Dedicated read-only/read-write cache design for data locality and coherence optimization*

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

Denis Barthou
INRIA Bordeaux Sud-Ouest, LaBRI
Bordeaux, France
denis.barthou@inria.fr

ABSTRACT

With the end of the CMOS scalability, it is important to find new ways to increase the energy efficiency of the memory system, as caches represent a significant portion of the energy. One classic way is to propose heterogeneous data management with specialized cache structures or policies to exploit the different properties that exist in subgroups of data. Heterogeneous data management has been studied mostly for two reasons: data locality improvement and coherence optimization. Read-only data specific management can help in both directions as they do not require coherence tracking and has a potential for data locality improvement. This work proposes a hardware-software co-design where the RW/RO data classification is performed by the compiler while a cache architecture is proposed with a dedicated cache for read-only. The light-weight and transparent classification scheme achieves a detection rate of 89.3% in average when compared to an offline analysis. Evaluated in a multi-core environment with a set of multi-threaded image processing benchmarks, the architecture takes benefits of the locality difference between RW/RO data to increase the cache reuse and achieves an average of 16.7% of energy savings without any loss in performance compared to a standard memory system.

KEYWORDS

Cache memory, read-only, data locality, coherence

ACM Reference Format:
Gregory Vaumourin, Alexandre Guerre, Thomas Dombek and Denis Barthou. 2018. Dedicated read-only/read-write cache design for data locality and coherence optimization. In *Proceedings of ACM International Conference on Computing Frontiers 2018 (CF 2018).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Caches are widely used in modern processors to effectively hide the data access latency. But with the number of cores increasing, the use of hardware-controlled caches increases the energy consumption and they are shown to take up to 45% of the core [20]. With the growing focus on energy efficiency, it is important to find ways to reduce energy without sacrificing performance. Usually the memory stream is handled in an uniform way by the memory system that has limited knowledge about the data being loaded. Heterogeneous management classifies data or accesses into several groups in order to isolate and take advantage of the properties that exist in sub-sets of the memory stream. In the state of the art, such techniques are used for two reasons: locality and coherence optimization.

For locality optimization, reuse patterns and cache needs have been shown to be very different between different classes of data such as instructions, heap data, stack data ... Those locality differences can be exploited specifically into dedicated cache structures. This type of solution has been studied with success for since the 90s, many solutions [4, 9, 10, 16] proposes specialized cache memories for a specific type of data (stack cache, heap cache ...). They bring important dynamic energy consumption reduction if the data classification results in sub-groups of data with very different locality properties. The main difficulty of those proposition is to classify data at hardware-level and drive the request to the relevant sub-memory. For coherence optimization, plenty of solutions classify data between shared and private as it has been shown that only small amount of data actually requires coherence [7]. It helps building light-weight coherence protocols and and therefore increasing the architecture scalability. In this context, lots of classification mechanisms (TLB, OS pages, hardware, ...) have been studied to perform the classification.

Mixing the two approaches, this paper studies the benefit of a dedicated RW/RO cache design both for locality and coherence optimization. Note that this property is dynamic so data can exist in a read-only state for a specific amount of time. The coherence potential is simple as read-only data can simply be taken out of the coherence tracking and the locality potential has been illustrated in a related work[21]. This study has shown a high potential of a heterogeneous management of read-only data, especially for image processing applications due to their implementation as a pipeline of tasks.

The contributions of the paper are the following:

- An illustration of the difference of behavior between read-only and read-write data
- A full hardware/software codesign solution for the specific management of the read-only data in the cache hierarchy.

The experimental evaluation shows that the data classification scheme performed at compile-time, achieves a detection rate of 89.3% in average compared to an offline analysis technique. Also, our proposed solution decreases the energy consumption of the

*Produces the permission block, and copyright information memory system by 16.7% and the network traffic by 26% on average without a performance penalty on a representative set of multi-threaded image processing benchmarks.

2 RELATED WORK

As mentioned before, the specific management in the memory system based on based on data classification has been successfully studied in the past. These solutions creates two (or more) sub-groups of data with different properties that can be exploited separately. This heterogeneous data management solutions has been proposed in previous works for 2 mains reasons: either locality or coherence optimizations.

2.1 Locality Optimization

Apart from the locality principle, the data locality properties are far from being uniform across application's working sets and those differences can be exploited to increase the data reuse in memory caches and proposed more specialized and therefore energy efficient cache designs. For example, Gonzales *et al.* [10] first proposed L1 data cache with separate sub-structures, one for data with high temporal locality (temporal cache) and one for high spatial locality data (spatial cache) that has been improved by Adamo *et al.* [1]. This is based on the observation that most data present either spatial locality such as arrays or temporal locality such as stack data, few data have both temporal and spatial locality at the same time. The cache sub-structures can be specialized for the handled data which leads to great improvement in terms of data reuse and energy efficiency. Lee *et al.* [16] propose to divide the data cache into separate structures for stack, heap, and global data. Geiger goes further by splitting the heap cache into 2 sub-parts [9]. Also, those solutions show great energy savings, they rely on being able to distinguish data types based on the address which is not realistic on current systems. Kang *et al.* [4] proposes an efficient stack data detection scheme based on the stack pointer register. As those methods show great energy efficiency improvement, identifying the different classes of data is difficult is a key problem there as it stays quite simple and at hardware-level. Also, none of those solutions makes distinctions based on read/write access patterns of data. The locality difference between RW and RO data has been studied in a previous study [OMMITTED FOR BLIND REVIEW] and an extension of this analysis is presented further in this paper, which justify the potential of locality improvement of dedicated RO data management.

