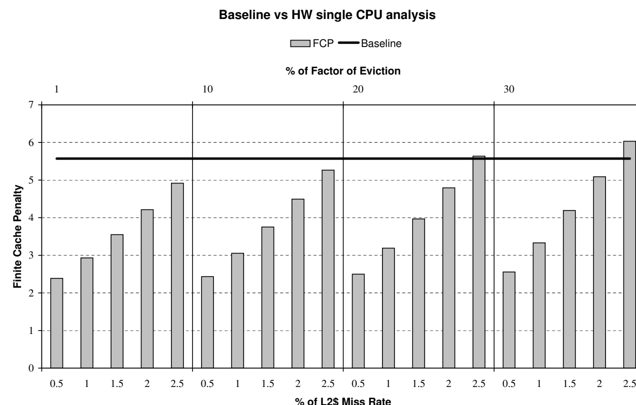


Fig. 5. Single CPU analysis

It is clear from Figure 5 that our accelerator can provide benefits for a wide range of miss rate in L2$ (mr_2) and factor of eviction (FE) values and that, in the best case, those benefits can reach a decrease of finite cache penalty (FCP) of 57.1% (the best FCP is when

3. The FE can be calculated from Table 5 by summing the number of writes to both the `src` and `dst` addresses and dividing the result by the number of instructions executed.

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on April 20,2010 at 12:02:58 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

Parameter	Default Value	Range
L2$ Service Time (S_2)	15 clk (used in Simics)	-
Memory Service Time (S_{main})	240 clk (used in Simics)	-
L1$ Miss Rate (mr_1)	10% (returned by Simics)	-
L2$ Miss Rate (mr_2)	3% (returned by Simics)	0.5% to 3%
Hit Rate of $OtherL1$ ($HRO1$)	20% (given by [13] and [14])	10% to 50%
Factor of Eviction (FE)	0% (wide range)	1% to 40%
Cast-Outs (CO)	20% (given by [13] and [14])	-
Trailing Edge (TE) delay	10% (given by [13] and [14])	-
$L1_cacheline_size$	32 bytes (typical value)	-
$L2_cacheline_size$	128 bytes (typical value)	-
bus_data_width	8 bytes (typical value)	-
$control_req_size$	1 bytes (typical value)	-
$bus_control_width$	1 bytes (typical value)	-
bus_cycle_time	1 clk (typical value)	-

TABLE 6
Parameters defining the system modelled

the value of mr_2 is 0.5% and the value of FE is 1%). However, if the FE and the mr_2 values are too high, then the utilization of our accelerator can actually decrease the performance. On the other hand, this is not a realistic scenario once the presence of our accelerator will always decrease the value of mr_2, as can be seen from Table 5. Moreover, as presented in Section 5.1, the value of FE for the receiver-side of the TCP/IP stack (for both with and without reads) ranges from 0.6% to 3.45%.

In addition, the previous section mentioned the possible influence of the cost-outs (CO) value and of the trailing edge (TE) delay. Therefore, the value of CO and the TE delay was increased by 20%, which increased the finite cache penalty (FCP) by 1.2% for the first case and 4.5% for the second. Subsequently, these parameters have a small impact in the system.

As mentioned before, one of the parameters of the multi-core system depends on is the application, modeled through the hit rate in $OtherL1$ ($HRO1$) value. Therefore, the impact of changing this parameter's values is studied and the benefit of one of the CPUs in the system (as the model calculates the FCP of one of the CPUs taking into account the presence of the others) is analyzed. Figure 6 depicts the impact of changing the $HRO1$ values, compared with the baseline scenario. As expected, the increase in $HRO1$ value reduces the FCP of the system compared with the baseline scenario.

As demonstrated for the single CPU, the usage of our accelerator is modeled mainly through the miss rate in L2$ (mr_2) and the factor of eviction (FE) values.

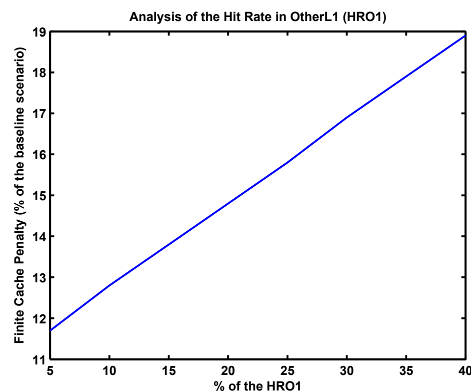


Fig. 6. Hit Rate in $OtherL1$ ($HRO1$) analysis

Therefore, our accelerator for a multi-core system is evaluated by analyzing the impact on the finite cache penalty (FCP) that the accelerator has for different parameter's values. Figure 7 depicts the percentage of FCP decrease compared with the baseline scenario for a constant service time (M/D/1). The analysis increases the mr_2 and the FE values in a multi-core system, keeping the remainder of the parameters with the default values described in the Table 6.

