Cache activity profiling tool for the LEON4 processor

Maria Ntogramatzi[1,3], Panagiotis Katsaros[2,3] and Spyridon Nikolaidis[1,3],

[1]Department of Physics [2]Department of Informatics
Aristotle University of Thessaloniki
Thessaloniki, Greece
mntogr@physics.auth.gr

[3]Information Technologies Institute
Centre of Research and Technology Hellas
Thessaloniki, Greece

***Abstract*—The performance of modern systems depends significantly on the cache activity. Hence, tools that monitor the cache performance are very useful for optimization of the software or even, whenever this is possible, the hardware, of a system. In this paper, a tool that provides statistics about the cache activity of the LEON4-N2X embedded system is discussed. The tool analyzes the memory access trace of a program executed bare metal and calculates, with the use of the reuse distance, the latency added by the cache activity. Furthermore, the tool measures all the misses occurred and classifies them per their cause.**

Keywords—LEON4; reuse distance; cache memories; memory access trace;

I. INTRODUCTION

For mitigating the well-known memory wall problem, modern multicore systems implement multi-level memory hierarchies. On the upper level of the hierarchy, the closest to the processor, there are the cache memories. Commonly, there are more than one levels of cache, usually up to 4 for desktop processors, and up to 2 for embedded systems. The first level is most often, private for each core, while the second is shared. Although, caches significantly improve a system's performance, a phenomenon called cache interference can downgrade that improvement.

When referring to a private cache, the problem of interference emerges mainly when different data are mapped to the same cache location [1]. For multicore systems and shared caches, the very nature of sharing constitutes another source of interference. In order to keep data consistency between all levels of memory, multiprocessor sharing can cause an extra number of misses. To elaborate, let us consider data located in the shared cache that are used by two or more processor cores. If one of them changes this entry written in the shared cache, all other copies of this particular memory block must be invalidated. That means that the copy in each processor's private cache should be invalidated and the next reference to it will be a miss. Many schemes have been proposed to cope with the multiprocessor cache interference. One of them implemented in this work, is way – based cache partitioning. The main idea is that the cache is partitioned in such a way that each processor can use exclusively a subset of the cache's ways. In this way, the interference is reduced, since there is no longer any part of the cache shared. The performance though can be degraded. Evidently, cache performance degradation results in system performance decrease.

Consequently, a system's performance depends significantly on the cache activity. Since the cache sub-system of a given system is almost always predetermined, the overall performance depends on how cache friendly regarding the specific cache system, is the application [2]. Hence, tools that evaluate the cache friendliness of applications, i.e. tools that measure the overall number of cache misses and also identify the cause of each miss [3], are really useful for optimization of either the software or, whenever it is possibly, the hardware.

Over the years, many profiling tools have been developed, e.g. the DProf presented in [3] which classifies the misses per their cause, or the commonly used Cachegrind [4] and Gprof [5]. All these tools work using the Linux kernel, as opposed to our tool, which profiles applications running bare metal on an embedded processor. The discussed tool uses the reuse distance (i.e. LRU stack distance) [6] which is an accurate and machine independent metric of locality [3], to evaluate the cache friendliness of an application. It measures the number of cache misses occurred during the execution of the application as well as it classifies the misses as either Compulsory, Capacity or Conflict (3C's). Moreover, the tool calculates the overall latency added by the cache activity. Though it was developed for performing cache interference analysis targeting a specific device the LEON4-N2X, the reuse distance makes it suitable for use with other devices sharing the same cache sub-system.

The reuse distance has been used as a metric for data locality since 1970s. Researchers have used it for many purposes e.g. for miss rate prediction [2] and cache performance prediction [7]. Although its applicability on multiprocessor systems has been questioned [8], the reuse distance approach is suitable for this work, as the shared cache has been partitioned.

The rest of this paper starts with a brief description of some basic concepts and the target device, in section II. In section III, the proposed tool is analyzed. The results are presented in section IV and the paper concludes with section V.

This work was sponsored by European Space Agency under ESA contract No. 4000111814/14/NL/MH.