2.2 Coherence Optimization

In order to reduce the coherence overhead and increase the scalability of architectures, a common approach is to classify data between private and shared [7, 8, 13, 14, 18, 19]. It allows to build smaller directory designs and reduce the network traffic. To classify between shared and private, the OS approach is the most common for such optimization because it does not impose extra requirements for dedicated hardware, since data classification at page granularity is stored along with the page table entries (PTEs) [13]. These solutions achieves great efficiency as they allow to build a new class of light weight coherence protocol without directory. However, they

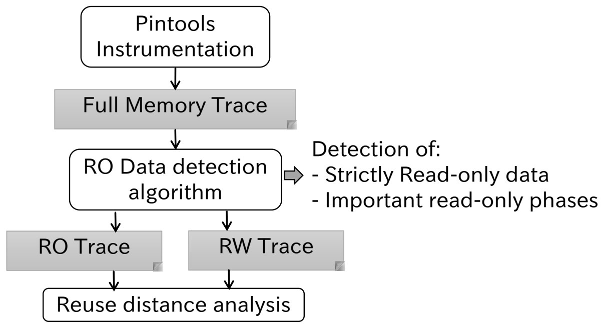


Figure 1: Workflow of the data locality analysis. Based on memory traces, read-only data are detected offline and sub-traces are generated. The data locality of each trace is compared through average reuse distances.

mostly detect read-only data at page granularity in a reactive fashion which can decrease the efficiency of the detection. Although incurring minimum hardware cost, compiler-driven classification is well suited for hardware/software co-design, but requires significant effort to determine at compile time the shared state of a variable. The only work in that direction is from Kim *et al.* [15]. To overcome the limitation inherent to compile-time analysis, the compiler uses a relax definition of the shared data and classification error are allowed in order to increase the proportion of detected shared data. Hardware mechanisms are built to recover from classification errors. The RO/RW classification in this context is seen as an extension on top of the private/shared classification. Also, they usually do not propose specific structures to store the shared data that would allow to optimize for locality and cache resource as well. An exception to this is the analysis of Cebrian *et al.* [5] that studies a dedicated shared L1 cache that is logically shared but physically distributed across all cores. This solution shows an important L1 misses reduction, however it does not scale very well as remotely accessing a cache line in the shared cache can be as expensive as a miss in the L2 cache.

Previous works show that heterogeneous data management can significantly increases the energy efficiency of the memory system and an efficient classification scheme is a key point for achieving a good solution is this context. However, none of the previous works directly address the problem of the specific management of read-only data. It could allow to optimize potentially for both locality (as illustrated next Section) and coherence.

3 READ-ONLY DATA ANALYSIS

This section illustrates the locality difference between RO and RW data in order to justify the proposition of a dedicated read-only data cache. We define read-only data as data used in a read mode either for the whole application execution (input data) or for a more limited scope such as a function, a kernel or a loop. In this case, RW data can switch into RO mode for some scope and then switch back to RW mode. Our analysis is mostly focused on image processing applications, since these applications are often implemented as a pipeline of tasks where the output (written) data of a task corresponds to the input (read) data of the next one. Such computations manipulate large regions that are in a read mode on a long scope.

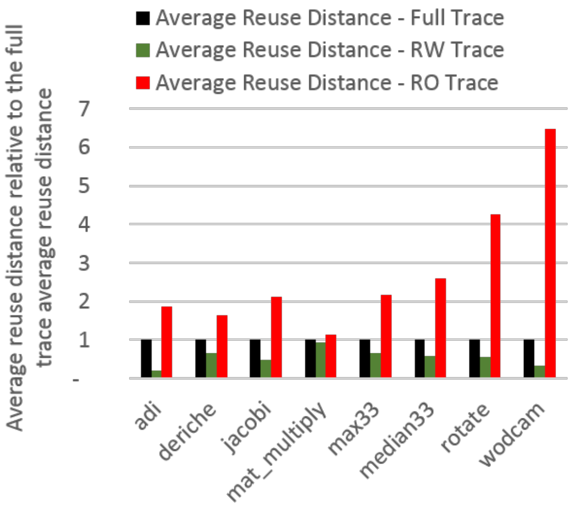


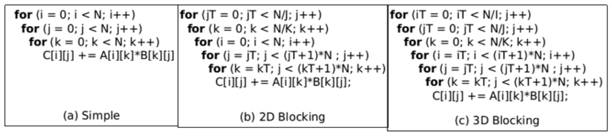
Figure 2: Variation of the average reuse distance. The read-only data present much longer reuse distances than average and separating them from the main memory stream increases the data locality of the memory stream

We study the separation of the memory stream with a data locality metric: the reuse distance [6]. This metric is defined as the number of different accesses between two accesses to the same address and depends only on the cache block size (fixed to 64B). A two-step analysis described 1 is followed: First, a memory trace is extracted from an instrumented execution of the analyzed application. Then, the read-only data and read-only periods are detected through an analysis of the memory trace. This offline analysis explores the trace in a 2D space (time and address) to detect rectangular areas solely composed of read accesses. Such analysis is manually tuned to detect read areas in an adequate grain in order to avoid capturing every read access as a small read-only area. The main memory trace is then split in read-only and in read-write traces. The average reuse distance is computed on all traces: The original trace and the 2 sub-traces. The results, shown in Fig. 2, are normalized to the average reuse distance of the full trace.

