On the Specific Read-only Data Management in Cache-based Memory Systems

Grégory Vaumourin[1], Alexandre Guerre[1], Thomas Dombek[1], and Denis Barthou[2]

[1] CEA, LIST, Algorithm and Architecture Codesign Laboratory, F-91191 France
[2] Inria / LaBRI, Univ. Bordeaux, Bordeaux INP, France

Abstract. Existing cache-based memory systems, especially in the context of embedded systems, present many challenges in power consumption and scalability. For computational loops of image processing applications, a previous study [14] has shown the difference in memory behavior for read-only data and read-write data. We propose in this paper to explore cache architectural designs able to take advantage of this difference, by separating these data streams. We show that this separation increases the overall data locality and reduces energy consumption. The design and integration of a dedicated memory for read-only data is discussed. We describe a compilation phase able to automatically detect and annotate memory accesses on read-only data in existing codes. This compiler analysis detects on average 89.3% of the memory accesses made on read-only data. Evaluated in a multicore environment with a set of multithreaded image processing benchmarks, the new memory organization with a shared read-only L1 cache shows on average an energy gain of 20.6% without any performance degradation.

1 Introduction

Caches are essential in modern processors to effectively hide the data access latency since memory reference patterns in most applications have good spatial/temporal locality. But with the number of cores increasing, adopting hardware-controlled caches increases the energy consumption for two reasons: The automatic data management in caches is getting more complex and coherence protocols do not scale very well. The cache hierarchy consumes between 25% and 50% of the energy of the chip [10] on current systems. This proportion motivates continued improvements in cache energy reduction.

For embedded systems, the combination of L1-level caches with scratchpads aims to get the best of both kind of memory. Scratchpads offer cheaper cost per access and a smaller area budget compared to caches but face programmability issues. Previous works [8, 12, 7, 4] often propose to manage only one particular kind of data into scratchpads. Such data are for instance array tiles[8], stack data [12], heap data [4], etc. These solution introduce a coherency problem by having two storages at the same level. In this case, it is up to the compiler/user to manually detect and resolve potentially incoherent accesses. Managing coherency

at compile-time can be quite difficult with non-predictable memory access patterns. In an other direction, few research works distinguish read and write in the cache hierarchy. Khan and al.[11] propose a new cache policy that discriminates read over write requests and partitions the last-level cache. They take advantage of the fact that stores can be buffered for some time before finally committing to the cache hierarchy. This solution enhances performance by resolving more quickly critical misses. Also, the NVIDIA Kepler architecture [5] shows the possible benefits of a read-only cache for a GPU. Using this cache removes loads and reduces the working set of the shared/L1 cache path. The read-only path is used automatically by the compiler or explicitly by the user. Similarly, a recent study [14] has shown for some crucial kernels used in image processing, there is an important difference in terms of data reuse and memory access distribution between read-only and read-write data.

The contribution of this paper is to explore the design of a cache specific for read-only data for multi-cores, and evaluate such solution for image processing applications, using a compiler analysis to automatically detect read-only data. The experimental evaluation shows that the proposed solution with specific read-only data management can decrease the energy consumption of the memory system by 20.6% on average without performance penalty on a set of multithreaded image processing benchmarks. The compilation support is evaluated to capture 89.6% of the memory accessed made on read-only data compared to an offline perfect analysis.

This paper is organized as follows: Section 2 motivates the need for a specific management of read-only data. Section 3 discusses the compilation support. Section 4 highlights our methodology for architecture exploration and evaluation and Section 5 describes the experiments results.

