Improving the Execution Time of Industrial Applications through Planned Cache Eviction Policy Selection

Sergio Arribas García, Giovani Gracioli, Denis Hoornaert, Tomasz Kloda, Marco Caccamo

Improving the Execution Time of Industrial Applications through Planned Cache Eviction Policy Selection

Sergio Arribas García[1], Giovani Gracioli[1], Denis Hoornaert[2], Tomasz Kloda[3], and Marco Caccamo[2]
[1]Federal University of Santa Catarina, Brazil, {arribas,giovani}@lisha.ufsc.br
[2]Technical University of Munich, Germany, {denis.hoornaert,mcaccamo}@tum.de
[3]LAAS-CNRS, Université de Toulouse, INSA, France, tkloda@laas.fr

***Abstract*—Modern industrial applications are demanding high computational power due to the evolution of features and components, such as real-time communication and control, image processing techniques, and security. In the microprocessor space, the high demand for computational power can only be satisfied by the integration of high-performance hardware components. Amongst them, caches and their replacement policies play a major role in preventing costly off-chip memory accesses.**

In this article, we argue that enabling caches with several eviction policies that can be selected at the software-layer and changed during an application execution can help substantially decrease its execution time (by increasing the cache hit ratio). To demonstrate this, we (1) present the implementation of an open-source cache simulation framework, (2) propose four distinct approaches to identify when to change the policy during the execution, and (3) assess the cache hit rate improvements brought. Experiments show that workloads running with the proposed approaches can feature up to 30% cache miss rate improvement in comparison to using the LRU replacement policy.

***Index Terms*—Cache memories, hit ratio, eviction policies**

I. Introduction

Since the creation of the term Industry 4.0, during the Hannover Fair in 2011, global manufacturing industries have been focusing on improving and digitalizing their production by introducing and using new technology (i.e., control through the internet, intelligent algorithms, image processing, etc). This constant evolution has prompted a demand for computational power that can only be sustained by microprocessors featuring various hardware modules aiming at speeding up the processing, such as cache memories.

Cache memories rely on the temporal and spatial locality of the memory accesses to improve the average execution time of applications. Since caches have limited space, the choice of which data should be kept in the cache during a time window is extremely important for system performance. A cache eviction policy is responsible for choosing a cache line to be replaced when there is a cache miss [1]. Different eviction policies are used by current commercial microprocessors, such as Least Recently Used (LRU) and its variants, First in First Out (FIFO), and Random. They offer different performance profiles w.r.t. the cache hit rate due to the target application's memory access pattern and the cache size [1]–[3].

Traditional microprocessors are usually designed with a single fixed cache eviction policy per cache level that is applied regardless of the target applications. Using a fixed eviction policy during the execution of a code is not a guarantee of best performance. As illustrated by Figure 1, an application presents fluctuating cache hits ratio performance throughout its execution depending on the eviction policy. This figure depicts the cache hit ratio (y-axis) for four different eviction policies (LRU, FIFO, RANDOM, and BIP[1]) and the number of executed instructions (x-axis) for the `pca-small` benchmark issued from the Cortex Benchmark Suite [4] running on top of Cachegrind (a cache profiler tool) [5]. Every point constitutes the percentage of cache hits in the last 1000 executed instructions. The benchmark's execution is divided into six segments (noted from a to f). Each segment is associated with the best-performing policy w.r.t. the hit-rate. For instance, in segment (c), the BIP is selected as it outperforms the other policies.

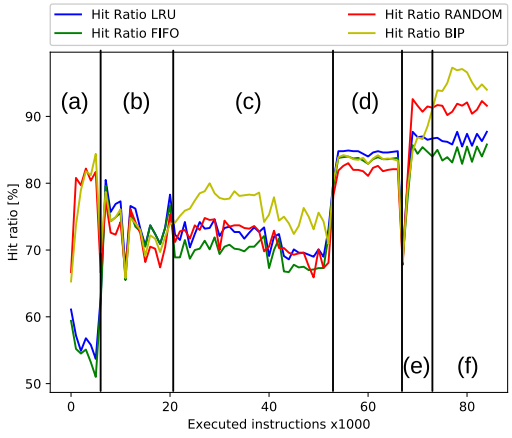


Fig. 1: Fragment of cache hit ratio for pca-small benchmark considering four cache eviction policies (LRU, FIFO, RANDOM, and BIP).