On all the evaluated benchmarks, two important observations can be made. First, average reuse distance of read-only data is 8.4 times bigger than the one of read-write data. Second, removing the read-only data from the main memory stream reduces the average reuse distance by 32% in average. The read-only data present much less data locality than average data and they can create potentially more pollution into the cache hierarchy. Removing them from the main memory stream flow therefore enhances data locality of the main memory flow and leads to better cache use. This conclusion has been extended to the standard set of benchmarks Mibench [OMITTED FOR BLIND REVIEW] and similar conclusions can be drawn for image and signal processing applications.

3.1 Sensitivity to code transformation

Read-only data detection and reuse distances depend heavily on parameters such as the degree of locality optimization applied to source code or input data sizes. As an example of locality optimization, the tiling is studied on the *matrix_multiply* benchmark. Fig. 3 shows 2D and 3D tiling transformation applied to *mat_mult* and different tile shapes are evaluated. The tile shape with the best result in terms of average reuse distance of the full trace is kept for the results of Fig. 4. In this case, tiles parameters are K=8,J=32 for 2D tiling, and K=8,J=16,I=1 for 3D tiling. It shows that even though the average reuse distance varies with the applied transformation, the average reuse distance between the sub-memory streams remains at the same level. This locality difference comes from the algorithm structure. The read-only matrix accessed in column-order, produces long reuse distances compared to the other, tiling can reduce this distance but will always be larger than the output matrix that reuses its data between each iteration of the inner loop. The conclusion drawn from the simple version of *matrix_multiply* stays valid with tiling transformation. However, some locality transformations such as loop fusion, can interfere with read-only data detection. For example, if a memory region is read in one loop body (creating a read-only period) and written in the other, merging the 2 loops would cancel the read-only period resulting in no detected read-only data.



```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  for (k = 0; k < N; k++)
    C[i][j] += A[i][k]*B[k][j]
```

(a) Simple

```
for (jT = 0; jT < N/J; j++)
 for (k = 0; k < N/K; k++)
  for (i = 0; i < N; i++)
   for (j = jT; j < (jT+1)*N ; j++)
    for (k = kT; j < (kT+1)*N; k++)
     C[i][j] += A[i][k]*B[k][j];
```

(b) 2D Blocking

```
for (iT = 0; iT < N/I; j++)
 for (jT = 0; jT < N/J; j++)
  for (k = 0; k < N/K; k++)
   for (i = iT; i < (iT+1)*N; i++)
    for (j = jT; j < (jT+1)*N ; j++)
     for (k = kT; j < (kT+1)*N; k++)
      C[i][j] += A[i][k]*B[k][j];
```

(c) 3D Blocking

Figure 3: Code of the different tiling strategies for *mat_mult*.

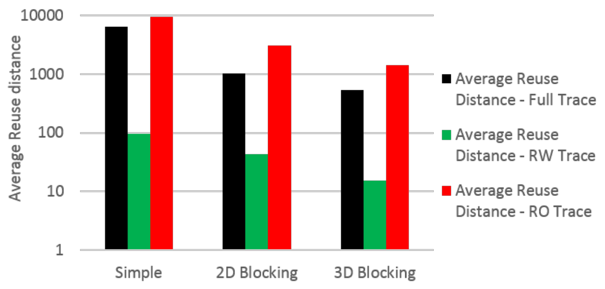


Figure 4: Evaluated tiling strategies for *mat_mult*. Even though the applied transformations affect the reuse distances, the difference between RO and RW reuse distances remains at the same level

3.2 Sensitivity to input data size

In another direction, the influence on the results of the input data sizes has been studied. The working set size that we study stays at best one order of magnitude under classic working set size used in embedded system. However, several measure points have been taken with different input data size and they are reported Table 1. The trend of the results suggests that increasing the working set sizes increases the difference of the reuse distance illustrated Fig. 2, reinforcing our conclusions. The illustrated difference of data locality between read-only and read-write data, motivates the idea of a specific management. The next section studies a data classification mechanism at compile-time in order to place the read-only data in the specific dedicated cache.

Table 1: Variation of the reuse distances relatively to input size averaged from all applications. The relative difference of reuse distances between RO and RW data grows with the input size of the data

	Avg Rd Full Trace	Avg Rd RW Trace	Avg Rd RO Trace
Small dataset	100%	60%	91%
Large dataset	100%	75%	291%

4 SOFTWARE SUPPORT

4.1 Compile-time RO/RW Data Classification

The first step of our hardware-software co-design solution is a compile-time static analysis to perform the RO/RW data classification. Our solution targets image processing applications which mainly use pointers, arrays and structures. We developed an extension of the alias analysis as an intra-procedural pass that focuses on nested computational loops. The algorithm is the following: for each loop nest, the pass iterates over inner loops bodies, and maintains a read-only and a read-write list of memory references. The results of alias analysis help us correlates the memory reference to memory regions. In order to consider a memory region as read-only, all references that points to the region must come from the read-only list. The status of memory regions can then be deduced for each loop nest and this property can vary between loop nests. The analysis uses a conservative approach in order to ensure that every reference captured is read-only. For example, if there are conditional branches in loops, the memory region needs to be read-only in all the paths in order to be considered read-only.

