Specific Read Only Data Managementgreg for Memory Hierarchy Optimization

Gregory Vaumourin, Dombek Thomas, Guerre Alexandre
CEA, LIST
Embedded Computing Laboratory
Gif-sur-Yvette, F-91191 France.
{firstname}.{lastname}@cea.fr

Denis Barthou
INRIA Bordeaux Sud-Ouest, LaBRI
Bordeaux Institute of Technology,
Bordeaux, France
denis.barthou@labri.fr

ABSTRACT

The multiplication of the number of cores inside embedded systems has raised the pressure on the memory hierarchy. The cost of coherence protocol and the scalability problem of the memory hierarchy is nowadays a major issue. In this paper, a specific data management for read-only data is investigated because these data can be duplicated in several memories without being tracked. Based on analysis of standard benchmarks for embedded systems, this analysis shows that read-only data represent 62% of all the data used by applications and 18% of all the memory accesses. A specific data path for read-only data is then evaluated by using simulations. On the first level of the memory hierarchy, removing read-only data of the L1 cache and placing them in another read-only cache improve the data locality of the read-write data by 30% and decrease the total energy consumption of the first level memory by 5%.

Keywords

read-only data, memory hierarchy, cache, data management

1. INTRODUCTION

As demands for higher performance keep growing, multi-core systems have become popular in embedded systems. Memory system design is a critical problem for multi-core embedded systems. With the increasing number of cores, the cost of adopting hardware-controlled caches and ensuring coherency in embedded systems becomes extremely high. There are two reasons for this cost increase. Firstly, the power overhead of automatic memory management in memory caches is growing fast. It represents almost half of the overall energy for a single-processor [1]. Secondly, the coherence protocols lack scalability beyond a certain number of cores. Two basic memory models are used for the memory [11]: hardware-managed caches and software-managed scratchpads (also called local stores or streaming memories). Uniprocessors have dominant and well-understood models for memory organizations. Whereas for multi-core designs, there is no widespread agreement on the memory model. Cache memories are composed of tag, data RAM and management logic that make them transparent to the user. They exploit the spatial and temporal locality of data. Their major drawbacks are their important power consumption and the lack of scalability of current cache coherence systems. One solution to these problems is to use scratchpad memories. They consume less energy for the same memory size [1] and have a smaller latency because they are composed of simple array of SRAM cells without tags or complex logic comparators. Moreover, they do not generate traffic caused by the coherence protocol but they introduce programmability burdens because they need to be explicitly managed by the user. In order to resolve this problem, users can rely on compiler code generation for scratchpads management. Methods for automatic data management for specific data on scratchpads have been proposed in many related works [12, 6, 14]. These solutions are mostly specific to the behavior of data in the application. This paper is focused on a particular kind of data: read-only data, that is data that are set only once for the whole application execution. We will consider also some particular cases where data is read-only for a limited span of the execution. They offer interesting optimization possibilities thanks to the fact that they are easier to manage. Indeed, they can be duplicated in the memory system without being tracked. By handling differently the read-only data, the energy consumption of the memory hierarchy could be reduced without adding complexity for developers. Data are used in a read-only way either for the whole application execution like input data, or for a limited scope such as a function, or a kernel. In the latter case, read-write data are in a read mode for a long time during the application's execution. The memory accesses of these data may also benefit from this specific memory organization. A data transfer cost between the two data paths must be considered in this case.

The long term perspective is to propose an architecture where read-only data are removed from the original memory hierarchy and are managed in a different memory organization. This idea is similar to the one in Harvard architecture where instruction and data are handled in different memories. This new memory subsystem would be added in parallel to a classic memory system, and optimized for read-only data. This solution aims to be transparent for the user and generic to embedded systems. In order to use this system

EWiLi'14, November 2014, Lisbon, Portugal.

