Coherence State Awareness in Way-Replacement Algorithms for Multicore Processors

Matheus Alcântara Souza[1], Henrique Cota de Freitas[1], Frédéric Pétrot[2]

[1] Pontifícia Universidade Católica de Minas Gerais,
Graduate Program in Informatics, CArT Lab,
Belo Horizonte, Brazil

[2] Univ. Grenoble Alpes, CNRS, Grenoble INP*, TIMA,
38000 Grenoble, France

matheus.alcantara@sga.pucminas.br, cota@pucminas.br,
frederic.petrot@univ-grenoble-alpes.fr

Abstract. *Due to their performance impact on program execution, cache replacement policies in set-associative caches have been studied in great depth. Currently, most general-purpose processors are multi-core, and among the very large corpus of research, and much to our surprise, we could not find any replacement policy that does actually take into account information relative to the sharing state of a cache way. Therefore, in this paper we propose to add, as a complement to the classical time-based related way-selection algorithms, an information relative to the sharing state and number of sharers of the ways. We propose several approaches to take this information into account, and our simulations show that LRU-based replacement policies can be slightly improved by them. Also, a much simpler policy, MRU, can be improved by our strategies, presenting up to* $3.5\times$ *more IPC than baseline, and up to 82% less cache misses.*

1. Introduction

Cache misses can be broadly classified in 3 categories [Hennessy and Patterson 2003]: (1) Compulsory misses, due to a first access to the data, (2) Capacity misses, because of the cache reduced size compared to the working set size, and (3) Conflict misses, due to the fact that even though all cache blocks may not be occupied, the function that maps a memory address to a cache set cannot target an empty block in this set. Reducing conflict misses can be done by increasing the associativity of the cache, *i.e.* having mapping functions targeting more blocks per set. Unfortunately, searching for the data then requires more hardware and more time to take the hit/miss decision. The solution used to allow resources/performance trade-offs is using set-associativity: a block may be placed anywhere in a set whose elements (called ways) can be searched in parallel. Searching a small number of ways is fast, but there may be many conflicts, whereas searching among many sets requires more time, but limits the conflicts.

We trace back the invention of set-associative caches to 1968 [Conti et al. 1968], and since then, way-replacement policies, *i.e.* choosing which way should be evicted, have been extensively studied. Already in 1978, Smith [Smith 1978] and Rao [Rao 1978]

*Institute of Engineering Univ. Grenoble Alpes.

independently published surveys and analysis of replacement algorithms. Since then, new replacement policies have been devised and many optimizations proposed to fit different hardware related constraints, such as area, power and number of ways.

The current computing systems are parallel in nature, and they contain several processors that share a single memory address space. For obvious performance and power efficiency reasons, these multicore systems include caches, raising a new concern: ensuring cache coherence. This leads to a new category of conflicts, coherence misses, due to a block having been evicted as a result of a coherence action. And this eviction may interfere with the local way-replacement policy.

In this paper, our goal is not to devise a new way-replacement algorithm, but to evaluate if the sharing state or any information relative to it, *e.g.* the number of sharers of a cache block, can be used to enhance the performance of existing way-replacement policies. To that aim, we propose several approaches to take the sharing state into account, and we evaluate our proposals on parallel kernels and applications from the SPLASH-2 benchmark [Woo et al. 1995] using the Sniper simulator [Carlson et al. 2014]. Given this, our contributions are the following:

- Introduction of way-replacement strategies based on the current coherence state of cache blocks, and based on its number of sharers,
- Comparative analysis of these strategies when combined with well-known replacement policies (LRU, MRU, NRU and Random).
- We provide experimental evidence that a less complex policy (MRU) can be improved with our strategies, to compete on equal terms with LRU, for instance.

The remainder of the paper is organized as follows. Section 2 presents the related work. Our proposal to take into account cache coherence state in way-replacement is presented in Section 3. Section 4 presents simulation results obtained on a state-of-the-art simulator for both our proposal and classical way-replacement algorithms. Finally, Section 5 summarizes the results and concludes the paper.

2. Related work

Choosing the best option for eviction is a challenge that has been addressed by numerous way-replacement policies. As it is impossible to know the future references of the program, by nature it is impossible to devise an optimal solution to that problem, hence the vast number of heuristics that have been proposed.

Simple algorithms do not take into account the way the blocks are accessed when choosing one for eviction. This is the case for *First In First Out (FIFO)* that evicts the block that first entered the set or *Last In First Out (LIFO)*, which takes the opposite stand, and evicts the last one. *Random* or *Round Robin* are even simpler policies. Using these algorithms can get counter-intuitive results: larger caches or higher associativities may lead to higher miss-rates [Bélády et al. 1969].

More sophisticated strategies care about the frequency of accesses or the moment accesses were made to a block. However, there is a disadvantage of hardware complexity which rapidly increases with the number of ways. For instance, *Least Recently Used (LRU)* maintains additional bits to track the entry which has not been used for the longest time. However, this comes at the cost of $\frac{k(k-1)}{2}$ bits per set for a k-way associative cache

for the most efficient encoding. The opposite idea is *Most Recently Used (MRU)*, in which the entries to be evicted are the recently used ones.