Using a RO/RW compile-time classification brings the important benefit of adding no cost on the hardware side and no execution overhead to perform the data classification. However, the compiler needs to be highly conservative regarding the classification and suffers from limitations such as pointer chasing, memory aliasing and other unknown events at compile-time which can restrict the scope of detected RO data. The main limitation in our situation remains the pointer aliasing analysis as within loop nest, it is important to distinguish pointers in order to detect RO/RW memory regions. To overcome this problem, some simple heuristics can be used and it has been shown that *may-alias* pointers rarely materialize into real aliasing at runtime [12]. So, we speculatively assume non-aliasing of *may-alias* pointers which is true in practice for our set of applications. Run-time checks can be added in the code when this assumption is made in order to keep correct analysis with other applications.

This algorithm has been implemented into GCC 4.9 and the accuracy of the method is compared to the offline analysis presented Section 3. Results in Fig. 5 show that our method is able to capture most of the read-only memory accessed compared to the offline analysis with a detection rate of 89.7% in average on the considered applications. The compiler analysis developed is sufficient for the evaluated benchmarks, however, the classification can be improved based on previous works ideas for shared and private data classification and information from different levels (OS, TLB, hardware, ...) can be gathered to improve the coverage of the classification.

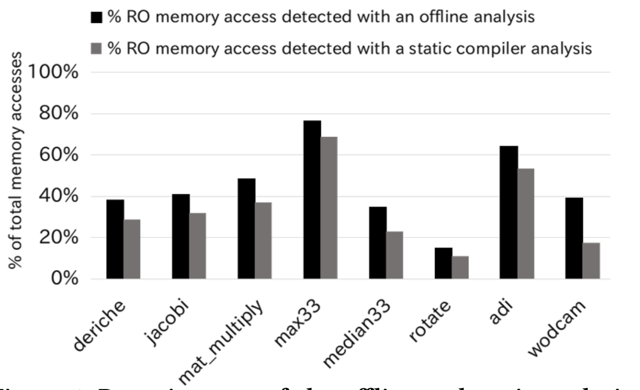


Figure 5: Detection rate of the offline and static analysis RO/RW classification method. The detection rate achieved by the static analysis almost equivalent to the (perfect) offline detection

4.2 Compiler-Hardware Interface

In our implementation, the compiler passes the hints to the hardware through setting one bits in the 64-bit instruction code. We assume this extra bit in each memory instruction that the compiler can use to drive the memory request to the correct cache structure. Existing architectures already use these kinds of extra bits in the instruction, such as the prefetch and evict-next instruction in the Alpha 21264. We believe that, in most architectures, the increasing speed gap between memory and processor will justify the inclusion of additional bits in the instruction code in order to improve caching.

The benefits of such interface are twofold: It allows to work on block granularity classification that is proven to be important for the detection rate [8] and it allows to transfer semantic information independent on the execution context. The information in this case is always accurate compared to prior works which exploit static information. Previous solutions use such interface to transmit locality information for data placement strategy [2] or dynamic cache replacement policy [11]. However, such hints can be inaccurate because it depends on dynamic behavior and it is hard to predict from the hardware point-of-view when the prediction made by the compiler is correct. In our proposition, since the classification is conservatively accurate, we do not need to plan for recovery mechanisms caused by classification errors.

5 HARDWARE SUPPORT

This section describes the hardware modification introduced for the dedicated RO/RW cache design, introduced Fig.7. The accesses identified as read-only by the compiler will be driven to the dedicated L1-RO cache structure.

5.1 Dedicated caches for RW and RO data

Our solution proposes a novel cache design that uses two separate caches to store the two types of data. Since the read-only property of the data is dynamic, data can be accessed by both caches depending on the execution's time. Starting from a generic memory hierarchy with two levels of caches, a design exploration has been performed regarding the level of the cache hierarchy where a dedicated read-only cache is relevant and the share/private property of this cache.

Some conclusions of this exploration are that adding a dedicated management logically at the second level next to the L2 cache increased prohibitively the number of access to the main memory as migration of cache blocks between the two caches have to go through the main memory. Also, a dedicated and private read-only cache next to the L1 data cache for each core show interesting results in terms of misses reduction for the L1 data cache but in this situation private read-only caches handle very few memory accesses and yet are not able to reduce read-only data misses which points at a poor use of cache ressources.

Based on those conclusion, the architecture that we propose adds a shared read-only data (L1-RO) cache which is shared across all cores next to private L1 data caches (L1-RW) for each core. The accesses identified as read-only by the compiler will be driven to the dedicated L1-RO cache structure.

5.2 Coherence Optimization

As the static analysis guarantees that the L1-RO cache issues only read requests, no tracking of the L1-RO blocks is then necessary from the coherence protocol point-of-view in the same way as instructions blocks. It opens up a degree of freedom to simplify transactions of the L1-RO cache. This idea directly relates to the related works about coherence optimization in previous sections and is orthogonal to them. Figure 6 details the 2 main situations that benefit from this situation. The first case is upon every block eviction, a L1-RW cache needs to inform the L2 cache about this event even if the block is not modified, so that the L2 controller can update its tracker list. This property is required in order to keep the system inclusive. This precaution is not needed with the L1-RO cache and evictions can be performed silently resulting in less messages issued. There is no performance gain in this case, as those transactions are off the critical path in the L1-RW case.