Different related works have studied the performance bene-

Giovani Gracioli and Sérgio Arribas García were supported by Fundação de Desenvolvimento da Pesquisa - Fundep Rota 2030/Linha V 27192.02.01/2020.09-00. Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research.

[1]BIP is described in the next section.

fits and impacts of varying the eviction policy [6]–[8]. Nevertheless, none of the previous studies have focused on finding the best eviction policy considering specific code sections. This process is not trivial as the memory access pattern and cache lines status must be considered. Moreover, having the ability to change the policy at run-time and adapt the cache as the process executes can improve its cache-hit rate and consequently, its performance. For instance, it would be possible to adjust the policy according to the factory evolution phases. As exemplified in Figure 1, it would be possible to achieve considerable performance gains by enabling and exploiting eviction policy change during the application's execution.

In this paper, we present a framework to improve the execution time of industrial applications by maximizing the cache hit ratio through the selection of cache eviction policies for specific code sections. We present four different techniques to analyze the code and assign cache eviction policies to code sections and implement the approaches in an open-source cache profiler framework. In summary, the contributions of this paper are: (i) We propose four different approaches to simulate compiled code and to select the best cache eviction policy for specific code sections; (ii) We implement the four proposed approaches as an extension of a cache profiler framework (Cachegrind [5]) and release it as an open-source artifact for future research on related topics; and (iii) Based on the framework implementation, we evaluate and compare the proposed approaches in terms of cache misses and execution times, using relevant benchmark applications. Our proposed techniques to select and assign cache eviction policies to code sections can reduce the cache misses of more than 30%.

II. Related Work

Cache memories are one of the elements that could impact the performance of industrial applications [9]. The capability of the system caches to keep hold of the adequate data and provide a high cache hit rate depends on many architectural aspects such as the number of sets, the number of ways, the write policy, and the hit latency. As highlighted by [10] and [1], amongst the state-of-the-art eviction policies such as Random, FIFO, and LRU, the latter provides the soundest resistance to "chaos" [10].

Unfortunately, in addition to requiring more local memory resources, the LRU policy suffers from one notable aspect: the minimal life span (abbreviated *mlp* in [1]), that is a function of the number of ways. In [6], Qureshi et al. proposed three new cache line policies derived from LRU. Referred to as *Adaptive Insertion Policies*, they differ by the way new cache lines are inserted in the cache set. The proposed LIP (LRU Insertion Policy) borrows all the precepts of LRU except that new caches line are inserted at the LRU position instead of the MRU (Most Recently Used) position. Alternatively, [6] proposed the BIP policy (Bimodal Insertion Policy), an extension of LIP where the insertion of each cache line at the LRU position instead of the MRU is decided with a probability $\epsilon \in [0, 1]$. Finally, they proposed a dynamically adapting policy called DIP (Dynamic Insertion Policy) that switches between the LRU and BIP policies. The decision to switch is taken by monitoring cache hits trends and selecting the policy incurring fewer misses.

The difficulty to assess the usefulness of a cache line at a given instant has prompted researchers to rely on artificial intelligence techniques. These approaches train their classifier using a sequence of eviction decisions generated by the optimal eviction policy (i.e., Belady's/Oracle's policy [11]). Tools such as Hawkeye [12] and Glider [13] train classifiers to guess whether a line is "cache-friendly" or "cache-averse" and base any eviction decision on this. These AI-based approaches, while successful at providing increased cache hit rates, are impractical to implement in real hardware as the network's depth drives the frequency down, and implementing extra memory, adders, and multipliers is expensive.

Table I summarizes the different approaches designed to improve the results of eviction policies as LRU. The DIP-based approaches [6] allow to use LRU or BIP as eviction policies and the selection is made dynamically online. DIP-Global implements two separated auxiliary tag directories for LRU and BIP and a counter that informs which policy has fewer misses in selecting the best policy. DIP-DSS (DIP Dynamic Set Sampling) reduces the hardware overhead of DIP-Global by using two ATDs with 32 sets instead. Although the authors claim that all DIP variants are practicable (in terms of hardware implementation), only the DIP-Set dueling option does not imply a big hardware overhead or extra ATDs. Hawkeye [12] and Glider [13] are both based on AI and have a considerable hardware overhead. They provide better results when compared with the DIP-based approaches. Our mechanism, listed in the last row, instruments the programs obtaining the best combination of eviction policies for specific code sections. The offline mechanisms return the lines of code where a policy change is triggered in the CPU. The online mechanism uses three standard policies and compares them through a set-dueling strategy.