Although those strategies have been proposed a long time ago, new way-replacement policies are still under investigation. For instance, an adaptive policy that detect the application behavior was developed recently [Tada 2018]. The idea is to calculate a global fluctuation of priority values on each cache set. This information is then used to demote or promote a way in the set for eviction. The strategy presents a 0.37% IPC improvement over LRU, with cache miss rate reductions.

Another adaptive cache replacement policy select blocks by monitoring their reuse characteristics [Bang et al. 2018]. The authors categorize the blocks as reuse or non-reuse, and as dirty or clean. Then, a priority between those categories is defined to select blocks in such situations for eviction. The IPC is improved by 13% was compared to LRU, according to the authors.

There is also a claim that some correlation exists between the number of hits in a last level cache block and its reuse distance [Vakil-Ghahani et al. 2018]. Thus, the authors suggest a block selection strategy based on the number of hits, stored in a table, which evicts the block with the lowest value. They obtain up to 12.2% performance improvements over pseudo LRU.

Finally, a cache replacement policy that cares about application behavior was proposed in [Pai et al. 2018]. In that strategy, memory level parallelism and data-reuse behavior are taken into account. This information is used in functions as weights and costs, to determine the candidates for eviction. Results using simulated multicores showed to 23.8% IPC improvements over SRRIP [Jaleel et al. 2010] and ABRIP [Lathigara et al. 2015] replacement policies[1].

To the best of our knowledge, very few studies address replacement algorithms making use of the coherence state of a cache block. In fact, we found only two. The first one [Mounes-Toussi and Lilja 1998] evaluates the effect of prioritizing a cache coherence state over others when choosing blocks for eviction. The strategy maintains an MRU state information to be used during the way selection process. Assuming the cache implements the MESI cache coherence protocol, the authors statically select which way has the block to evict following the order 'Invalid', then 'Shared' and not MRU, 'Exclusive' and not MRU, 'Modified' and not MRU, and finally MRU. If more than one block in a set is candidate for replacement, a random strategy is applied to choose among them. Later on, another study evaluated a similar strategy, but this time using dynamic priorities [Agarwal 2015]. In this dynamic strategy, the priority coherence state is stored for each cache set, being updated on every block access. This state is called Most Recently Used State.

Both policies are relatively easy to implement. However, the studies reported satisfactory results only with a very small number of processors according to today's standards – they do not bring much improvement over the traditional and local LRU approach. Furthermore, none of them took into account the number of sharers a cache

[1]SRRIP and ABRIP are replacement policies that take into account streaming applications with mixed access patterns. Those strategies use the concepts of MRU and LRU to keep cache blocks which tend to be reused as MRU and not reusable blocks from streaming applications at LRU.

block has. Thus, there is an open challenge regarding the use of the coherence state and sharing of cache blocks to choose the best amongst the k for eviction in k-way set-associative caches. Our contribution focuses on this challenge. We design and implement a second chance approach that differs from the related work since, unlike our proposal, most of it does not care about the sharing information of cache blocks. In addition, the two initiatives that took into account this information evaluated their proposals only for a random replacement policy.

3. Coherence Aware Way-Replacement Policy

We consider two main approaches to prioritize the eviction of the block contained in one way over another. The first one consists of counting how many sharers a single cache block has and take this information into account when an eviction is needed. In the second one, basically the decision of evicting or not the cache block depends on its coherence state. This approach is subdivided into two strategies that differ in the choice of the coherence state to be checked, which can be either static or dynamic.

Although past works also implement state-based replacement algorithms [Mounes-Toussi and Lilja 1998, Agarwal 2015], they only evaluate it for random replacement algorithms. We improve their work by using the replacement policies that are known to give much better results than random. We modified four widely used replacement policies to evaluate our strategies: LRU, MRU, NRU and also RANDOM. The main idea is to let the way-replacement algorithm select the candidate for eviction, and then check if that candidate should or not be evicted. In essence, we implemented a second chance approach. In the next sections we detail the way-replacement policies we designed and implemented. For the purpose of this work, we use the 'Modified', 'Shared', 'Invalid' (MSI), cache coherence protocol.