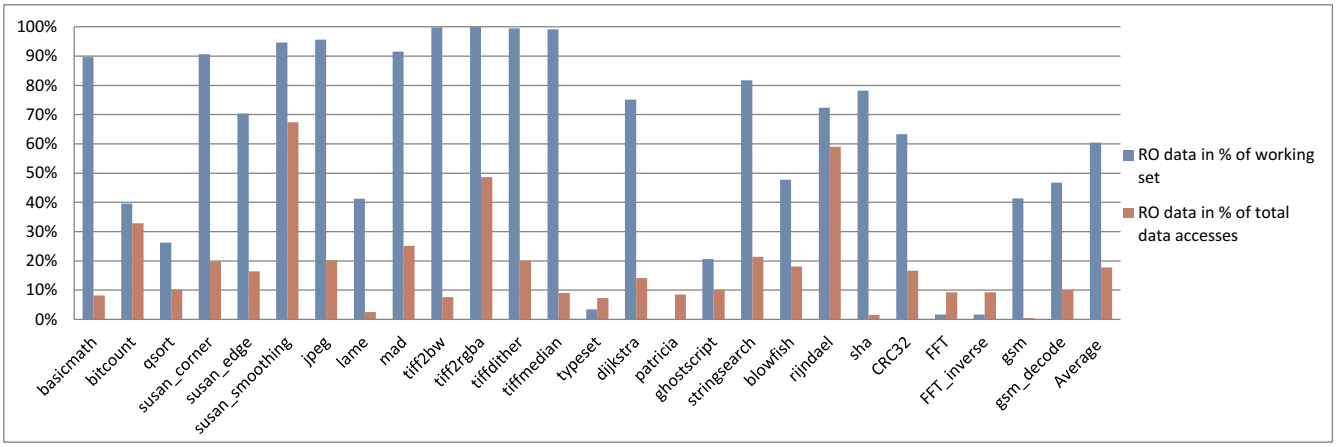

Figure 1: Proportion of read-only data and memory accesses done on read-only data

transparently, some steps are to be considered during compilation. The compiler has to detect read-only data, and may use some user information, such as those provided in parallel languages like OpenCL, or OpenACC.

This study considered several scenarios where read-only data are removed from the original memory organization and handled in a different one. The special management of read-only data is not optimized yet and will be considered in future work. Firstly, read-only data are detected and quantified for a whole set of applications. Secondly, by using simulation, different data management are tested and compared in terms of energy consumption and data locality. The rest of the paper is organized as follows: Section 2 describes a quantification on read-only data. In Section 3, several scenarios of memory access separation are introduced. These scenarios are compared in terms of data locality in Section 4 and in terms of energy consumption in Section 5. Finally, related works are discussed in Section 6.

2. READ-ONLY DATA ANALYSIS

The first step of the evaluation is to show that read-only data count for a significant part of the working set of applications in embedded systems. This analysis is a trace-driven analysis on the standard Mibench benchmark [10]. Mibench is studied because it is a representative set of applications used in embedded systems. All the applications of the benchmark are compiled on a x86 platform and are used with their default input data set given with the applications.

The memory access analysis is performed through traces. A trace records the flow of memory access occurring during an execution of the application (including those done in external libraries and in the stack). The study focuses only on data, instruction fetches are not recorded. The trace allows to launch several simulations on the same flow of memory accesses to compare different memory hierarchies and data management policy. The generation of the trace file is achieved by Maqao [2], a static instrumentation tool that operates directly on binary code. It is used to record which address is read/written for every instruction that access the memory and the size of the accessed data. The memory trace is compressed on the fly with the zlib library. Statistics about read-only data are deduced from the trace.

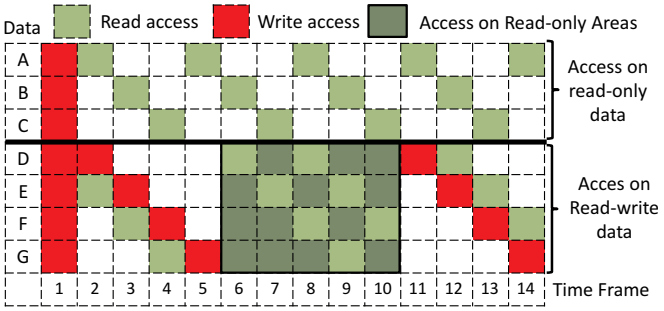

Figure 2: Example of the memory access classification

The analysis results are shown in Fig.1. On average, 62% of used data are in a read-only state but they represent only 18% of the accesses made by the application. The proportions of read-only data and the number of accesses are very asymmetrical. It could be partially explained by the fact that the stack is not removed from the analysis. The data in the stack represent few data, but the same stack addresses are used many times. Intuitively, this asymmetry between access proportion and data proportion suggests that read-only data are not reused as much as other data and can cause some pollution in the memory systems. This simple analysis shows that read-only data count for a significant proportion of data used by applications in embedded systems. It is important enough to consider some specific memory hierarchy optimizations for these data. In the following sections, the data path separation between read-only and read-write data is studied.