III. Framework for Cache Eviction Policy Selection

In this section, (1) we present the assumptions and system model we considered to allow the usability of the proposed approaches and (2) we discuss the implementation of the approaches, creating a framework for cache eviction policy selection in Section III-B. We close the section by discussing relevant aspects of the proposed framework in Section III-C.

A. Assumptions and System Model

We consider a System-on-Chip (SoC) design featuring an arbitrary amount of cores. Each core must be associated with private Level-1 (L1) data and instruction caches. The cache hierarchy can be extended to an arbitrary number of levels before leading to the DRAM and can either be private, shared, or unified. We assume that the cores do not interfere with each other thanks to state-of-the-art isolation techniques (e.g., cache coloring) [14]. In addition, despite the memory level parallelism they offer, we do not consider non-blocking caches to keep the model simple and prevent predictability issues [15].

TABLE I: Related work overview.

Approach	Practicable	Overhead	Online/Offline	Technique/Mechanism
DIP-Global	Yes	2 Tag directories	Online	Parallel execution and comparison (Hits Counter) [6]
DIP-DSS	Yes	2kB (2 ATDs)	Online	Parallel execution and comparison of some sets (Counter) [6]
DIP-Set dueling	Yes	15 bits	Online	Dedicates some sets to each policy (Counter) [6]
Hawkeye	Yes	28kB	Online	AI, predictor based in optimal policy for previous accesses [12]
Glider	Yes	62kB	Both	AI, predictor based in sequence of last PC + offline training [13]
Belady	No	-	Offline	Optimal solution, based in future memory accesses [11]
Proposed Work	Yes	15 bits, Multiplexer	Offline or online	Detection of best policy for section of code

The main memory model does not take into account typical DRAM architecture and memory transaction arbitration under saturation. These assumptions ease the modelization of the platform and, most importantly, help emphasize the benefits of the proposed approaches by removing any source of noise.

We assume there is a cache architecture capable of (1) enforcing different eviction policies following the software-layer/end-user directive and (2) providing a strict partition for every task of a given core in order to eliminate cache-related costs during context switches. The second point requires the cache to be informed of the currently running application. This can be implemented in many ways such as via memory-mapped registers or dedicated assembly instructions. While the proposed cache architecture can be implemented at any level of the cache hierarchy, this article focuses on the L1 cache. A sound analysis of the interplay between the cache level eviction policies is considered out-of-scope for this paper and left as future work.

Regarding the application model, it is up to the system engineering to define the best policy for each task or application, taking into consideration the criticality levels of the system. For instance, if a time-sensitive task is assigned to a cache partition, then predictable and analyzable eviction policies, such as LRU, must be considered [1]. If a task or application demands only performance, then any eviction policy could be applied. In this paper, we focus on the optimization of the average execution time by selecting and changing the cache eviction policy during code execution.

B. Proposed Approaches

Even with automated mechanisms, identifying the right places to choose a new eviction policy in the code is not trivial. The optimal selection should consider the current status of the cache memories and the next memory addresses accessed by the software. In this paper, we consider offline and online methods to select the eviction policy. Table II presents the main features of the proposed approaches. All approaches implement an automated policy selection mechanism and requires extra memory space.

TABLE II: Overview of the approaches features.

Approach	Execution	Policy selection
Fixed-Window	Offline	Window Size
Sliding-Window	Offline	Window Size and threshold
Dip+Aging	Online	Counter
Set Dueling	Online	Counter

1) Fixed-Window Approach: In this approach 4 eviction policies are simultaneously applied to 4 copies of the cache memory (LRU, FIFO, BIP and RANDOM). During a time window, all data accesses are applied to each cache individually. Upon a cache hit, a counter associated to that cache (and indirectly its eviction policy) is incremented, effectively denoting the hit ratio within that window.At each instant, these hit counters are compared for the current window. The framework selects the policy with the greatest performance for the windows and generates an output that will make the CPU to switch to that policy in real hardware. After every window the framework copies the cache of the selected policy to the other cache policies, starting after that a new cycle.