2 Background and Motivation

Read-only data are defined as data used in a read mode either for the whole application execution as for input data, or for a more limited scope such as a function, a kernel or a loop. In this case, read-write data can switch into read-only mode for some scope and then switch back to read-write mode. The size of the data and the length of the scope considered are obviously an important criteria for designing a specific data path for read-only data. We focus on image processing applications, since many of them manipulate 2D arrays in read mode for long time period. To assess the possible benefits of separating read-only data stream from read-write data stream, we extend the analysis conducted in [14] and evaluate *reuse distances* [3] for read-only and read-write data. This metric is defined as the number of different accesses between two accesses to the same address and is hardware-independent. First, a memory trace is extracted from an instrumented execution of the analysed application. Then, the read-only data and read-only periods are detected through an analysis of the memory stream. After this operation, the main memory stream is split in 2 : the read-only stream and the read-write stream. Finally, reuse distance analysis is run and an average

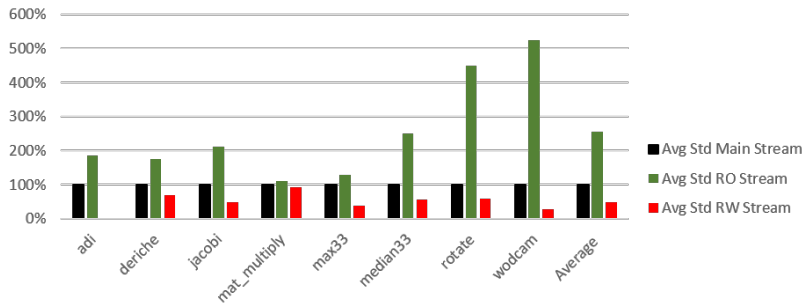

Fig. 1. Reuse distance for various benchmarks, obtained through memory traces, and separate reuse distances for read-write data and read-only data.

reuse distance is computed on both streams. The average reuse distance of the original stream and of the 2 sub-streams are shown Fig.1

On all the evaluated benchmarks, separating the RO (Read-Only) data from the main memory stream always improves its data locality. The average reuse distance of read-only data is 8.4 times bigger than the reuse distance of the read-write stream. The read-only data behave very differently in terms of reuse and removing these accesses from the memory stream improves its data locality and therefore its cache efficiency. Note that these results has been developed for the standard set of benchmarks Mibench in [14].

This working set analysis motivates the idea of a specific management of this particular part of the working set into the memory system. The following section explains the feasability of the automatic detection of the read-only data is developped.

3 Compile-Time Detection of Read-Only Data

In this section, a compilation support is considered in order to perform the detection of read-only memory accesses and to issue them in the appropriate data path. The compilation support is decomposed as follow: a static analysis and a cache collaborative technique.

3.1 Static analysis

The aim of the analysis is to detect read-only data and their scope. The static analysis contains two main steps. First, an intra-procedural pass analyzes the imbrication of loops inside functions. Since our analysis focuses on image processing applications, the intra-procedural pass focuses essentially on nested computational loops. For each loop, memory references are collected and read/write operations are detected in order to detect the status of the references: read-only or read-write. The aliasing analysis made by the compiler correlates the detected references to memory regions. Information about memory regions and

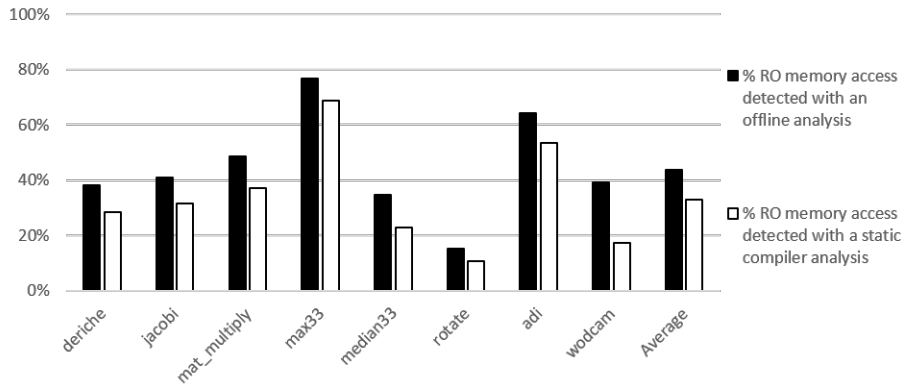

Fig. 2. Comparison between the read-only memory accesses detected with offline analysis and with the static analysis

references are then merged in order to get the status of memory regions. The data structures managed during this pass are pointers, multi-dimensional arrays and simple structures (no recursive structures).