3. SCENARIOS PROPOSITION

In order to explore the possibility of adding specific data path in the memory hierarchy for read-only data, memory accesses are divided in several categories. Two scenarios of memory access separation are tested on simulation and analyzes are performed for comparison.

3.1 Classification of memory access

For this study, memory accesses are classified as follows: 1) accesses to read-only data, 2) accesses to detected read areas or 3) accesses that do not belong to one of the two previous categories. Read areas are defined as a group of

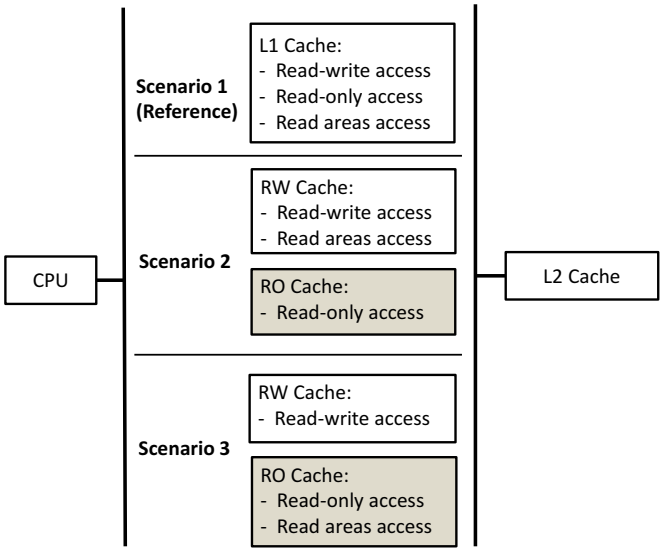

Figure 3: Scenarios for the read-only data management

read accesses which are not separated from each other by write accesses in time and address directions. This corresponds for instance to a read access to an array region. Read areas are detected on read-write data and cannot include accesses to read-only data so an access belongs to only one of three categories defined previously. The concept of read areas is introduced in order to place in the specific read-only memory, read-write data that present read-only behavior during execution. For example, read areas can be detected with intermediate results of an algorithm that are generated (written) first and then read for the rest of the algorithm. For a better understanding, an example of this classification is shown in Fig. 2. It represents the accesses made by an application during the first time frames. The data set is composed of 7 distinct data (A, B, C, D, E, F, G). All the data are initialized (written) in the first time frame. Then, A,B and C are only read (never written to) so they are considered as read-only data and all accesses to these A,B,C belong to the first category. Then, the area detection algorithm is launched on the remaining accesses and one read area is detected, the accesses in the dark green area in Fig. 2 belongs to the second category. All other accesses belongs to the third category.

A minimal size for the detected area is fixed for two main reasons. First, if this limit is not set, every read memory access can be considered as a read area on its own. Second, the proposed detection focuses on big areas of read access with data reuse, detecting data structures or data regions more than individual data accesses. After experimentations, read areas are kept for this study only if more than 128 memory accesses are done on this area. According to the defined classification, on Mibench benchmarks, the repartition shows that on average, 17.8% of all memory accesses are accesses to read-only data and 6.9% are accesses to read areas. The remaining 75.3% of the memory accesses are not concerned by the solution.