The size of the window is the only parameter the developer must set and tweak. This represents the biggest disadvantage of this method as this parameter directly impacts the quality of the eviction decisions and it cannot be changed during the offline process execution. While coarser granularity may include code blocks that could work better with another policy different from the policy selected for the whole window, small window sizes are likely to result in many policy changes.

2) Sliding-Window Approach: Similarly, the sliding-window approach also uses dedicated caches for each policy and defines a window size. However, the window defines the number of previous memory accesses used to calculate the average of cache hits. The code is executed simultaneously using all the eviction policies and different copies of the cache for each policy, in every single memory access the hit rate is calculated. The algorithm in the framework selects the best hit rate among the policies after comparing all the cache copies and select the best policy. The output of the approach are the code lines where the best eviction policy switches. In contrast to the fixed-window approach, the sliding-window approach can change the eviction policy with a higher frequency. This could be disadvantageous because it can cause an eviction policy change in every single data access. To avoid this effect a threshold is added to the algorithm, this parameter fixes the minimum difference in the hit ratio between policies before to select a new eviction policy.

3) DIP+Aging: In this approach, 3 policies are used (LRU, FIFO and BIP) and each policy has its cache with only 32 sets, as suggested in [6] for the DIP Set Dueling. The framework assigns a counter attached to each cache (similar to the DIP-Global policy proposed in [6]). The counter is defined with a maximal number of bits. During the program execution, each policy modifies its cache simultaneously according to the

executed instruction. In every cache access a hit increments the counter by one. After that, the framework shifts the counter one bit to the left and compares all the counters. The difference between our approach and DIP [6] is that DIP is limited to only two policies, whereas our online selection method can use more than two. This minor modification, however, completely changes the way the counter is used. For instance, DIP has a unique counter that is shared by the two policies. Here, we have a dedicated counter per policy. DIP uses the medium value of the counter (for example, 512 when the counter max is 1024), while our method starts from zero and considers aging of hits. The eviction policy is selected by taking the option with the greatest counter.

4) Set Dueling: This approach is based on DIP+aging. Here 3 policies are also used (LRU, FIFO and BIP) but they do not have a separate cache memory. This method uses a set dueling technique, as proposed in [6]). For each policy 32 sets of the cache memory are selected. These sets use its own eviction policy and have a counter with the same strategy as in the DIP+aging. The sets of the cache memory that were not assigned to a policy are used by the policy with the greatest counter value.

Set Dueling avoids the overhead of having 32 extra sets per policy and the need for updating the counters in every single memory access. The advantage of these two last approaches lies in the simplicity of their implementation and the minor computing needed. The determination of the eviction policy can be done offline (through simulation) as well as online in the CPU by having a per-policy counter.

Framework Implementation. We implement the described approaches in a framework to select the best cache eviction policy (the one that maximizes the cache hit ratio). We extended Cachegrind, one of the tools composing the instrumentation framework Valgrind [5]. Our framework with the modified Valgrind/Cachegrind code is available online.[2]

C. Important Remarks

Performance. The time overhead added to run the proposed framework is not big, existing only a difference between the online and offline approaches. For instance, the average execution time of the offline approaches (Fixed-Window and Sliding-Window) for the pca-small benchmark (presented in the next section) is the same, but it is 60% bigger than the execution time of the online approach (3 minutes more in this case). In benchmarks with more cache accesses, we state that the differences are comparable.

Framework output. Independently of the selected approach, the framework generates an output informing a code line for every policy change. The information contained in the line indicates to the developer the source file, the function, the code line number, and the moment for the insertion of the function call which generates the policy change in the processor. The automation of the adaptations and compilation of the source code after the outcome of the framework is out of the scope of this paper and has been left for future work. Future work concerns the integration of the framework with a C compiler that generates the compiled code including the policy selection.

Program inputs. During the offline process we propose to execute the code in the framework varying the inputs and merging the results. Comparing the outcomes, it is possible to identify parts of the code where the same policy could be used with multiple inputs. In code sections where the outcomes are divergent, the developer is responsible to choose the more convenient policy manually. In our experiments, we compared the outcomes of the same benchmarks with different inputs and we detected that they affect the improvements obtained with a dynamic eviction policy. For instance, we detected a difference of up to 8% in the execution time when the PCA benchmark was executed using the medium input in comparison with the small input. The implementation of a tool to compare the results will be part of future work.

Loops in code. To allow a policy change within loop statements is challenging. The framework indicates the specific iteration of the loop when the policy switches. However, the implementation in the hardware implies the creation of extra counters in the CPU. An approach to solve this situation could be processing the looping as a unique block and returning the best policy for the entire block.