The second step is an inter-procedural pass that finds in each function the constant memory allocations and follows pointers through function calls to keep track of the size of the read-only memory region. The analysis does not manage dynamically allocated memory regions. The analysis uses a conservative approach in order to ensure that every reference captured is read-only. For example, if there are conditional branches inside loops, only memory regions that are read-only for all execution paths are considered read-only. Memory regions are conservatively modeled by adresse intervals and, entire intervals are either read-only or read/write.

This analysis is implemented in GCC 4.9 through several GIMPLE passes, using the GCC plugin interface. The code analyzed can be C/C++ codes with OpenMP pragmas. The analysis performed by the plugin is limited for several reasons. First, the status (RW or RO) of a variable is not always determined at compile-time. The plugin is in an early stage of development and does not handle yet all the cases. To determine the quality of this static analysis, the results are compared to an exhaustive offline analysis based on memory traces. This offline analysis explores the trace in a 2D space (time and adresses) to detect rectangular areas solely composed of read accesses. Such analysis is manually tuned to detect read-only areas in an adequate grain in order to avoid capturing every read access as a read-only area. Results presented in Figure 2 show that despite its limitation, the compiler analysis is able to capture most of the read-only memory regions. The detection error is 10.8% in average for the considered applications compared to the offline analysis.

If the user wishes to refine the results, hints can be provided through custom pragmas to the compiler in order to enhance the detection. The user can indicate the start address symbol and the size of the read-only region in the block

immediately following the pragma. Memory regions that are manually specified with pragmas are removed from the scope of the static analysis.

3.2 Collaborative Cache Technique

The solution supposes that each processor knows for each memory access, which cache to access between the data cache and the read-only cache. Relied on the static analysis result, every memory request made on RO memory regions is flaged with on bit that encapsulate the status information of the request. It allows to drive the memory request between the one of the two caches so only one is looked-up. This idea is comparable to a number of current hardware systems that have been built or proposed to provide an interface for software to influence cache management. These techniques are known as collaborative caching and an old example is the cache hints on Intel Itanium [1]. Every memory load and store in such architecture has a 2-bits cache hint field in which the compiler encodes its prediction of the locality of the data being accessed. The memory system uses this prediction to fetch the data to the adapted memory. Others industrial[13] or research works[6] illustrate the use of cache collaborative technique. It has been proven that it can improve greatly cache management.

The implementation we have chosen is still simple because all detected read-only memory accesses are placed on the read-only data path, independently of a possible gain/loss this can bring. Based on these results, a automatic memory system optimization specific for the read-only data can be considered.

4 Methodology

We first present the architecture exploration and then the simulation methodology used.

4.1 Architecture Exploration

This section describes the architectural exploration of read-only data specific management. Figure 3 provides an overview of the methodology (L1 instruction caches not represented). The exploration focuses on two aspects: The level of memory where it is pertinent to add a RO cache and the shared/private property of this cache. It creates three possibilities presented Figure 3. The baseline architecture is composed of a cluster of cores with 2 cache levels, data are shared through an inclusive MESI protocol.

As mentioned previously, read-only data can be declared in read-only mode for a limited scope. It means that they can be accessed through both data paths depending on the execution time. Having two memory at the same level potentially creates coherency problems. The resolution of this problem can incur overheads if a cache line has to do a lot of switches between the RO and the classic data path. Note that we do not envisage direct transfer between memories of the same level.

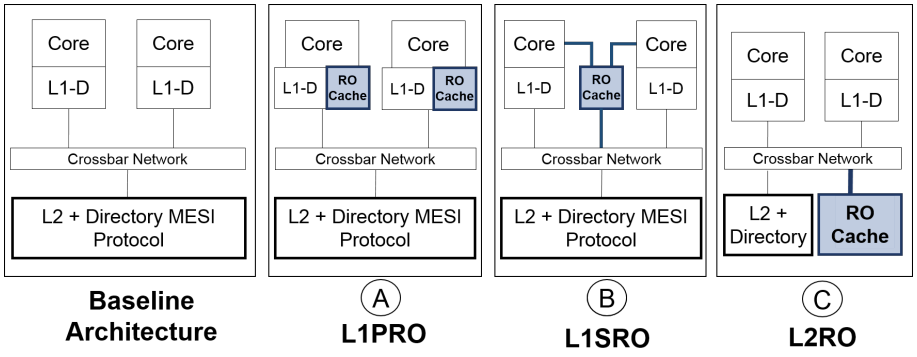

Fig. 3. Overview of cache design methodology for specific read-only data management