3.2 Data Management Policy

As mentioned in the introduction, the possibility of adding a new data path along the memory hierarchy specific for the read-only data is studied. All the other data use a classic cache hierarchy. The instructions are not considered in these simulations and are supposed to be handled in a different memory organization. The impact of adding this specific memory is studied on the first level of memory hierarchy. Fig. 3 shows three different scenarios studied in this analysis. Scenario n°1 is the reference scenario where there is no specific management for the read-only data and all the categories of access use the classic cache memory organization. In scenario n°2, accesses to read-only data are removed from the classic way and are handled in the specific memory. Scenario n°3 is the same as scenario n°2, but the read areas accesses are also placed in the specific memory. Since read areas are detected on read-write data, the data in read areas take both paths depending on the timing. During simulations, the specific memory is modeled as a simple memory cache. In order to simulate these scenarios, five memory traces are generated, one for each memory of each scenario. The original full trace of the application is used for the scenario n°1 as a reference. For scenarios 2 and 3, partial memory traces are generated from the original trace, accesses are removed according to the data management policy, in order to form the memory access flow for each memory. If no read area is detected, scenarios 2 and 3 are the same. This case happens for 10 out of 26 benchmarks tested. The L2 cache is unified for all the data. For the rest of the paper, the first level cache for read-only data will be called the RO cache and the first level cache for read-write data will be called RW cache. In the following section, two analysis are presented, a data locality analysis and an energy consumption analysis, to compare these scenarios.

4. DATA LOCALITY ANALYSIS

Data locality is important in order to take advantage of CPU caching. The data locality can be evaluated by the stack distance which is computed for all scenarios on all benchmarks.

4.1 Definition

The stack distance [5] measures the distance in time between the use and subsequent reuse of the same data location. It is an indicator of the temporal locality of the data and depends solely on the software. Bad temporal locality leads to pollution in the memory hierarchy. The pollution happens when a data is loaded in the cache and is evicted from the cache before being reused. In this situation, copying the data in the cache is a waste. Moreover, it takes the place of another potentially more interesting data in the cache. The application should access the data directly through main memory. For a LRU (least recently used) fully-associative cache, cache misses can directly be deduced from the stack distance computation. For more complex caches, the stack distance remains still a good predictor [4]. Generally, the higher the stack distance is, the higher is the probability that this access provokes a cache miss. Lots of algorithms are proposed in literature to compute the stack distance efficiently. The algorithm implemented for the scenario is based on the Bennet version [3].

4.2 Analysis

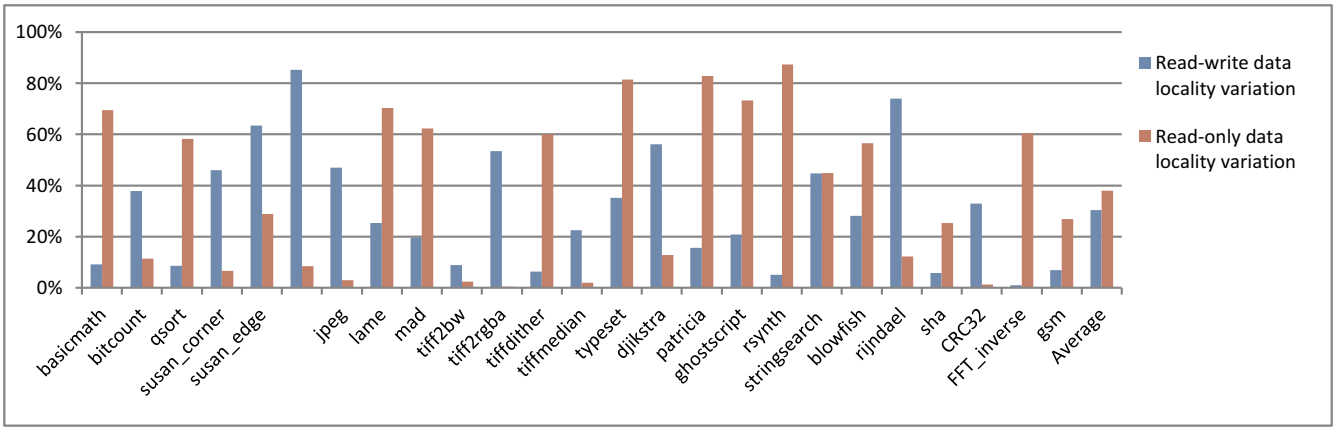

Figure 4: Stack Distance Variation between scenarios 1 and 2 for read-only data and read-write

To allow a meaningful comparison of different scenarios, the average stack distances are always compared separately for read-only and read-write data. Three steps are followed for the analysis: In a first step, for each application of Mibench, the stack distances are computed on the full memory trace of the scenario 1. Once the stack distances are computed for all accesses, the average stack distance is computed separately for accesses to read-only data and accesses to read-write data (read area accesses are included in the accesses to read-write data). For all the applications, the average stack distance of read-only data is 16 times higher than the average stack distance of read-write data. So, the difference of locality is very significant, and means that read-only data are less reused and pollute the classic L1 cache.