The second example illustrates that L1-RO cache operations do not change coherence state of others caches. Usually, a read request of a cache block that is already modified in another L1-RW cache creates a costly downgrade transaction of the cache block from *Modified* to *Shared* and the modified value of the data needs to be written back the L2 cache first. As blocks allocated in the L1-RO will never require to have write access the block, the downgrade is not necessary resulting in less messages and no modification of the coherence state of the L1-RW and L2 cache blocks .

6 EVALUATION METHODOLOGY

6.1 Benchmarks

All benchmarks are presented in Table 2. They come from in-house set of benchmarks and are written in C, parallelized with OpenMP and compiled with the presented static analysis and 03 optimization. Most of them are taken from image processing domain. The Fig.8 shows the memory accesses distribution for the benchmarks. The results are produced according to the compile-time RO/RW classification scheme presented Section 4.

6.2 L1-RO Cache Design

The design of the L1-RO cache is set to 32KB 2-way, it has been determined based on a design exploration study and optimized for energy consumption. Also, the L1-RO cache is not multi-ported because

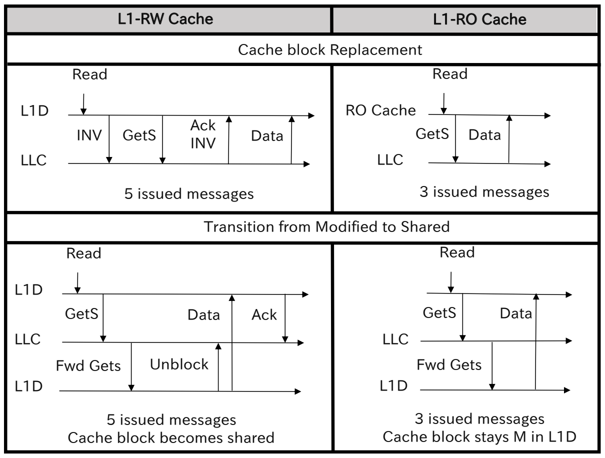


Figure 6: Illustration of the simplification of the L1-RO cache transactions to remove L1-RO blocks tracking for block replacement and downgrading operations. It reduces the number of required messages and do not modify the L1-RW blocks states.

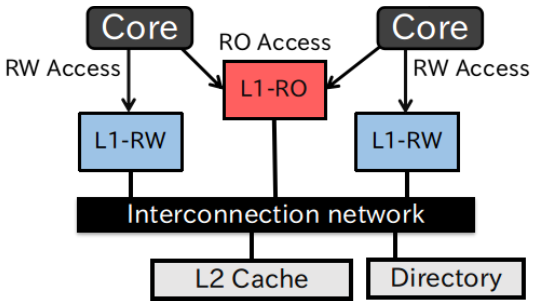


Figure 7: Dedicated RO/RW cache design with the L1-RO that handles requests performed by the RO data identified by the compiler

this increases prohibitively the cost per access and the requests coming from different cores to the L1-RO cache are sequentialized. and a processor may has to wait for other processors to send their request before being able to accessed the L1-RO cache. Note that all the caches (including the L1-RO cache) are non-blocking caches with MSHR. Other requests can be served while a miss is pending, so it limits the stalled time on the L1-RO cache.

6.3 Simulation

The full architectural simulation has been implemented into the Gem5 simulator [3] and the sub-simulator Ruby is used to describe the different components of the memory system and the coherence protocol. The L1-RO cache controller and the directory controller have been modified to remove the tracking of the L1âĹŠRO cache blocks. The x86 instruction set has been modified to include the cache hints included by the compiler on memory requests. The energy consumption of caches and interconnection network are obtained from the McPat framework v1.3 [17].

Table 2: Benchmarks description

Benchmarks	Input Data	Description
matrix_multiply	512x512 arrays	Blocked version of matrices multiplication
deriche	image of 1024x1024px	Canny edge detector
rotate	image of 1024x1024px	Image Rotation algorithm
max33	image of 1024x1024px	Image blur algorithm
median33	image of 1024x1024px	Image median algorithm
jacobi	1024x1024 matrices	Linear equation system solver
adi	256 elements arrays (10 iterations)	non-linear differential equations algorithm
wodcam	1000 test images (168x192px)	Recognition faces application based on eigenface method

Table 3: Simulation parameters of the different scenarios

	L1-RW Cache	L1-RO Cache
CPU	4 cores in-order, 1GHz, 1 load/store unit	
Baseline	32kB/64kB 2-ways	-
RW/RO Cache design	32kB 2-ways	32KB 2-ways
L2	256kB 16 ways assoc.	
Local Network	Crossbar , 64B wide, 2 cycles lat.	
DRAM	512MB, 1 memory channel at 12.8GB/s	

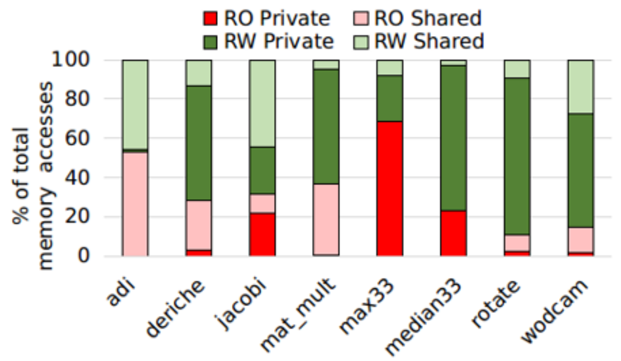


Figure 8: Memory access classification across benchmarks between RO/RW and Shared/Private (cache line granularity).