II. BACKGROUND AND TARGET DEVICE

A. Theoretical Background

For an overall presentation and a better understanding of our work, a few basic concepts are briefly introduced below:

The **memory access trace** is a sequence of all the memory accesses performed during the execution of a program.

The **reuse distance** of an access to a memory block is the number of the distinct accesses between the current and the previous access to the specific block. The reuse distance of a **first** access to any block equals to ∞**.**

Cache miss types 3 C's

As Beyls and D'Hollander state in [9], the cause of a cache miss can be used to categorize it according to the 3C's model [10] into one of these three categories: compulsory, capacity and conflict misses.

- **Compulsory** or **cold** misses occur on the first access to a block. Since the specific data have not been previously requested, it is highly unlikely for them to already be in the cache so a miss is almost inevitable.
- **Capacity** misses happen due to cache's limited capacity. With the cache full, a request for new data, data that has not already been in the cache, will result in a cache miss: a resident of the cache will be evicted for the new data to come in.
- **Conflict** misses occur in caches implementing set – associative or direct mapped placement policies. When several blocks are mapped to the same set a miss can happen even if there are empty slots in the cache. They are misses that would not happen in a full associative cache.

B. The Target Device

The discussed tool was built targeting the LEON4-N2X device [11]. In this device, a four core LEON4 CPU is implemented, with a cache sub-system and two trace buffers.

The LEON4 is a 32-bit processor core conforming to the IEEE-1754 (SPARC V8) architecture [12]; Its cache sub-system implements two levels of cache. The first level is private for each core, while the second level is shared. Moreover, the level 1 (L1) cache is implemented as Harvard architecture, while the level 2 (L2) cache is shared for both instructions and data. All three caches are implemented as multi – way caches with associativity of four. The way size for L1 caches is 4 KiB, while for the L2 cache it is 64 KiB. The cache lines are 32 bytes. The replacement policy for L1 caches is the Least Recently Used (LRU) policy. The write policy for the data cache is write – through with no allocation on the write miss. For the L2 cache's replacement policy there are three options: LRU, pseudo – random and master – index (where the way to replace is determined by the master index). All the experiments were conducted with the master – index policy; it was used to implement way – based cache partitioning, a scheme proposed for coping with the multiprocessor cache interference. For the write policy, the cache can operate either as write-through or as copy-back (write-back). The default policy is copy-back.

The two trace buffers: the instruction trace buffer which stores the executed instructions and the Advanced High-performance Bus (AHB) trace buffer which stores the AHB data transfers, were used to compose the memory access trace, used as an input to the discussed tool.

III. CACHE ACTIVITY PROFILING TOOL

The proposed tool was designed to process the memory access trace of a program already executed by the target device and ultimately calculate the latency added by the cache activity. The total latency is calculated by summing up the latency added by each cache miss. According to the type of a miss, i.e. if it is a read or write, a L1 or L2 miss, a different amount of time is added. Consequently, for a precise estimation of the overhead added by the cache activity, for each miss its type must be recognized and its latency must be calculated according to the user manual of the target device [11]. On top of that, the latency added by a L2 access depends on the type of the previous access to the L2 cache. Additionally, the cache friendliness of the application is estimated via calculating the number of the occurred misses and categorizing them into the 3C's. The criterion for all the categorizations was the reuse distance of the access.

The algorithm was implemented in C. The input of the tool is a text file containing the memory access trace to be processed. The output can either be printed on the screen or in a file and states the latency added by all misses of both cache levels, the number of L1 instruction cache (icache) misses, the number of L1 data cache (dcache) misses, the number of L2 misses and how many of them are Compulsory, Conflict or Capacity misses.

A. Reuse Distance Approach for Defining Hit and Miss Probabilities

As mentioned by Reineke in [13], the hit and miss probabilities of a memory access can be given based on its reuse distance for both randomized and deterministic caches. The cache sub-system of the LEON4-N2X device is deterministic.

In deterministic caches the reuse distance can be used to predict, in a definite way, if an access will hit or miss the cache. That is, if the reuse distance is greater than the cache capacity, the access will end up to a miss. If it is lesser or equal, there will be a hit.