In a second step, the values of the stack distances computed previously are compared to the stack distance of scenario 2. The variation of the stack distance for read-only data and read-write data between the scenario 1 and 2 are shown in Fig. 4. The decrease of the stack distance for scenario 2 is expected because separating data always leads to global data locality improvement. The reason is that in each way of the hierarchy, the number of accesses between two calls to a same data is reduced. For the RW cache, only 18% of the accesses are removed and the stack distance is reduced by 30% and for read-only data, 82% of the access are removed and stack distance is reduced by 38%. This is asymmetric between the number of removed accesses and the decrease of the stack distance. Separating read-only and read-write data improves significantly the read-write data locality.

In a third step, a metric is introduced in order to compare the locality between all the scenarios. For the scenario 1, the average stack distance of the full trace is computed without data distinction. For scenario 2 and 3, a weighted sum is computed by adding average stack distances for read-only and read-write data in proportion of their respective number of accesses. It gives a comparable global stack distance for each scenario. The Fig. 5 shows the variation of the global stack distance of scenario 2 and 3 compared to scenario 1. On average, the global stack distance is improved by 19% for the scenario 2 and 30% for the scenario 3. For almost every application, the data separation improves the overall locality. These results suggest that it is profitable to separate the read-only and read-write data in terms of data locality. The following section studies the proposition with respect to energy consumption.

5. ENERGY CONSUMPTION ANALYSIS

The main motivation of this work is to reduce the energy consumption of the memory hierarchy. An energy model is introduced and the three scenarios are simulated to compute energy consumption. On the contrary to the stack distance, this analysis depends on the hardware.

5.1 Energy Consumption Model

The energy consumption of the cache is computed with a simple energy model. The CPU is not modeled and the study focuses only on the dynamic energy of the first level memories of the memory hierarchy. For each cache, the dynamic energy consumption is determined as in equation 1. The underlined terms are ignored for the moment. The *energyCPUStall* is the energy consummed when the CPU is stalled while waiting for the memory system to provide data and the *energyCacheBlockFill* is the energy for writing a block into the cache. The *energyPerAccessL1* and *energyPerAccessL2* are determined with Cacti v6.5 [13] and the *cacheHitsL1* and *cacheMissL1* are determined by simulations with the cache simulator dineroIV [7]. For the scenario 2 and 3, the energy consumption of the first level hierarchy is the addition of the energy consumption of the RW cache and the RO cache.

$$\begin{cases} DynEnergyCache = energyPerAccess* \\ nbHits + energyMiss * nbMiss \\ \\ energyMiss = energyPerAccessL2+ \\ \underline{energyCPUStall} + \underline{energyCacheBlockFill} \end{cases} \quad (1)$$

5.2 Analysis

The energy consumption of all the applications of the Mibench benchmarks is studied. To simulate with dineroIV and Cacti the described scenarios, cache designs have to be chosen. First, the scenario 1 is explored extensively in order to determine the most efficient design of the classic cache. The cache's design that minimizes the average energy consumption on Mibench is a cache of 16KB with 2-way set associativity. The energy consumption of the scenarios 2 and 3 is compared to this reference. To do a fair comparison between scenarios, an equivalent storage size at the first