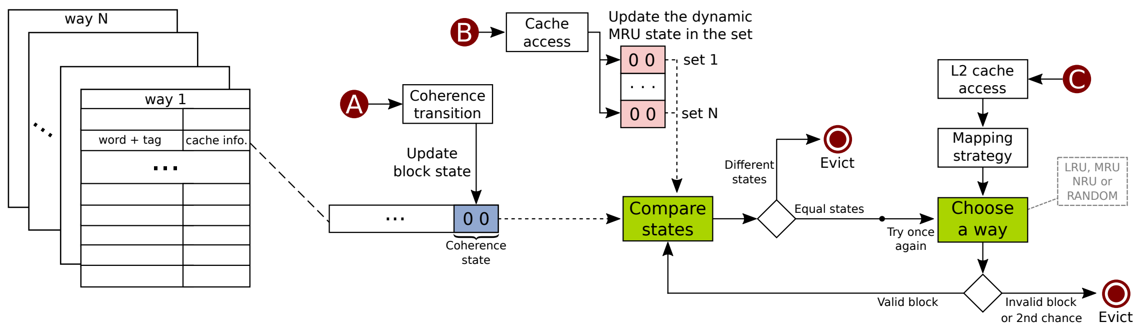


Figure 1. Overview of the coherence state based eviction

3.1. Coherence state based eviction

Our first proposal is to decide evicting or not a cache block based on its current coherence state. We have two approaches, dynamic and static, taking into account the 'Modified' and 'Shared' states. Figure 1 shows an overview of the dynamic strategy. The 'cache info.' contains information about the cache entry, such as the coherence state and the sharing set. Each cache block has its own coherence state, which is updated when there is a coherence transition, as in Ⓐ. We adopt the Most Recently Used State (MRU-State) approach. Thus, each cache set must know the state the most recently used block was,

either 'Modified' or 'Shared'. Additional bits are required per set, and they are updated when a cache access occurs on each set, as in B.

The replacement algorithm is performed after the cache set selection, accordingly to the set-associative mapping strategy in C. A candidate way is chosen, and if the block is valid, its coherence state is compared to the MRU-State. The replacement algorithm favors blocks that are not in the same state than the current MRU-State for eviction. If they are equal, the way selection is performed only one more time. In the static strategy, we set a global state that has priority for staying in the cache. This strategy is performed with 'Modified' and 'Shared' states. For instance, consider the 'Modified' state as priority. When the replacement algorithm choose a way whose block is in 'Modified' state, this block will not be evicted. Hence, another way will be selected in the modified replacement policy. For LRU, MRU and NRU, the new selected way is the second in the ordered list. For Random, simply another random way is selected.

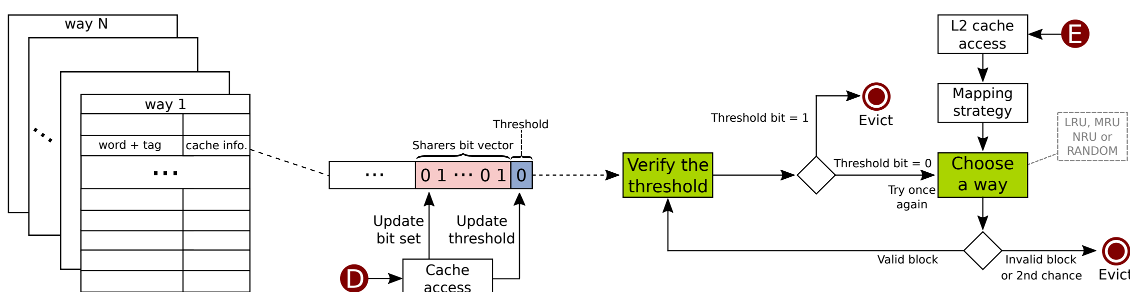


Figure 2. Overview of the sharer count based eviction strategy

3.2. Sharer count based eviction

The class of cache coherence protocol we used in our tests is the directory-based one. In directory-based protocols, additional meta-data is needed to keep track of the sharing set, *i.e.* the set of caches that cache a given block. In its initial form, the representation of the sharing set is done using a simple bit-vector. Figure 2 presents an overview of our strategy, and also depicts a single cache block and its sharers bit-vector. A 1 in position i in this bit-vector indicates that cache i caches this block. With that in mind, our approach simply tracks this bit-vector, counting how many sharers each cache block has.

We added a bit to each block, which is set to 1 when the number of sharers is higher than a threshold, otherwise 0. The bit-vector and the threshold bits are updated when there is a cache access, as in D in Figure 2. Thus, despite the 1-bit for each cache block, this approach adds new computational complexity to update the bit on every coherence transaction. For simplicity reasons, we set the threshold at 50%. However, the idea is that it can be predetermined by the architect, or even be stored, for instance, in a hardware register.

The replacement process is triggered after the mapping strategy in E. A candidate way is chosen, and if the block is valid, the algorithm checks the threshold bit. If set 1, the block is chosen and the data is evicted. Otherwise, a second try is performed, with a new candidate way.

4. Simulations and Result Evaluation

In this section we present the results that we obtained by evaluating our proposal using simulation. But first, we explain our methodology and detail the simulation environment.

4.1. Simulation environment and benchmarks

To perform our experiments, we used Sniper [Carlson et al. 2014], a multicore simulator. Sniper has integrated to its core the McPAT framework [Li et al. 2009], which we used to measure power consumption. We modified Sniper to set up our test architectures and way-replacement algorithms. Regarding McPAT, it uses the architecture parameters from Sniper, and simulations output to calculate the results. Thus, we did not modify the power model in the framework.