We used the reuse distance to characterize each access as a miss or a hit. Furthermore, the reuse distance was used for classifying the misses into one of the 3 C's categories.

According to [9], in the case of a fully associative cache a stack should be created in order to measure the stack distance for each access. The memory blocks accessed, get on the top of the stack. Every time a new block arrives, the stack is scanned and if the block is found somewhere deeper in the stack, its distance from the top is measured and the entry is relocated to the top of the stack. The distance measured is the stack distance of that memory block. If the block is not found anywhere in the

stack, then the stack distance is ∞. In the case of a set associative or a direct mapped cache a stack must be created for each set; and all the above apply for every set.

All three caches are set associative. The misses are classified into one of the 3 C's per their stack distance as described in the following bullets:

- If the stack distance is greater than the number of ways - 4 for L1 caches and 1 for L2 cache (master-index policy) - and is less than the number of sets - 128 for L1 caches and 2048 for L2 cache - then a **conflict miss** has occurred.
- If the stack distance is greater than the number of sets - 128 for L1 caches and 2048 for L2 cache - then a **capacity miss** has occurred.
- If the stack distance is ∞ then we have a **cold miss.**
- If the stack distance is less than 4 for L1 caches or equal to 0 for L2 cache the block has **hit** the cache.

a) Notes

During our tests, it was noticed that some memory blocks needed more than one calls to become cache residents. Each call produced an additional cache miss. Although, these additional misses were included in the overall number of misses they were not included in any of the above three categories. Thus, overall number of misses may be different from the sum of the misses belonging in the above mentioned three categories.

B. Special Considerations Regarding Data Cache

The trace is a list of mixed AHB and instruction trace buffer entries, sorted after time. The first value in each entry is either the literal string AHB or INST indicating the type of entry. The execution of an instruction is signified by the literal string INST. The literal string AHB indicates an AHB access (i.e. an access through the AHB processor bus). An access to the instruction cache can easily be recognized as the execution of an instruction is signified by the literal string INST. However, no information is given about accesses to the data cache. The accesses in data cache cannot be identified directly, only indirectly. A data access occurs when a load or a store instruction is executed. During "load", data are transferred from the memory to the registers and during "store", data are transferred by the registers to the memory. Thus, to locate all data transfers, we had to locate all store and load instructions. That was achieved by decoding the opcode for every instruction and comparing it with the opcodes of the SPARC instructions "load" and "store" [12]. A match signified a data access. However, this process does not provide any information about the identity of the data (i.e. memory address the data are written to/will be written to). As the address of the requested data could not be obtained, the reuse distance approach could not be applied. Consequently, the classification of the data misses into one of the 3 C's was not possible.

C. Calculation of Latency

The steps followed for calculating the latency added by the cache activity are (Fig. 1.):

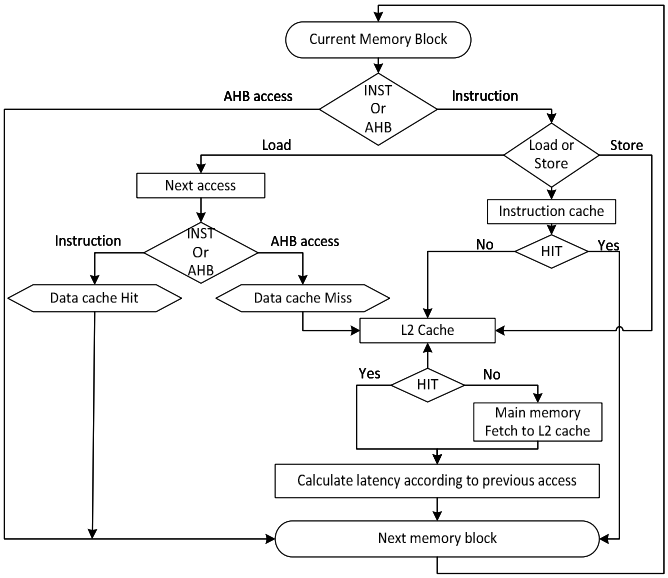

Fig. 1 . Flowchart of the algorithm calculating the latency added by the cache activity.