The first possibility is to attribute a private RO cache for each core (Scenario A in Figure 3). This solution is similar to the existing separation of the instruction cache and data cache. A cache line can not be in the two caches so the L1D-cache and RO cache are mutually exclusive. If a cache line present in one of the cache needs to be accessed by the other cache, it needs first to be invalidated and eventually written back to the L2 if modified. The second scenario considers a shared RO cache across all processors (Scenario B in Fig. 3). This situation is similar to Kepler architectures in NVIDIA GPUs where the read-only cache is shared across a whole warp. The coherency problem is resolved in this case by using the same MESI coherency state machine for RO cache as an L1D cache. In the directory protocol point of view, it is equivalent to have an other L1D cache to manage. It adds one bit overhead per directory entry into the sharers list. The RO cache shares cache lines the same way as a L1D cache. Note that in our simulations, the RO cache is not multi-ported because this increases prohibitively the cost per access which is not suitable for a cache close to a processor. Instead, the requests are sequentialized into a FIFO buffer and a processor may have to wait for another processor to finish its request before being serviced by the read-only cache. The last idea is to consider a specific management at L2-level. In this case, the L2 RO cache is shared across cores. The coherency problem is more difficult to solve. For each access, the cache line at L1-level is tested for its status, encoded in one bit. If there is a status switch, the L1 controller warns the L2 cache of a possible update for all sharers of this cache line. Then a switch of the cache line at L2-level is operated between L2 and L2RO (depending on the status modification). Note that during the switch, the L1 cache lines are not invalidated but this operation stalls the memory system in order to guarantee that the inclusive property of the coherency protocol is preserved. As in scenario A, the L2 caches are mutually exclusive. All these architecture possibilities introduce an overhead in their management due to cache line switching. Our conviction is that this overhead stays small compared the cache efficiency improvement these solutions can bring.

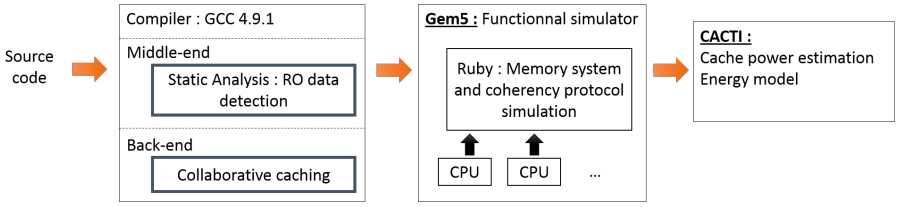

Fig. 4. Simulation Workflow

4.2 Simulation Protocol

For our experiments, a modified version of the Gem5 simulator [2] is used, with the sub-simulator Ruby modeling the memory system and the coherency protocol. It allows detailed simulation of the cache hierarchy. The dynamic and static power of each cache memory is obtained with Cacti [9]. Then an energy model taken from [15] computes the total static and dynamic energy of the memory system. The power of the interconnection network is not modeled in our results. The overall workflow is represented Figure 4. The baseline system configuration is shown in Table 1. For each scenario proposed, two sizes for the read-only cache are proposed because size is the most sensitive parameter for cache design. The additional storage needed for read-only cache is compensated by reducing in size the L2 cache, removing the necessary amount of ways. The Table 2 describes the storage balance between the L2 and the read-only cache. Note that the area taken by cache controllers are not considered here, which means that the comparison is storage-constant and not strictly area-constant. All caches are managed through LRU cache policy.