Additional cache space. Supporting more than one cache eviction policy increases the required space in the integrated circuit (i.e., space that would be allocated to the cache is reduced due to extra logic to support the eviction policies, at least a multiplexer to select the chosen policy, and extra metadata). However, eviction policies such as LRU, FIFO, and BIP, which keep the information of each line that has been recently accessed, can share the same structure to store the metadata, and thus reduce the additional space. RANDOM, for instance, does not require any metadata.

IV. Evaluation

To evaluate the proposed framework with the four approaches, we used a representative set of benchmarks from the San Diego CortexSuite, which is a benchmark suite that provides algorithms from machine learning, natural language processing, and computer vision areas and includes real datasets for each algorithm [4]. We use the Principle Component Analysis (pca), Singular Value Decomposition (svd), Image Stitch, Latent Dirichlet Allocation (lda), and SIFT algorithms, representing important memory-intensive algorithms for machine learning, feature extraction, and image processing and also used in embedded systems, such as industrial applications. We also varied the input for each benchmark, using the standard inputs available in CortexSuite (`small`, `medium`, and `large`, or `cif`, `qcif`, and `hd` when the algorithm is related to image processing).

We executed each benchmark version in our modified Cachegrind framework varying the cache size[3] (4, 8, 16,

[2] https://github.com/donxergio/cachegrind.

[3] Cache size here means a cache partition size dedicated to a task.

32, and 64KB), the four proposed approaches, and the four standard cache eviction policies (LRU, FIFO, RANDOM, and BIP). For BIP, we set the bimodal probability parameter (ϵ) to 1/64 as also used in [6], [8]. The cache line size was defined as 32 bytes and the number of ways as 4. For the fixed-window, sliding-windows approach, DIP+Aging and Set Dueling (named FIX, SLIDE, DIP_3A, DUEL), we consider window sizes of 4096 cache data accesses. For the sliding-window approach, we consider a threshold of 128. The LRU policy serves as a baseline comparison, since it is the more frequently used policy in modern microprocessors.

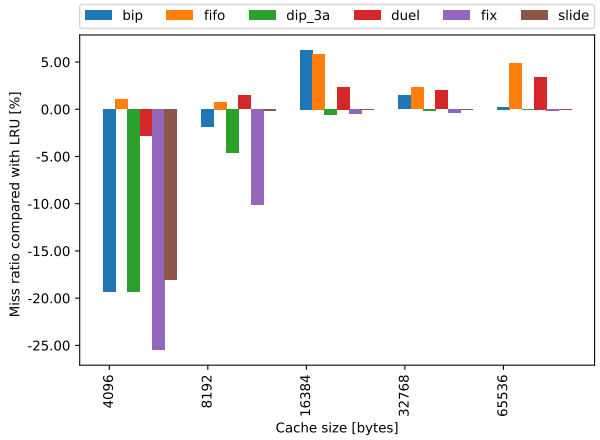
Based on the execution of the described experiments, we obtained the number of cache misses for each configuration. Figures 2 and 3 show the miss ratio for each policy and approach compared with the miss ratio of LRU for the same code. The calculation of the miss ratio uses the number of misses in the L1 data cache in relation to the number of read accesses in the L1 data cache. On the x-axis, we vary size of the cache. On the y-axis, we present the miss ratio of the policies compared with the miss ratio of LRU.

In Figure 2, for the svd3-small benchmark among the standard policies, BIP is the one with the best performance for up to 8 KB of memory, after that LRU performs better. All the proposed approaches, except Set Dueling, perform better that the traditional policies. The fixed-window (FIX) method is 25.5% better than LRU and 7.7% better than BIP for cache memories of 4 KB, while DIP_3A performs similar to BIP for 4 KB but better than it for the other cache sizes. The sliding-window (SLIDE), on the other hand, performs close to BIP for 4 KB and close to LRU for other memory configurations. For the lda-small benchmark the behaviour of the approaches follows the one observed in svd3-small. The fixed-window performs in all configurations better than LRU but not as good as BIP for 16 KB. The DIP_3A follows the performance of BIP (18% better than LRU) up to 16 KB, and after that memory size performs better than BIP.