The simulation parameters used for the simulation are presented Table 3. The baseline to evaluate our solution is a standard MESI memory system with 32KB and 64KB L1-RW data caches.

7 RESULTS

The performance of the system is measured through the AMAT metric (Average Memory Access Time) because all other parameters (branch predictions, computation, ...) are constant between simulations. The energy consumption results are presented Fig. 9, the AMAT results Fig. 10 and the L1 misses repartition are shown in Fig. 11, for the different scenarios.

7.1 L1-Misses, Energy and Performance Evaluation

The RW/RO design presents a great cache miss reduction and the best energy improvement of 16,8% in average with a AMAT relatively constant compared to the baseline with 32kB L1-RW caches.

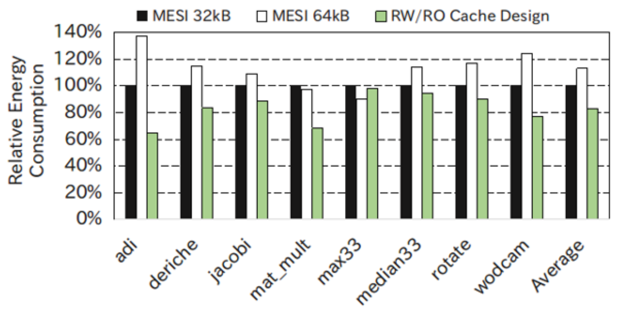


Figure 9: Normalized energy consumption. The RO/RW cache design brings the most energy efficient solution as it reduces the L1 misses with smaller caches than the 64KB L1-RW caches baseline.

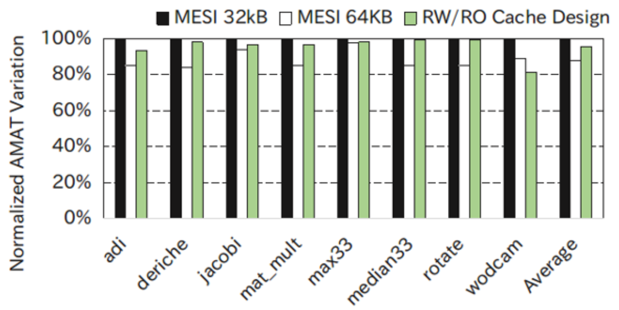


Figure 10: Normalized AMAT (less is better). The 64KB L1-RW cache design is the best design for performance as it benefits from larger caches. The RO/RW cache design is able to perform better than the 32KB baseline because of the L1 misses reduction it brought

Compared to the baseline with 64KB L1-RW caches , the RO/RW cache design does not achieved a miss reduction as important but is much more energy efficient for dynamic energy because they have a smaller cost per access and for the leakage power because the caches used are smaller.

As anticipated by the result of Section 3, the locality of the main memory stream is improved when the read-only data stream is removed resulting in misses reduction on the L1-RW caches. Some applications such as *adi*, *mat_multiply* and *wodcam* shared a significant part of their read-only datasets as seen in figure 8 and that allows to share blocks directly into the L1-RO cache, and decreases

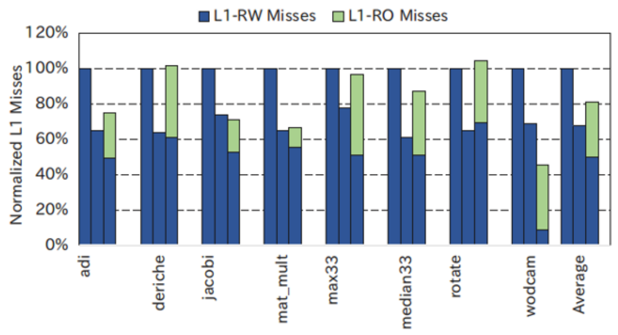


Figure 11: Normalized misses of the L1-RW and L1-RO caches. The RW/RO cache design allows to reduce L1 misses while using less cache storage

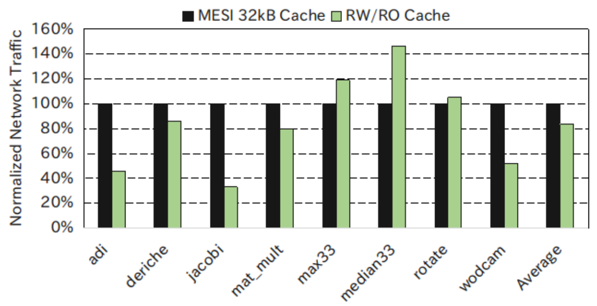


Figure 12: Normalized network traffic of the RO/RW cache design. The miss reduction and non tracking of L1-RO blocks allows to reduce the network traffic and mitigate coherence overhead

the reuse distance of read-only blocks by building constructive interferences. The L1-RO cache is shared across cores provoking resource conflicts that can lead to performance degradation. However, this effect stays marginal here because the L1-RO cache handles only 28.5% of the accesses and is also compensated by the significant reduction of L1 misses brought by the solution.