A 16-core processor based in a x86 Nehalem micro-architecture at 2.66 GHz was set. Some specific characteristics from this micro-architecture cannot be reproduced in Sniper though (e.g., Sniper only simulates an inclusive cache model). Furthermore, data prefetching is disabled in our simulations. The memory hierarchy has three levels. L1 has 32kB for data and 32kB for instructions, each one 4-way associative, private, with 4 cycles data access time, using LRU replacement policy. L2 has 2MB, is 8-way associative, shared by all 16 cores, with an access time of 8 cycles. We modified the policy on L2 during our experiments. The last level is L3, with 8MB size, 16-way, shared by 16, with 35 cycles of access time, and also using LRU. The aforementioned configuration is similar to a compute cluster in the manycore processor MPPA-256 [de Dinechin et al. 2013]. This processor comprises 16 of such clusters, each with 2 MB memory inside.

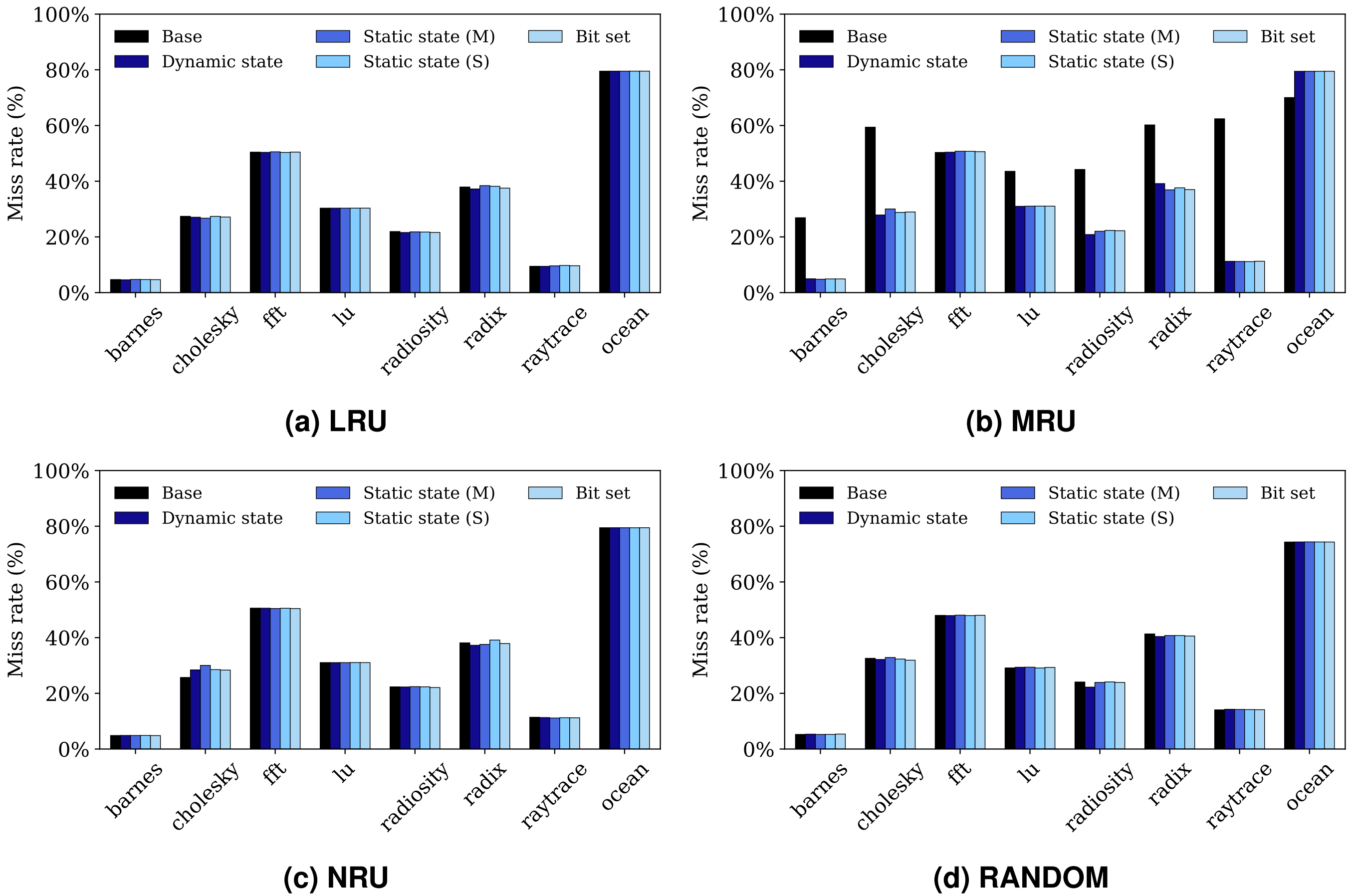
Table 1. SPLASH-2 workloads input sizes

Workload	Input	Workload	Input
barnes	32768 particles	radiosity	large room
cholesky	tk29.O	radix	1048576 integers
fft	4194304 data points	raytrace	car
lu	1024×1024 matrix	ocean	1026×1026 grid

We used 8 well-known parallel workloads from the SPLASH-2 benchmark [Woo et al. 1995]. Those workloads were run using all 16 cores. Table 1 shows the input sizes used for the workloads we selected. We chose to evaluate three metrics to understand the behavior of our cache replacement proposals: (i) L2 cache miss rate, (ii) instructions per cycle (IPC) and (iii) power consumption. In our set of simulations, we compared existing policies and our approaches. We present one chart per unmodified policy (*Base*) with corresponding modified versions. *Dynamic state*, *Modified state (M)* and *Static state (S)* represent the coherence based way-replacement strategies, as introduced in Section 3.1. *Bit set* is the approach we presented in Section 3.2.