Table 1. Constant system parameters during simulations

	System Parameters
CPU	4 cores, in-order, 1GHz clock frequency 1 load/store unit
L1-D cache	32KB, 2-way associative, 64-byte blocks, 1 port
Local Network	Bidirectional bus, 64 bytes wide, 2 cycle latency
DRAM	512MB one memory channel at 12.8GB/s

Table 2. L2 and read-only cache storage Allocations

	Baseline	**Scenario A**		**Scenario B**		**Scenario C**	
RO Cache	- -	8kB FU	16kB 2-way	16kB 2-way	32kB 2-way	64kB 2-way	128kB 4-way
L2 Cache	256kB 8-way	224kB 7-way	192kB 6-way	224kB 7-way	224kB 7-way	192 kB 6-way	128kB 4-way

5 Experimental Results

5.1 Benchmarks characterization

All benchmarks are presented in Table 3. They come from a in-house set of benchmarks and are written in C, parallelized with OpenMP and compiled with the presented plugin. Most of them are taken from on image processing domain.

Table 3. Benchmarks description

Benchmarks	**Input Data**	**Description**	**% of detected RO Accesses**	**% of Shared RO Accesses**
matrix_multiply	512x512 arrays	Blocked matrix multiplication	37.0%	37.0%
deriche	1024x1024px	Canny edge detector	28.6%	25.8%
rotate	1024x1024px	Image Rotation algorithm	10.8%	0.1%
max33	1024x1024px	Image Blur Algorithm	68.8%	0.4%
median33	1024x1024px	Image Median Algorithm	23.0%	0.1%
jacobi	1024x1024 mat	Iterative solver for a system of linear equations	31.7%	9.9%
adi	256 elements 10 iterations	Solver of nonlinear differential equations	53.3%	53.3%
wodcam	168x192px 1000 images	Face recognition based on Eigenface method	14.6%	12.6%

The proposed solutions are compared in terms of energy reduction, for equivalent performance. The variation of the $AMAT$ (Average Memory Access Time), the energy consumption and the misses at L1-level are shown Fig. 5. Note that the RO cache misses are counted as L1-level misses for Scenario A and B.

5.2 Scenario A

In scenario A, the number of cache misses stays relatively similar to the baseline even if the miss rate of L1D caches is decreased by an average of 17.56%. As anticipated by the result of Section 2, the locality of the main memory stream is improved when the read-only stream is removed. Note that this depollution of the L1-D caches is observed in the same proportion in scenario A and B for every RO cache size. The number of misses stay similar because, miss rates stay relatively high for the read-only cache. Read-only accesses represent only 28.5% of the total memory access in average and they are divided between the 4 cores. In this case, private read-only caches handle very few memory accesses and yet are not able to resolve misses. It nullifies L1D-cache efficiency improvements and points out a poor utilization of the RO cache ressouces. The small energy improvement in this proposition is based on the fact that the read-only memory accesses are handled in a smaller cache than the baseline, the energy cost per access is lower.