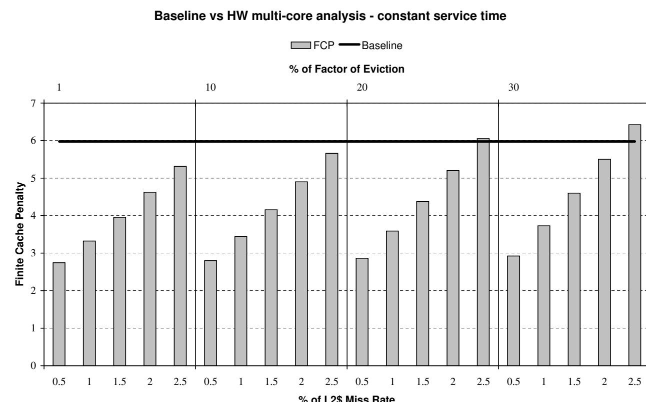


Fig. 7. Multi-core system analysis with constant service time

From the previous analysis it is possible to reach a FCP decrease of 54.1% for one of the CPUs for the best case (the best FCP is when the value of mr_2 is 0.5% and the value of FE is 1%).

Figure 8 depicts the same analysis, now assuming a exponential service time (M/M/1) for each server. For this case it is possible to reach a finite cache penalty (FCP) decrease of 54.6% for one of the CPUs for the best case (the best FCP is when the value of mr_2 is 0.5% and the value of FE is 1%). Moreover, an exponential service time imposes increase on the baseline FCP by 9.2%, compared with the constant service time. However, the different service times in the servers have a smaller impact on the usage of our accelerator, approximately 7.9%. Moreover, the impact on using an exponential service time for our accelerator is bigger for higher miss

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on April 20,2010 at 12:02:58 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

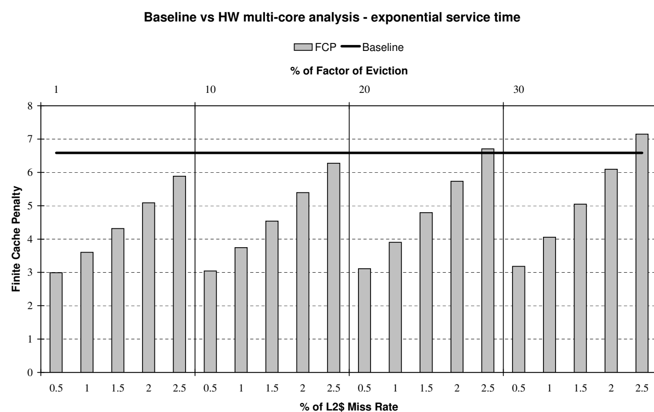


Fig. 8. Multi-core system analysis with exponential service time

rate in L2$ (mr_2).

As mentioned previously, the other parameters that have impact in the system are the cast-outs (CO) value and the trailing edge (TE) delay. However, as demonstrated for the single CPU case, these have small impact on the system.

The previous results are for a 3 CPU system. If the system is scaled to a larger system, there will be an increase in the finite cache penalty (FCP) (with or without our accelerator), due to the bus bottleneck. Therefore, the benefits of our accelerator depend on the performance of the system without it, as the architecture of our accelerator does not depend on the number of CPUs connected to the system, but on the cache it is connected to. As such, our accelerator can be used in any system but its benefits are depending on the performance of the system without it.

6 Conclusions

In this paper, we presented the cache-based memory copy hardware accelerator in a multi-core system supporting message passing communication model and we utilized an analytical analysis based in open-queuing theory to study the system. The accelerator (that was previously presented) is able to accelerate data-intensive algorithms with no data-parallelism, such memory copy algorithms. Our solution performs memory copies faster than the traditional manner (utilizing loads and stores) and avoids cache pollution and duplicating data in the cache (as the copy is simply a pointer to the original data). The access to the copied data is also faster, because the pointer allows the copied data to be accessed from the cache. Furthermore, we delay the actual data movement until its impact in the system is minimized.

In order to correctly model our solution in a multi-core system supporting message passing communication model, we gathered the necessary information by utilizing a full-system simulator. We demonstrate the advantages of our solution utilizing several applications: the STREAM benchmark and the receiver-side of the TCP/IP stack. Our accelerator reaches speedups from 2.96 to 4.61 for the receiver-side of the TCP/IP stack, reduces the number of instructions from 26% to 44% and achieves higher cache hit rate.

In addition, we presented the analytical analysis based on open-queuing theory. We validated the analysis by first gathering the results for a single CPU case and afterwards for the multi-core system. Our cache-based hardware accelerator is able to achieve a reduction on the average number of cycles executed per instruction up to 50% for one of the CPUs in the system, when executing the message passing protocol.

References

[1] S. Vassiliadis, F. Duarte, and S. Wong, "A Load/Store Unit for a *memcpy* Hardware Accelerator," in *Proceedings of the International Conference on Field Programmable Logic and Applications*, 2007, pp. 537–541.