4.2. L2 cache miss rate

In a first evaluation we look at the miss rate on L2 cache. Figure 3a shows the L2 cache miss rate for LRU. The *Dynamic state* approach presented equal or better values than *Base* for all applications. The best reduction was 2.20% in *barnes*. We keep in cache the least



(a) LRU **(b) MRU**

(c) NRU **(d) RANDOM**

Figure 3. L2 cache miss rates

recently used cache block that is in the current mostly used state. Thus, cache blocks in this state tend to be used in short periods of time, favoring this approach.

Static state approaches also presented cache miss rates slightly better than *Base* in some applications. The best cases with 2.48% and 0.89% less miss rates are *cholesky* and *radiosity*, respectively. However, this was not the case for *radix* and *raytrace*. Those are applications with working sets that might not fit in cache [Woo et al. 1995], presenting irregular access patterns and cache state behavior. Thus, evicting cache blocks based on static decisions is not a good approach. This *raytrace* behavior also led to an increase in cache miss rate for *Bit set*. However, other six applications have their cache misses slightly reduced when we use this heuristic. For instance, using LRU with *Bit set*, it was possible to obtain up to 1.51% less cache miss rate than *Base* in *radiosity*.

Figure 3b presents the L2 cache miss results when using MRU. At first glance, we can see high reductions in miss rate for some applications, in all approaches. In fact, by modifying the MRU policy, the mostly recently used data is kept in cache. Since this data tends to be used in short time, less misses were triggered. *Raytrace* and *barnes* were the ones with the highest relative reductions, around 82% when compared to *Base*.

It is worth noting that, if we compare the modified versions of MRU against the *Base* results from LRU, we can obtain similar results. They are not better than the

unmodified LRU, however, two observations must be taken into account. First, MRU is much simpler to implement than LRU. Second, the L1 cache replacement policy used in our experiments was the unmodified LRU. This led to a problem called local replacement hazard [Zahran 2007]. When an eviction is performed in L2, copies in L1 must be invalidated, further increasing the number of L2 cache accesses in the near future. This process is aggravated by the lack of synergy between cache levels, due to LRU blocks being evicted in L2 (MRU), but without care if they are MRU blocks in L1 (LRU).

Figure 3c presents the miss rates in L2 for the NRU-based algorithms. For most applications, low variation in rate is perceived in any approach. One exception is *cholesky*, that deals with factorization of sparse matrices. This leads to unstructured data in caches and NRU performs well in such situations. The NRU strategy evicts a not recently used block, rather than the one which is least used. If we look at the other applications, all of them were improved by using the *Bit set* strategy. Generally speaking, the worst strategy was the static one with shared state as priority. The best reduction was 2.31% for *radix* using the *Dynamic state* approach, when compared to *Base*.

The Random policy and its alternatives results are presented in Figure 3d. As the name says, the algorithm choose random blocks for eviction. In *cholesky*, *radiosity* and *radix*, the *Dynamic state* presented low improvements. This is due to a decrease in the randomness of the algorithm when the dynamic state condition was established. On the other hand, when modified replacement policies avoid to evict a block, they choose to evict another random block. Indeed, there are few changes in miss rates when this approach is used, regardless of the modified approaches.

To summarize the evaluation, we highlight that LRU using the *Dynamic state* can lead to better results in terms of L2 cache miss rates. This extends the results obtained in past works which used only the Random policy [Mounes-Toussi and Lilja 1998, Agarwal 2015]. Furthermore, if the architect needs simpler way-replacement solutions (e.g. with less hardware costs), a modified MRU version can be used over LRU.

4.3. Instructions per cycle

To understand the performance of our approaches, we measured the instruction per cycle (IPC) when using each of them. Figure 4a shows the IPC for all applications we tested with the LRU policies. Overall, IPC did not change much when we modified the strategy. The *Dynamic state* strategy appears as the best in seven of eight results, followed by *Bit set*. This was not the case for *radiosity*, which presented results slightly worst than the baseline. It was possible to obtain up to 6.8% more IPC with *cholesky* using *Dynamic state*. Still about *cholesky*, but using *Bit set*, more instructions were executed than in other strategies, so that it was worse than *Static state shared*. Generally, the number of cycles reduces when the cache miss is lower, even with more complexity added to the cache replacement policy. However, in some cases the reduction in cache miss rate was not sufficient to surpass this complexity (*fft, raytrace*).