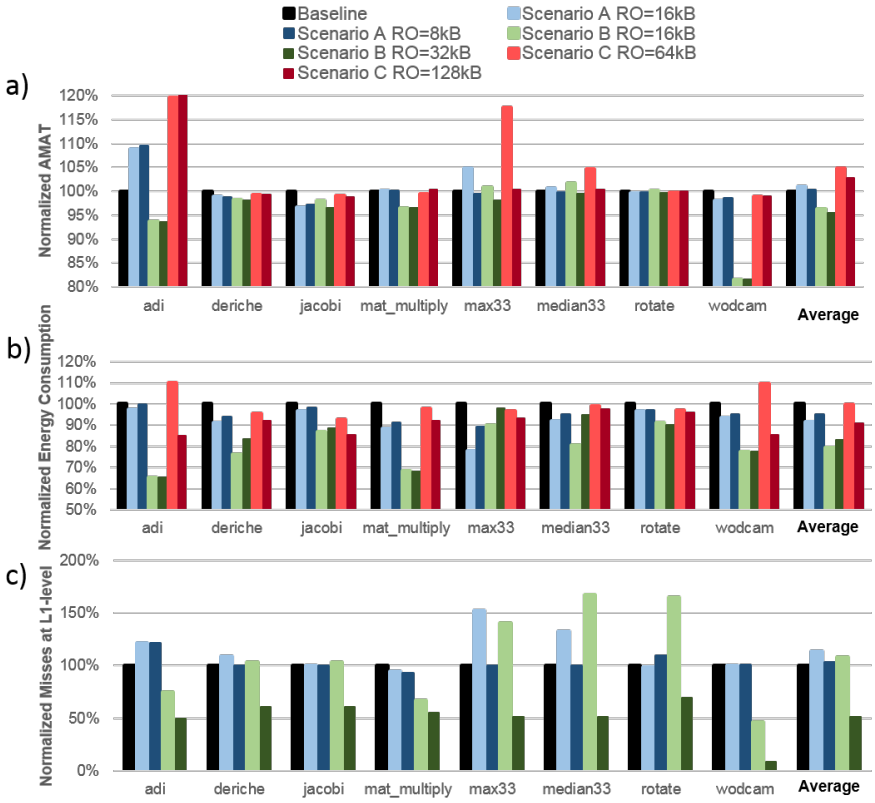

Fig. 5. a) Normalized AMAT variation and b) Normalized energy consumption variation of the architectural propositions. c) Normalized miss variation at L1-level for scenario A and B

5.3 Scenario C

Placing the specific management at L2 level brings limited energy reduction and no performance gain. Generally, less reuse occurs at L2-level compared to L1 level. Dividing the memory stream at L2 level impacts this reuse and creates a prohibitive increase of main memory accesses: +46% for 128kB L2RO case and +70% for 64kB L2RO case. The system loses time and energy due to an inefficient way to switch cache lines between the two L2 caches that has to transit through main memory. One may propose a better mechanism or a cache partitioning technique in this case, such as proposed by Khan and al.[11], to improve this solution.

5.4 Scenario B

Scenario B presents a great cache miss reduction for a 32kB read-only cache and the best energy improvement. With a RO cache of 16kB, the misses reduction

is not as important but due to a lower cost per access, there are more energy savings due to a more aggressive cache design. Generally a shared cache implies a better use of resources because reuse can occur for the same cache line between threads into the RO cache, so misses can be resolved by having a shared cache. The drawback is that it can entail destructive interferences or performance degradations due to resource contention. Since read-only accesses represent only 28.5% of the total memory accesses and the improvement on cache efficiency is important, the negative effects on performance are canceled. The miss penalty observed for *rotate*, *max33* and *median33* are not significant. The baseline of these benchmarks behaves very efficiently, the miss rate at L1 level stays under 1% and most of the cache misses are compulsory, nothing the proposed separation can resolved. The additional cost caused by the miss increase is largely compensated by the smaller cost per access of the RO cache in these cases. *adi* and *mat_multiply* have important miss reduction at L1-level even for a 16kB RO cache (25% and 33%). The reason is that most of their read-only working set is shared and reuses occur between cores into the read-only cache. We can oberserve that particular behaviour greatly benefits from this scenario. This solution is the best for energy improvement so particular analysis is made in the next sub-sections for the scenario B with a RO cache of 16kB.

5.5 Impact of the RO cache latency

Concerns can be made about the performance results presented previously for the scenario B. During the experimentations, the RO cache is configured with the same latency as a classic L1-D cache because they are logically on the same level. In a real system, the RO cache would be accessed in a longer time than a L1D cache because it is shared across cores. Moreover, the latency of this cache is more critical than a classic L1D cache because it can stall all processors trying to access to the RO cache. Additional experiments have been made in order to study the impact on the overall system performance of the RO cache latency. All values betwee 3 cycles (the L1D cache latency) and 15 cycles (The L2 cache latency). The relation between the latency of the RO cache and the average AMAT degradation relative to the baseline is linear with the formula : $\%AMATdegradation = 1.67\% * ROLatency + 0.9267$. So, each cycle added to the RO cache latency adds an average AMAT degradation of 1.67%. For a latency of 5 cycles, the AMAT degradation is null compared to the baseline architecture. The 5% degradation barrier is crossed with a RO latency of 9. These results give input s for a more realistic implementation of the solution and illustrate the feasability of the solution and .