[2] S. Wong, F. Duarte, and S. Vassiliadis, "A Hardware Cache *memcpy* Accelerator," in *Proceedings of the IEEE International Conference on Field-Programmable Technology*, 2006, pp. 141–148.

[3] F. Duarte and S. Wong, "A *memcpy* Hardware Accelerator Solution for Non Cache-line Aligned Copies," in *Proceedings of the International Conference on Application-specific Systems, Architectures and Processors*, 2007, pp. 397–402.

[4] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hllberg, J. Hgberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[5] H. Shan and J. P. Singh, "A Comparison of MPI, SHMEM and Cache-Coherence Shared Address Space Programming Models on Tightly-Coupled Multiprocessors," *International Journal of Parallel Programming*, vol. 29, no. 3, pp. 283–318, May 2001.

[6] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis, "Comparing Memory Systems for Chip Multiprocessors," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 358–368, May 2007.

[7] 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 *Proceedings of the 12th International Conference on Supercomputing*, 1998, pp. 243–250.

[8] "Cray T3D System Architecture," Cray Research, Inc.

[9] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, D. N. J. Chapin, M. H. J. Baxter, A. Gupta, M. Rosenblum, and J. Hennessy, "The Stanford FLASH Multiprocessor," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994, pp. 302–313.

[10] J. Heinlein, K. Gharachorloo, R. Bosch, M. Rosenblum, and A. Gupta, "Coherent Block Data Transfer in the FLASH Multiprocessor," in *Proceedings of the 11th International Symposium on Parallel Processing*, 1997, pp. 18–27.

[11] S. C. Woo, J. P. Singh, and J. L. Hennessy, "The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors," in *Proceedings of the 6th International Conference on Architectural support for Programming Languages and Operating Systems*, 1994, pp. 219–229.

[12] D. Buntinas, G. Mercier, and W. Gropp, "Data Transfers between Processes in an SMP System: Performance Study and Application to MPI," in *Proceedings of the 2006 International Conference on Parallel Processing*, 2006, pp. 487–496.

[13] R. E. Matick, "Comparison of Analytic Performance Models using Closed Mean-Value Analysis versus Open-Queuing Theory for Estimating Cycles per Instruction of Memory Hierarchies," *IBM Journal of Research and Development*, vol. 47, no. 4, pp. 495–517, Jul. 2003.

[14] R. E. Matick, T. J. Heller, and M. Ignatowski, "Analytical Analysis of Finite Cache Penalty and Cycles per Instruction of a Multiprocessor Memory Hierarchy using Miss Rates and Queuing Theory," *IBM Journal of Research and Development*, vol. 45, no. 6, pp. 819–842, Nov. 2001.

[15] L. Kleinrock, *Queueing Systems, Vol. I: Theory, Vol. II: Computer Applications*. John Wiley and Sons, Inc., 1975.

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on April 20,2010 at 12:02:58 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

[16] S. Lavenberg, *Computer Performance Modeling Handbook*. Academic Press, Inc., 1983.
[17] D. Gross and C. M. Harris, *Fundamentals of Queueing Theory*. John Wiley and Sons, Inc., 1998.
[18] T. Lovett and R. Clapp, "STiNG: A CC-NUMA Computer System for the Comercial Marketplace," *ACM SIGARCH Computer Architecture News*, vol. 24, no. 2, pp. 308–317, May 1996.
[19] J. D. McCalpin, "A Survey of Memory Bandwidth and Machine Balance in Current High Performance Computers," in *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 1995, pp. 19–25.
[20] L. Zhao, L. Bhuyan, R. Iyer, S. Makineni, and D. Newell, "Hardware Support for Accelerating Data Movement in Server Platform," *IEEE Transactions on Computers*, vol. 56, no. 6, pp. 740–753, Jun. 2007.
[21] "SimpleScalar," http://www.simplescalar.com/.

Filipa Duarte received her PhD in Computer Engineering from the Electrical Engineering, Mathematics and Computer Science faculty of the Delft University of Technology, The Netherlands, in December 2008. She is currently a researcher at the Ultra Low Power DSP group at the Holst Centre/IMEC, The Netherlands. Her research interests include ultra low power digital design, in particular memory and IO subsystem of uni- and multiprocessors, both from performance and low power perspectives.

Stephan Wong received his PhD in Computer Engineering from the Electrical Engineering, Mathematics and Computer Science faculty of Delft University of Technology, The Netherlands, in December 2002. He is currently working as an assistant professor at the Computer Engineering Laboratory at Delft University of Technology, The Netherlands. He has considerable experience in the design of embedded reconfigurable media processors. He has worked also on microcoded FPGA complex instruction engines and the modeling of parallel processor communication networks. His research interests include embedded systems, multimedia processors, complex instruction set architectures, reconfigurable and parallel processing, microcoded machines, and distributed/grid processing. He is a member of the IEEE and ACM.

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on April 20,2010 at 12:02:58 UTC from IEEE Xplore. Restrictions apply.