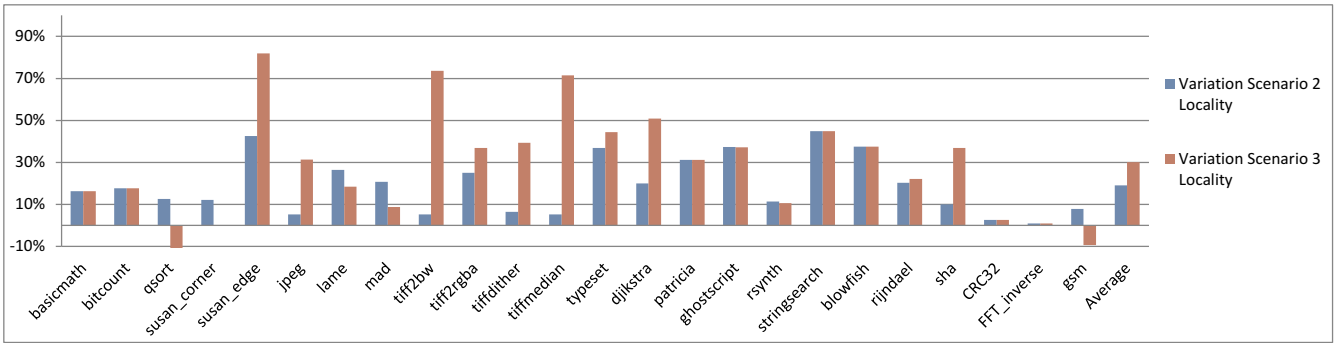

Figure 5: Global stack distance variation of scenarios 2 and 3 compared to scenario 1

Table 1: Memory hierarchy design for the simulation

Cache	Design
Classic Cache	16KB, 2-way associative, 64B/line
RO Cache	8KB, 2-way associative, 64B/line
RW Cache	8KB, 2-way associative, 64B/line
L2 Cache	4MB, 8-way associative, 64B/line

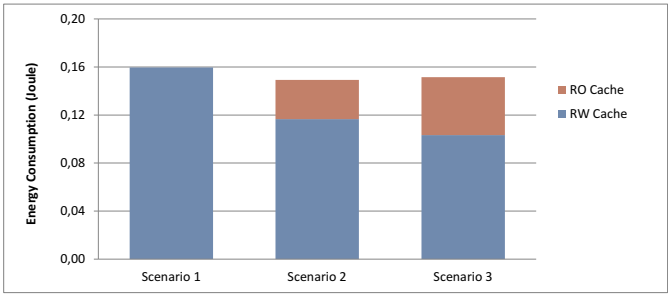

Figure 6: Energy Consumption of the scenario

level of the memory hierarchy for each scenario has to be used. In Section 4, it has been pointed out that read-only data have a weak locality, it suggests that the RO cache must have an important size relatively to the number of memory access it will handle. The choices for the design for the RW cache and the RO cache are shown in Table 1. All the applications are simulated independently in the cache, and the energy consumption model gives the total energy consummed by each application for the first level memories. The energy consumption of each application is then added to get the total energy consumption foFerr each scenario of all the Mibench applications.

Creating a two caches system instead of one has two consequences on the energy consumption. The resources are not mutualized so some of them can be underutilized leading to energy consumption increase. On the other hand, a good data separation decreases the pollution in each cache which reduces energy consumption. For each application of the Mibench benchmark, the generated memory traces are simulated through dineroIV and the energy consumption is computed for each application. Then, the energy consumption of each application is added for each scenario. As shown in Fig. 6 energy consumptions in scenarios 2 and 3 are approximately the same. There is an improvement of 6.5% of energy for scenario 2 and 5.0% for scenario 3 compared to the scenario 1. It shows that read-only data can be handled in a different memory without adding energy overhead. Indeed, even with conventional caches for the specific read-only memory, there is no overhead when dividing L1-cache. For scenario 3, the cost of switching read accesses between the RW and the RO cache has been ignored for the moment so the energy consumption for scenario 3 is probably under-evaluated compared to a real situation.