7.2 Network Traffic Evaluation

The figure 12 show the normalized number of flits sent along the interconnection network. The RO/RW cache design allows to reduce the network traffic of 26% in average compared to the baseline, decreasing the associated energy consumption. This effect rises because of the L1 miss reduction which requires less accesses to the L2 cache and the non tracking of L1-RO blocks by the directory which simplify cache transactions. On the evaluated applications, only *adi* presents an important amount of Modified->Shared transitions on L1-RW caches that really benefit from the situation described in the latter case of Fig. 6. For *adi*, cache efficiency is improved and significant part of the AMAT and energy consumption gain can be found for this reason.

7.3 Impact of the L1-RO cache latency

During the previous simulations, the L1-RO cache is configured to have the same latency as L1-RW caches because they are logically on the same level. However, the L1-RO cache latency is expected to be higher as it is shared across several cores and the latency

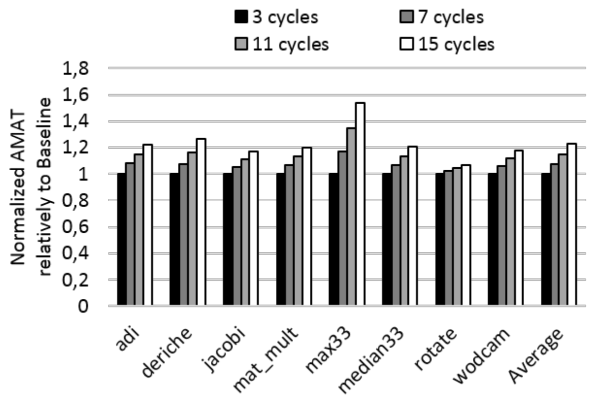


Figure 13: Sensitivity of the AMAT relatively to the L1-RO cache latency. The performance degradation of the system becomes significant for pessimistic scenario for the L1-RO cache latency

of the L1-RO cache is more critical as it can potentially stall all processors. Some additional experiments have been made in order to study the impact of the latency of the L1-RO cache on the overall system performance. This latency have been varied from 3 cycles to 15 cycles and the AMAT of the system compared to the baseline architecture (L1-RW cache of 32kB) is shown Fig. 13[1]. For a latency of 5 cycles, there is no performance degradation compared to the baseline architecture and the 5% degradation barrier is crossed with a RO latency of 9 cycles. The sensibility to this parameter can be explained mostly by the numbers of requests handled by the L1-RO. *max33* is more sensitive than *rotate* to this parameter because the proportion of total CPU fetches handled through the L1-RO cache is respectively 68% and 11% as seen in figure 8. These results show that there is no significant performance degradation even for pessimistic L1-RO cache latencies. Those results are to be put into perspective with the solution of Cebrian et al. [5] that proposes a similar cache design for shared data. The major flaw that they encounter is the latency to access the shared cache which is logically shared but physically distributed. Depending on which cache bank the shared data is stored, the request takes longer time to proceed which adds a prohibitive overhead of latency. Our simulations are less sensitive to the L1-RO cache latency parameter because the amount of access handled by our specialized L1-RO cache in our scenario is much less important than in their scenario (28.5% in our case compared to more than 50%), and also because the design is evaluated with a cluster of a smaller number of cores.

7.4 Scalability of the solution

We study the scalability of the memory system. Fig. 14 shows the average AMAT for the two datapaths, normalized to the baseline unicore solution. As the number of cores increases, bank conflicts, contention on shared resources (bandwidth and caches) and coherence protocol may impact performance of any of the datapath (RO and RW). Fig. 14 shows that the response time of the RW data path is essentially impacted by these destructive interferences. On the contrary, for the RO datapath, the AMAT stays relatively stable because increasing the number of cores increases the cache

[1]The latency of an L1-RW-cache is 3 cycles, for the L2 cache, 15 cycles

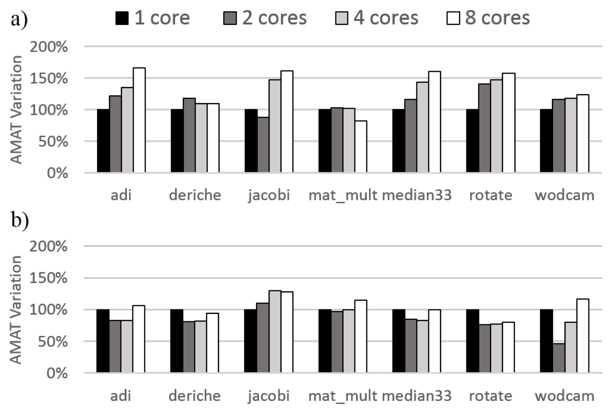


Figure 14: For scenario B, AMAT variation of a) the RW datapath (L1-RW) and b) RO datapath (L1-R0) normalized to the unicore baseline

sharing effect, outbalancing the previous negative impacts. As the applications are parallelized with OpenMP and threads run on identical processors, threads run similar codes at the same IPC, so they tend to work on shared data at the same moment of the execution allowing reuse across processors into the L1-RO cache.