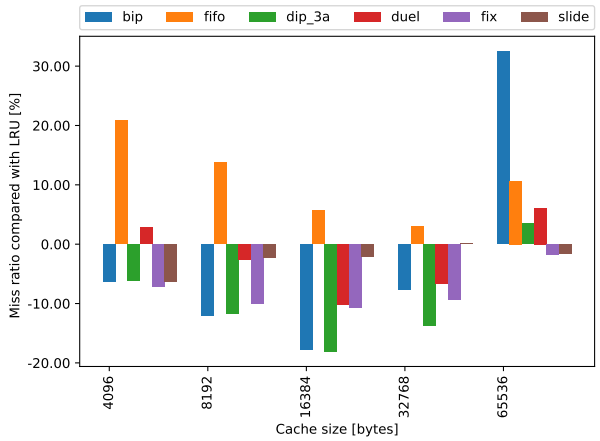
Figure 3 shows the results obtained in the experiments with the benchmarks (a) pca-small and (b) spc-small. For pca-small all the presented approaches perform better than LRU, having the greatest difference for a cache size of 64 KB, where DIP_3A, FIX and SLIDE are more than 30% better than LRU, Set dueling follows the results but its results are worse. For the spc-small benchmark only the fixed-window and sliding-windows approaches perform better than the traditional policies, in the case of the sliding-windows it replicates the LRU policy, which is the traditional policy with minimal miss rate, for this reason the difference with LRU is zero. FIX yields better results for larger caches with up to 2.2% improvement for 64 KB wide cache compared to LRU. For all the experiments, SLIDE emulates LRU's performance. In contrast, BIP performs poorly. As described in [16], this can be credited to the benchmark's memory access patterns and its working-set. We expect that Set Dueling performs worse than DIP_3A, because it uses part of the cache memory (32 sets per policy) instead of dedicated caches for every eviction policy. This makes the approach very dependent on the memory access pattern. With small cache sizes the difference between them should be more evident. In constrast, the fixed-window approach returns the smallest miss ratio, because it checks the four traditional policies and uses the best one as output.

As highlighted by Figure 3(a), A variation in the cache size also affects the behavior of the approaches and policies within the same benchmark. For instance, the pca-small application shows gains and variation until 64 KB while, for svd3-small, a loss can be observed until 64 KB. As expected, the larger the cache partition, the smaller the eviction policy impact is.

In summary, we have obtained a reduction between 5% and 30% with the fixed-window and DIP+aging approaches in most of the configurations. The framework allows to simulate the code with a cache memory architecture as the actual cache architecture of an industrial hardware and choose the approach to select the eviction policy leading to the highest performance. The reduction of cache misses improves the execution time and brings other benefits, such as a reduction of power consumption.

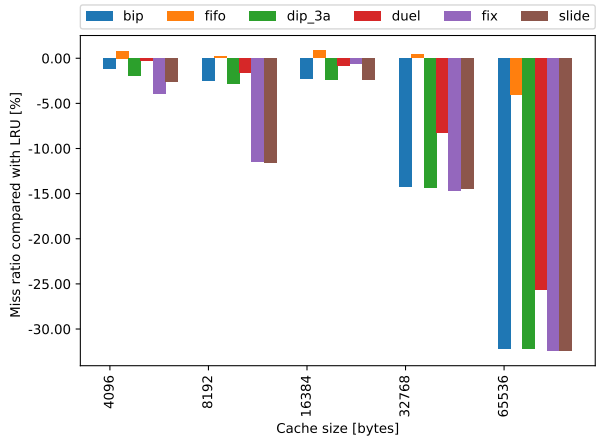


(a) svd3-small

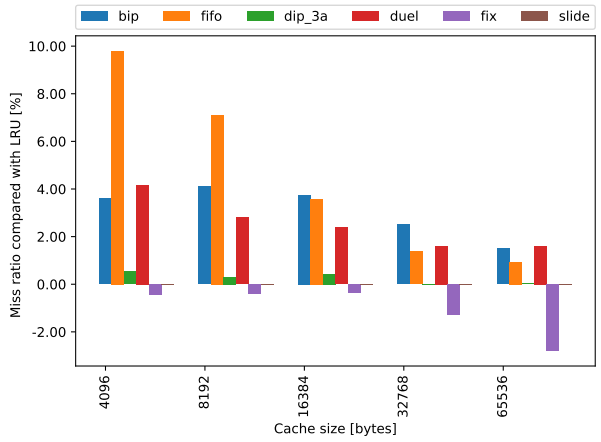


(b) lda-small

Fig. 2: Miss ratio compared with LRU for different cache sizes. (a) svd3-small. (b) lda-small.