6. RELATED WORK

Proposing special memories on the hierarchy to manage specific data is not new. A lot of solutions have been proposed in literature to automatically use scratchpads memories for specific data management. The rest of the data, are often accessed directly through main memory or can go through a cache parallel to the scratchpad. Some examples could be found for the heap [6], the stack [14] or array tiles [12]. These solutions targeted uni-core systems and also multi-core systems. In [6], an algorithm for heap management in scratchpad is proposed. Managing the heap is challenging since the actual size of the data is known only at runtime. This solution divides the application in region and a compile-time analysis is performed on these regions to place the most used heap variables in the scratchpad. Code is added automatically to (de)allocate the scratchpad. In [12], an extension to the openMP compiler is proposed to place array tiles on scratchpads. The compiler realizes pattern recognition. It detects regular and irregular array access patterns and automatically produces code to activate the DMA transfers between scratchpads and the main memory in order to distribute the array tiles on the scratchpads. Partitioning cache is another technique that allows specific data management. The solution proposed in [15] separates I/O data from CPU data by adding a specific cache for I/O data. Even if the technique has different motivations, the work makes an analysis similar to the one presented in this paper. The specific data management of read-only data is not widely studied in literature. Some solutions have been proposed like in the Fermi and Kepler [8] architecture of the NVIDIA GPU's architectures. At the first level of the memory hierarchy, a specific read-only cache is added in parallel to a shared memory and a private L1 cache. The developer or the compiler needs to indicate the data that will go through the read-only cache. Another solution is proposed by Guo *et al* [9], on a VS-SPM (virtually shared scratchpad memories) architecture which proposes a solution of data management for shared data is proposed. The proposed algorithm allows the duplication of read-only data

in several scratchpads if the duplication comes with an energy reduction.

7. CONCLUSION

The paper has an analysis of the opportunity to handle in different caches, data that is read-only, either for the whole application or a limited amount of time. The results show that there are some optimization potentials for specific data management for read-only data. Even if the memories where read-only data are placed do not yet exploit the read-only property, simulations show that the division between read-only and read-write data on the first level of memory hierarchy improves the data locality (in average, 30% on Mibench benchmarks) and does not introduce energy consumption overhead. The main interest of exploiting read-only data appears in a multi-core environment, since they can be shared without being handled by a costly coherence protocol. The future work will focus on optimizing the read-only sub-hierarchy in an actual multi-core environment. Furthermore, the impact of compiler on the detection of read-only data and read areas need to be evaluated.

8. REFERENCES

[1] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, New York, NY, USA, 2002. ACM.

[2] D. Barthou, A. Charif Rubial, W. Jalby, S. Koliai, and C. Valensi. Performance tuning of x86 openmp codes with maqao. In M. S. MÃijller, M. M. Resch, A. Schulz, and W. E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 95–113. Springer Berlin Heidelberg, 2010.

[3] B. T. Bennett and V. J. Kruskal. Lru stack processing. *IBM J. Res. Dev.*, 19(4):353–357, July 1975.

[4] K. Beyls and E. H. Hollander. Reuse distance as a metric for cache behavior. In *In processings of the IASTED conference on parallel and distributed computing and systems*, 2001.

[5] E. G. Coffman, Jr. and P. J. Denning. *Operating Systems Theory*. Prentice Hall Professional Technical Reference, 1973.

[6] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *J. Embedded Comput.*, Dec. 2005.

[7] J. Edler and M. D. Hill. Dinero iv trace-driven uniprocessor cache. 1998.

[8] P. Glaskwsky. Nvida's fermi: The first complete gpu computing architecture. *White Paper*, 2009.

[9] Y. Guo, Q. Zhuge, J. Hu, J. Yi, M. Qiu, and E.-M. Sha. Data placement and duplication for embedded multicore systems with scratch pad memory. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(6):809–817, June 2013.

[10] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, Washington, DC, USA, 2001. IEEE Computer Society.

[11] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing memory systems for chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):358–368, June 2007.

[12] A. Marongiu and L. Benini. An openmp compiler for efficient use of distributed scratchpad memory in mpsocs. *Computers, IEEE Transactions on*, 61(2):222–236, Feb 2012.

[13] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Architecting efficient interconnects for large caches with cacti 6.0. *Micro, IEEE*, 28(1), Jan 2008.

[14] A. Shrivastava, A. Kannan, and J. Lee. A software-only solution to use scratch pads for stack data. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, pages 1719–1727, Nov 2009.

[15] D. Tang, Y. Bao, W. Hu, and M. Chen. Dma cache: Using on-chip storage to architecturally separate i/o data from cpu data for improving i/o performance. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, 2010.