Figure 4a shows the IPC bars for MRU based policies. The results are similar in most cases. Our strategies achieved up to $3.5\times$ more IPC (running *lu*). These results are an effect of the cache miss improvements we mentioned in last section. If we reduce the number of cache misses, the number of cache access also reduce, thus, less load and store cycles are spent. For *radiosity*, we remember that it has the smallest dataset among

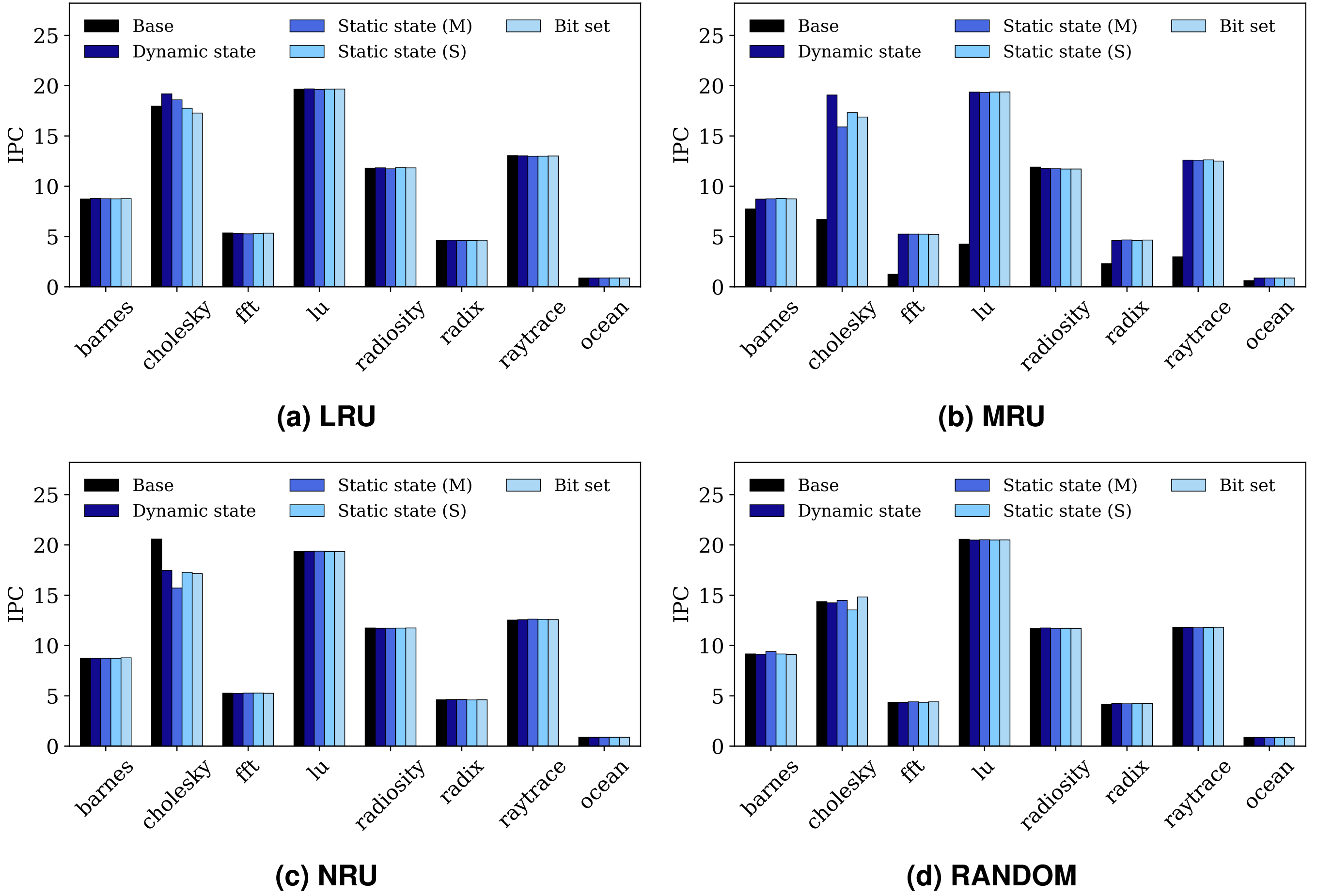


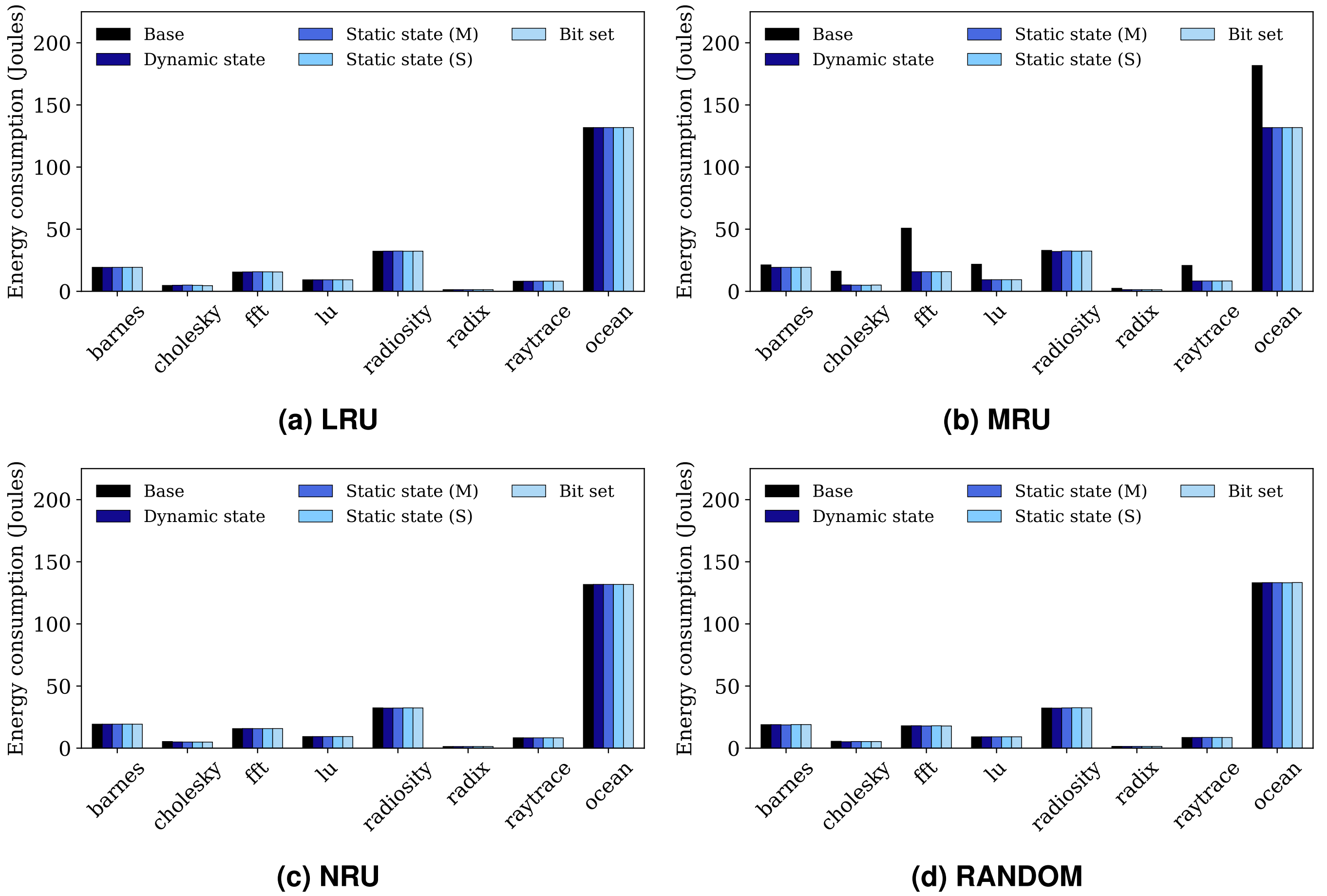
Figure 4. Instructions per cycle

SPLASH-2 application, and further the smallest traffic rate (bytes/instructions). That is, the cache miss reduction has less influence in IPC than in other applications.

The evaluations for NRU and Random policies with our approaches are similar to the others. The results for them are in Figures 4c and 4d. Basically, when we have less cache misses, there are less cycles being spent, and IPC increases. Specifically for *cholesky*, a weird behavior is presented with all strategies. Furthermore, *ocean* presented high cache miss values, and then poor IPC results. The used input size was not suitable for the simulated architecture. Thus, we should readjust the architecture to better fit *ocean*, or the application should be adapted. For those two applications, the use of cache coherence based way-replacement approaches was not good.

4.4. Energy consumption