0) The entries are read and registered as either INST or AHB.
1) The next (or first) entry is processed:
   a) If the entry is an INST, the opcode is checked.
      i) If it is a "load", the next entry is checked (step1).
         (1) If it is AHB, a data miss has occurred.
         (2) If it is INST, the data hit the dcache.
      ii) If it is not a "load" we proceed to step 2.
   b) If the entry is a AHB access, we return to step 0.
2) The set of the appropriate cache the memory block will be mapped to is determined.
3) The memory block's tag is placed in a stack which is scanned to determine if the block is already a resident.
4) The stack distance is measured according to the memory block's place in the stack or its absence from the stack.
5) The access is classified as either miss or hit.
   a) If the access was a hit, we return to step 1.
   b) If the access was a miss the kind of the miss is determined and the steps 2, 3, 4, and 5 are followed for the L2 cache.
6) The latency added by the L2 interference is calculated according to the current and the previous access type.
7) The total latency added by the access is calculated by summing the extra time added by each memory level.

IV. RESULTS

The tool has been tested by processing the memory access traces of a number of test programs. For all our test cases the number of misses, for all three caches, calculated by our tool

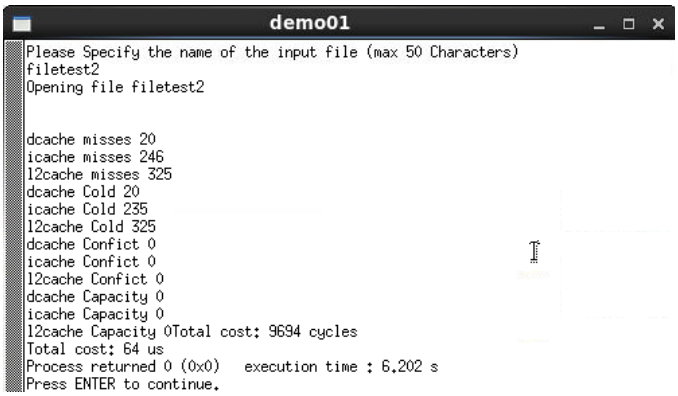

Fig. 2. Output of tool developed for analyzing the cache interference test case no 2.

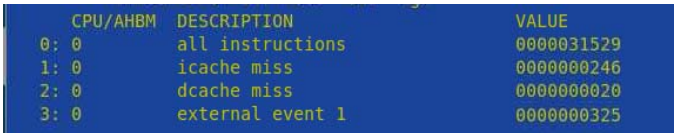

Fig. 3. Output of the statistic unit's counters of the device, test case no 2.

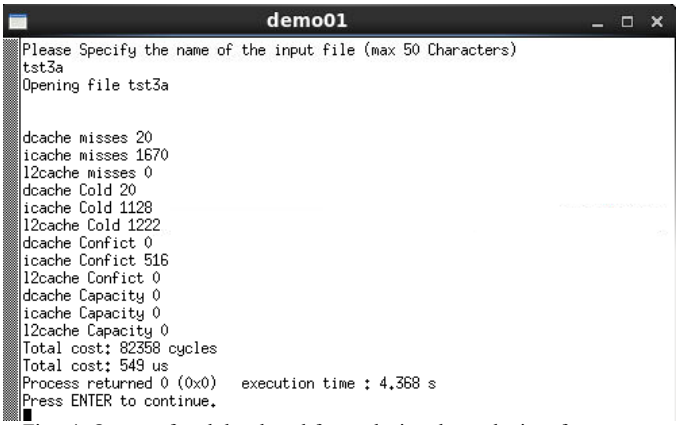

Fig. 4. Output of tool developed for analyzing the cache interference test case no 3.