8 CONCLUSION

A way to improve the energy efficiency of cache-based memory system is to propose heterogeneous management of data. Du to its potential for data locality and coherence optimization, this paper evaluates the specific management of read-only data as a hardware-software co-design with a dedicated L1-RO cache design. The RW/RO data classification is performed at compile-time with an efficient and light-weight static analysis that achieves a read-only data detection rate of 89.3% without adding hardware overhead. Our hardware design adds a dedicated shared L1-RO cache, next to regular L1 data caches. The solution improves the overall data locality by exploiting the difference of locality between read-only and read-write data and also by building constructive interferences in the L1-RO cache that reduces reuse distances and increase cache reuse. The solution reduces the coherence overhead by removing the tracking of the read-only cache blocks from the directory. Overall, the proposed architecture yields an energy consumption reduction of 16.7% and reduce the network traffic by 16.7%. on average without a performance penalty on a set of multi-threaded image processing benchmarks.

REFERENCES

[1] Oluwayomi Adamo, Affrin Naz, Kavi Janjusic, Tommy ans Krishna, and Chung-Ping Chung. 2009. Smaller split L-1 data caches for multi-core processing systems. *ISPAN, International Symposium on Pervasive Systems, Algorithms, and Networks* (2009).

[2] Kristof Beyls and Erik H. D'Hollander. 2005. Generating Cache Hints for Improved Program Efficiency. *J. Syst. Archit.* 51, 4 (April 2005), 223–250. https://doi.org/10.1016/j.sysarc.2004.09.004

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, and Rathijit Sen. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* (2011), 7. https://doi.org/10.1145/2024716.2024718

[4] S. c. Kang, C. Nicopoulos, H. Lee, and J. Kim. 2011. A High-Performance and Energy-Efficient Virtually Tagged Stack Cache Architecture for Multi-core Environments. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on.*

[5] Juan M Cebrián, Ricardo Fernández-Pascual, Alexandra Jimborean, Manuel E Acacio, and Alberto Ros. 2017. A dedicated private-shared cache design for scalable multiprocessors. *Concurrency and Computation: Practice and Experience* 29, 2 (2017).

[6] Edward G. Coffman, Jr. and Peter J. Denning. 1973. *Operating Systems Theory.* Prentice Hall Professional Technical Reference.

[7] Blas Cuesta, Alberto Ros, Maria E. Gomez, and Antonio Robles. 2013. Increasing the Effectiveness of Directory Caches by Avoiding the Tracking of Noncoherent Memory Blocks. *IEEE Trans. Comput.* 3 (March 2013), 14. https://doi.org/10.1109/TC.2011.241

[8] Mahdad Davari, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. 2015. The Effects of Granularity and Adaptivity on Private/Shared Classification for Coherence. *ACM Trans. Archit. Code Optim.* 12, 3, Article 26 (Aug. 2015), 21 pages.

[9] Michael J. Geiger, Sally A. Mckee, and Gary S. Tyson. 2005. Beyond basic region caching: Specializing cache structures for high performance and energy conservation. In *In Proc. 1st International Conference on High Performance Embedded Architectures and Compilers.* 70–75.

[10] Antonio González, Carlos Aliagas, and Mateo Valero. 1995. A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality. In *Proceedings of the 9th International Conference on Supercomputing (ICS '95).* ACM, New York, NY, USA, 338–347. https://doi.org/10.1145/224538.224622

[11] Xiaoming Gu and Chen Ding. 2011. On the Theory and Potential of LRU-MRU Collaborative Cache Management. *SIGPLAN Not.* 46, 11 (June 2011), 43–54. https://doi.org/10.1145/2076022.1993485

[12] Brian Hackett and Alex Aiken. 2006. How is Aliasing Used in Systems Software?. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14).* ACM, New York, NY, USA, 69–80. https://doi.org/10.1145/1181775.1181785

[13] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. [n. d.]. Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches. *SIGARCH Comput. Archit. News* 37, 3 (June [n. d.]), 12.

[14] H. Hossain, S. Dwarkadas, and M. C. Huang. 2011. POPS: Coherence Protocol Optimization for Both Private and Shared Data. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on.* 45–55. https://doi.org/10.1109/PACT.2011.11

[15] Daehoon Kim, Jeongseob Ahn, Jaehong Kim, and Jaehyuk Huh. 2010. Subspace Snooping: Filtering Snoops with Operating System Support. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10).* ACM, New York, NY, USA, 111–122. https://doi.org/10.1145/1854273.1854292

[16] Hsien-Hsin S. Lee and Gary S. Tyson. 2000. Region-based Caching: An Energy-delay Efficient Memory Architecture for Embedded Processors. In *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '00).* ACM, New York, NY, USA, 120–127. https://doi.org/10.1145/354880.354898

[17] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42).* New York, NY, USA, 12. https://doi.org/10.1145/1669112.1669172

[18] Seth H Pugsley, Josef B Spjut, David W Nellans, and Rajeev Balasubramonian. 2010. SWEL: Hardware cache coherence protocols to map shared data onto shared caches. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques.*

[19] Alberto Ros and Stefanos Kaxiras. 2012. Complexity-effective Multicore Coherence. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12).* ACM, New York, NY, USA, 241–252. https://doi.org/10.1145/2370816.2370853

[20] Avinash Sodani and CAM Processor. 2011. Race to exascale: Opportunities and challenges.

[21] Gregory Vaumourin, Thomas Dombek, Alexandre Guerre, and Denis Barthou. 2014. Specific Read Only Data Management for Memory Hierarchy Optimization. In *Proceedings on the 4th Embedded Operating Systems Workshop.*