In this section, we present the energy consumption results we obtained form McPAT and Sniper, measured in Joules. We present in Figure 5a the energy consumption when LRU-based replacement policies were used. Most of the results did not vary much, presenting less than 1% of variation when compared to *Base*. An exception was, again, *cholesky*, which consumed up to 5.7% more energy when *Modified static* was used. McPAT results showed that the power consumption (Watts) did not change much in all cases. However, there is a trade-off between the energy consumption reduction when



(a) LRU (b) MRU (c) NRU (d) RANDOM

Figure 5. Energy consumption

we have less cache misses, and the overhead cost of the cache coherence and bit-vector update.

The same analysis is presented in Figures 5c and 5d, which show the energy consumption for NRU and Random. For the other cases, most of them presented energy consumption results that do not changed much (less than 1%) when we used our way-replacement strategies. We also highlight the same observations from IPC evaluation for *ocean*, which thus presented energy consumption in comparison to other applications.

The MRU-based policies stand out again with expressive reductions in energy consumption. Figure 5b shows the results for this strategy. For *radiosity*, recall that it is not too much influenced by cache replacement alternatives, due to its lower traffic and smaller dataset than other applications. About 69% of energy consumption improvements can be obtained (the case for *cholesky* and *fft*). The *Static shared* approach was the best 3 times (*barnes*, *cholesky* and *raytrace*), tied with other approaches for *lu*, and with *Bit set* for *radix*. For *fft* and *radiosity*, the *Dynamic state* approach was the best. *Modified state* was the best for *ocean*, however, the energy consumption values were quite similar.