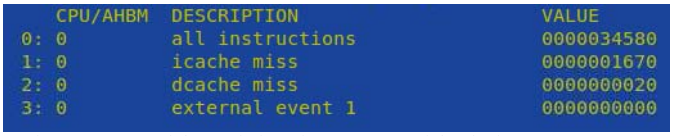

Fig. 5. Output of the statistic unit's counters of the device, test case no 3.

was compared with the measurements taken by the LEON4 Statistics Unit counters and they matched perfectly. In Fig. 3. and Fig. 5. the output of LEON4 Statistics Unit counters for test cases 2 and 3, respectively is shown. In Fig. 2 and Fig. 4. the output of our tool for test cases 2 and 3, respectively is displayed. It is obvious that the fields for dcache, icache and L2 misses from our tool match the corresponding fields produced by the LEON4 Statistics Unit counters. For the test case no 2 all the misses are classified as cold as it is shown in Fig. 3. Test case no 2 was a very small program that did not fill the icache and thus no conflict or capacity misses were produced. Test case no 3 was deliberately designed to fill the instruction cache and thus to induce conflict misses. Moreover, in Fig. 3. and Fig. 5. the calculated overhead by the cache activity is shown, under the field "total cost". As it is easily recognized a higher number of misses results in a greater latency.

V. Conclusions and Future Work

In this paper, a tool that analyzes the memory access trace of a program executed bare metal and calculates the latency added by the cache activity was presented. The tool was built using the C language and targeting the LEON4-N2X device. Its input is a text file containing the memory access trace to be processed and its output can either be printed on the screen or in a file and states the latency added by all misses of both cache levels, the number of L1 icache misses, the number of L1 dcache misses, the number of L2 misses and how many of them are Compulsory, Conflict or Capacity misses. The reuse distance was used as a metric of locality. The tool can be used to evaluate how cache friendly is an application with respect to the specific device and estimate how cache friendly would it be if run to a similar device with different cache placement policy or different capacity. This information can be used for optimization of the software or whenever it is possible the hardware.

References

[1] O. Temam, C. Fricker, and W. Jalby, "Cache interference phenomena," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 22, no. 1, pp. 261–271, 1994.

[2] Y. Zhong, S. G. Dropsho, X. Shen, A. Studer, and C. Ding, "Miss rate prediction across program inputs and cache configurations," *IEEE Trans. Comput.*, vol. 56, no. 3, pp. 328–343, 2007.

[3] A. Pesterev, N. Zeldovich, and R. T. Morris, "Locating cache performance bottlenecks using data profiling," *EuroSys '10*, p. 335, 2010.

[4] "Valgrind." [Online]. Available: http://valgrind.org/docs/manual/cg-manual.html. [Accessed: 13-Jan-2017].

[5] "GNU gprof." [Online]. Available: https://sourceware.org/binutils/docs/gprof/. [Accessed: 13-Jan-2017].

[6] X. Shen, J. Shaw, B. Meeker, and C. Ding, "Locality approximation using time," *ACM SIGPLAN Not.*, vol. 42, no. 1, p. 55, 2007.

[7] R. Sen and D. a. Wood, "Reuse-based online models for caches," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 1, p. 279, 2013.

[8] Y. Jiang, E. Z. Zhang, K. Tian, and X. Shen, "Is reuse distance applicable to data locality analysis on chip multiprocessors?," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2010, vol. 6011 LNCS, pp. 264–282.

[9] K. Beyls and E. H. D'Hollander, "Reuse distance as a metric for cache behavior," in *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, 2001, pp. 617–662.

[10] M. D. Hill and A. J. Smith, "Evaluating Associativity in CPU Caches," *IEEE Trans. Comput.*, vol. 38, no. 12, pp. 1612–1630, 1989.

[11] Aeroflex Gaisler AB, "LEON4-N2X Data Sheet and User's Manual," 2015. [Online]. Available: http://www.gaisler.com/doc/LEON4-N2X-DS.pdf. [Accessed: 01-Jan-2016].

[12] SPARC International Inc., "The SPARC Architecture Manual V8," p. 295, 1992.

[13] J. Reineke, "Randomized Caches Considered Harmful in Hard Real-Time Systems," *Leibniz Trans. Embed. Syst.*, vol. 1, no. 1, p. 03:1-03:13, 2014.