5.6 Scalability of the solution

We propose to observe Scenario B when changing the number of cores. The average AMAT of all the presented benchmarks is computed and normalized to the baseline unicore solution (Fig.6). In a perfect scalable system, the AMAT should stay stable regarding the number of cores. In our scenario, as the number of cores

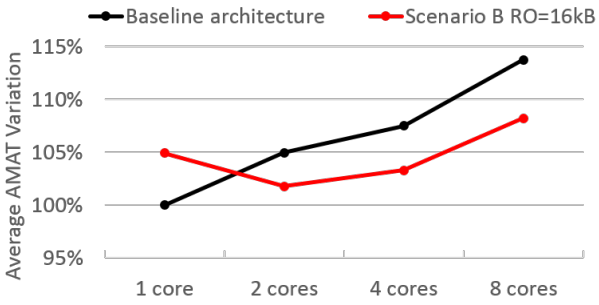

Fig. 6. Average AMAT variation depending on the number of cores

increases, the memory system increases its response time to memory requests, due to the increased pressure of the coherency protocol. Having a shared read-only cache at L1-level allievates this effect because sharing cache lines through the RO cache is less harmful in terms of control. The amat degradation acceleration is more important for the baseline system compared to scenario B. We conclude that our solution increases the scalability of coherent-based memory system. The counter effect of this improvement is that it increases the pressure on the RO cache bandwidth that can become the bottleneck of the solution.

6 Conclusion

In this paper the feasibility and benefits of a new specific cache for read-only data is studied. This solution exploits the properties of the different sub-workloads in order to increase the overall data locality and cache efficiency, much like the usual separation of instruction and data today. Our conviction is that a modification of memory architecture will only be deployed if the benefit can be achieved with no additional effort from the programmer. To this end, we have implemented and tested a compiler extension has been implemented and tested. The detection rate is shown to be 89.6% in average close to the theoretical maximum. The potential benefits of such organization have been demonstrated on a modified memory architecture compared to a configuration of equivalent storage capacity. Simulated in a multicore environment, the evaluation of the memory hierarchy with a shared RO cache at L1-level shows an average of 20.6% of energy saving without any penalty on the performance. This solution opens design space for specific optimization for the read-only cache. Despite the findings and revealed potential, there is no doubt that much further studies can be made to find specific optimization for the RO cache that can enhance these results. An optimization track can be that since cache issues only read requests, it is not concern about coherence problems and can emit non-coherent read requests which would decrease the impact of the coherence protocol and the number of messages issued on the network.

References

1. K. Beyls and E. H. D'Hollander. Generating cache hints for improved program efficiency. *J. Syst. Archit.*, Apr. 2005.
2. N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, and R. Sen. The gem5 simulator. *SIGARCH Comput. Archit. News*, 2011.
3. E. G. Coffman, Jr. and P. J. Denning. *Operating Systems Theory*. Prentice Hall Professional Technical Reference, 1973.
4. A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *J. Embedded Comput.*, 1(4):521–540, Dec. 2005.
5. P. Glaskwsky. Nvida's fermi: The first complete gpu computing architecture. *White Paper*, 2009.
6. X. Gu and C. Ding. On the theory and potential of lru-mru collaborative cache management. *SIGPLAN Not.*, 46(11):43–54, June 2011.
7. A. Jadidi, M. Arjomand, and H. Sarbazi-Azad. High-endurance and performance-efficient design of hybrid cache architectures through adaptive line replacement. In *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, pages 79–84, Aug 2011.
8. 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.
9. N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Architecting efficient interconnects for large caches with cacti 6.0. *Micro, IEEE*, 28(1):69–79, Jan 2008.
10. S. S. Low power design techniques for microprocessors. In *International Solid State Circuit Conference*, February 2001.
11. K. Samira, A. Alaa, and W. Chris. Improving cache performance by exploiting read-write disparity. In *In processings on High Performance Computer Architecture (HPCA)*, 2014.
12. 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*, 28(11):1719–1727, Nov 2009.
13. B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. *IBM J. Res. Dev*, 2005.
14. G. Vaumourin, T. Dombek, A. Guerre, and D. Barthou. Specific read only data management for memory hierarchy optimization. In *Proceedings on the 4th Embedded Operating Systems Workshop*, 2014.
15. C. Zhang, F. Vahid, and W. Najjar. A highly configurable cache architecture for embedded systems. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 136–146, June 2003.