5. Conclusions

In this paper, we presented some way-replacement strategies for the eviction of blocks in cache memories. In spite of devising new temporal based algorithms, we chose to evaluate if the sharing state or the number of sharers of a cache block can be used to improve the overall performance and energy consumption. To perform this evaluation, we selected four well-known replacement policies: LRU, MRU, NRU and Random. We adapted those policies to consider the cache block state, making the decision to evict or not the block based on it. We also modified the policies to check if the cache block is shared by more than half of the processors.

Using the Sniper simulator and applications from SPLASH-2 benchmark, our results showed that such approaches should be considered in the next generation of memory architectures that rely on cache coherence. The most used way-replacement policy is LRU. Although our modifications in LRU achieved few performance gains, other cache related algorithms could be further evaluated to increase this improvements. Nevertheless, we highlight that using MRU with our strategies showed great results, being $3.5\times$ faster than the baseline, with cache miss reductions up to 82%. MRU is a less complex algorithm, thus the replacement policy should be simpler and faster than LRU-based ones. With those remarks, we conclude that the sharing state and the number of sharers of a cache block are information that can, somehow, be used to enhance the performance of cache way-replacement policies.

As future work, we intend to modify cache parameters, such as the number of ways, the cache size, and the cache sharing strategy, to verify the suitability of our approach in general. In the same way, to simulate more complex cache coherence protocols is a future work (e.g. MESI and MOESI). Exploring novel state-of-the-art way-replacement policies, modifying them, is a future goal as well. Furthermore, a fine tune in the architecture parameters is an interesting future work, to verify the one which fits better our strategy. Finally, we suggest to perform a deeper and more detailed power and area evaluation.

Acknowledgment

This study was financed in part by the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES)* - Finance Code 001. We also thank CNPq and FAPEMIG for their partial support.

References

Agarwal, T. K. (2015). Cache coherence state based replacement policies. Master's thesis.

Bang, D.-J., Kee, M.-K., Lim, H.-Y., and Park, G.-H. (2018). An adaptive cache replacement policy based on fine-grain reusability monitor. *IEICE Electronics Express*, 15(2):20171099.

Bélády, L. A., Nelson, R. A., and Shedler, G. S. (1969). An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM*, 12(6):349–353.

Carlson, T. E., Heirman, W., Eyerman, S., Hur, I., and Eeckhout, L. (2014). An evaluation of high-level mechanistic core models. *ACM Trans. on Architecture and Code Optimization (TACO)*.

Conti, C. J., Gibson, D. H., and Pitkowsky, S. H. (1968). Structural aspects of the system/360 model 85, i: General organization. *IBM Systems Journal*, 7(1):2–14.

de Dinechin, B. D., Ayrignac, R., Beaucamps, P. E., Couvert, P., Ganne, B., de Massas, P. G., Jacquet, F., Jones, S., Chaisemartin, N. M., Riss, F., and Strudel, T. (2013). A clustered manycore processor architecture for embedded and accelerated applications. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6.

Hennessy, J. L. and Patterson, D. A. (2003). *Computer architecture, a quantitative approach*, chapter 1.6, Quantitative Principles of Computer Design, pages 42–45. Morgan Kaufmann Publisher, Inc.

Jaleel, A., Theobald, K. B., Steely, Jr., S. C., and Emer, J. (2010). High performance cache replacement using re-reference interval prediction (RRIP). *SIGARCH Comput. Archit. News*, 38(3):60–71.

Lathigara, P., Balachandran, S., and Singh, V. (2015). Application behavior aware re-reference interval prediction for shared LLC. In *IEEE International Conference on Computer Design (ICCD)*, pages 172–179.

Li, S., Ahn, J. H., Strong, R. D., Brockman, J. B., Tullsen, D. M., and Jouppi, N. P. (2009). Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–480.

Mounes-Toussi, F. and Lilja, D. J. (1998). The effect of using state-based priority information in a shared-memory multiprocessor cache replacement policy. In *Proceedings. 1998 International Conference on Parallel Processing (Cat. No.98EX205)*, pages 217–224.

Pai, S., Singh, N., and Singh, V. (2018). AB-Aware: application behavior aware management of shared last level caches. In *Great Lakes Symposium on VLSI*, GLSVLSI '18, pages 237–242, New York, NY, USA. ACM.

Rao, G. S. (1978). Performance analysis of cache memories. *Journal of the ACM*, 25(3):378–395.

Smith, A. J. (1978). A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Trans. on Software Engineering*, (2):121–130.

Tada, J. (2018). A cache replacement policy with considering global fluctuations of priority values. In *International Symposium on Computing and Networking Workshops (CANDARW)*, pages 383–386. IEEE Computer Society.

Vakil-Ghahani, A., Mahdizadeh-Shahri, S., Lotfi-Namin, M., Bakhshalipour, M., Lotfi-Kamran, P., and Sarbazi-Azad, H. (2018). Cache replacement policy based on expected hit count. *IEEE Computer Architecture Letters*, 17(1):64–67.

Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. (1995). The SPLASH-2 programs: Characterization and methodological considerations. *SIGARCH Comput. Archit. News*, 23(2):24–36.

Zahran, M. (2007). Cache replacement policy revisited. *WDDD held in conjunction with ISCA*, pages 1–8.