(a) pca-small



(b) spc-small

Fig. 3: Miss ratio compared with LRU for different cache sizes. (a) pca-small. (b) spc-small.

The implementation of multiple replacement policies in a cache architecture demands some extra memory costs, as explained in [7]. For the online approaches, we have an extra cost of 3x32 sets to implement DIP+aging plus an extra 4 bytes per counter in each policy (DIP+aging and Set dueling) [6], avoiding the necessity of cache duplication. In the experiments we concluded that DIP+aging performs better than the Set Dueling approach for most of the cases. This is because the Set Dueling approach depends on the position of the sets selected for the eviction policies inside the cache memory and the memory access pattern of the program, while DIP+aging follows the best traditional policy independent of the memory access pattern, because all memory accesses are processed by all the policies. The reduction of miss ratio has a considerable impact in the execution time of the applications. For instance, we used parameters of different processors to calculate the execution times of the benchmarks as done in [8] and considering an A53 CPU running the pca-small with the configuration of 64 KB, where the miss ratio was reduced 30%, the execution time was reduced 20% when compared with the execution time using only the LRU policy.

V. Conclusions and Future Work

This paper has presented an extension of the Cachegrind framework to support the assignment of cache eviction policies considering code sections. We discussed and implemented four different approaches to select the most appropriate cache eviction policy (the one that maximizes the cache hit ratio) in specific code segments. Our evaluation using benchmark applications has shown a reduction of up to 30% in terms of cache misses, consequently, improving the execution time of industrial applications.

In future work, we will investigate how to integrate the policy selection techniques into a compiler to also support loop unrolling and a finer-grained selection and to investigate machine learning-based techniques to detect memory access patterns of code sections and to correlate those patterns with cache eviction policies.

References

[1] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Syst.*, vol. 37, no. 2, p. 99–122, Nov. 2007.

[2] V. Touzeau, C. Maïza, D. Monniaux, and J. Reineke, "Fast and exact analysis for lru caches," *ACM Program. Lang.*, vol. 3, jan 2019.

[3] J. Segarra, R. Gran Tejero, and V. Viñals, "A generic framework to integrate data caches in the wcet analysis of real-time systems," *Journal of Systems Architecture*, vol. 120, p. 102304, 2021.

[4] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. B. Taylor, "CortexSuite: A Synthetic Brain Benchmark Suite," in *Proc. of the IISWC*, Oct. 2014.

[5] J. Seward, N. Nethercote, and J. Weidendorfer, *Valgrind 3.3 - Advanced Debugging and Profiling for GNU/Linux Applications*. Net. The., 2008.

[6] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. of the 34th ISCA*, 2007, p. 381–391.

[7] R. Mancuso, H. Yun, and I. Puaut, "Impact of DM-LRU on WCET: A Static Analysis Approach," in *Proc. of the 31st ECRTS*, vol. 133, Dagstuhl, Germany, 2019, pp. 17:1–17:25.

[8] B. A. Araujo, G. Gracioli, T. Kloda, D. Hoornaert, and M. Caccamo, "Implementation and evaluation of adaptive cache insertion policies for real-time systems," in *Proc. of the XI SBESC*, 2021, pp. 1–8.

[9] O. Kotaba, M. Paulitsch, S. Petters, H. Theiling, and J. Nowotsch, "Multicore in real-time systems - temporal isolation challenges due to shared resources," in *Proc. of the WICERT 2013*, 2013.

[10] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of wcet tools," *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1038–1054, 2003.

[11] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[12] A. Jain and C. Lin, "Back to the future: Leveraging belady's algorithm for improved cache replacement," in *Proc. 43rd ISCA*, 2016, pp. 78–89.

[13] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proc. 52nd MICRO*, 2019, p. 413–425.

[14] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A Survey on Cache Management Mechanisms for Real-Time Embedded Systems," *ACM Computing Surveys*, vol. 48, no. 2, 2015.

[15] P. K. Valsan, H. Yun, and F. Farshchi, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *2016 IEEE RTAS*, 2016, pp. 1–12.

[16] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," *Proceedings - International Symposium on Computer Architecture*, pp. 60–71, 2010.