DyCa: Dynamically Adaptable Cache Bypassing Mechanism *

Mariana Carmin[1], Paulo Cesar Santos[2], Marco Antonio Zanata Alves[1]

[1] Department of Informatics – Federal University of Paraná (UFPR)
Curitiba – PR – Brazil

[2]Institute of Informatics – Federal University of Rio Grande do Sul (UFRGS)
Porto Alegre – RS – Brazil

[1]{mcarmin, mazalves}@inf.ufppr.br, [2]pcssjunior@inf.ufrgs.br

Abstract. *As the number of cores increases, more cores and threads share the Last-Level Cache (LLC), which consumes a large portion of the chip's total power and area. Therefore, sophisticated solutions must guarantee the best resource usage addressing cache conflicts and cache pollution problems. This work exploits the knowledge that many applications present poor temporal and spatial locality. Thus, an adaptive cache mechanism can benefit such applications, improving general system performance and decreasing energy consumption. In this paper, we propose an online and application-aware predictor to adapt the use of LLC. As a result, DyCa shows up to 22% and 21% performance increases in single and multi-program workloads, respectively.*

1. Introduction

Over the years, the industry has adopted large shared cache memories to allow several applications to share hardware resources and run concurrently in a multi-task environment. Consequently, the cache memory hierarchy consumes a large portion of the total area and energy budget [Egawa et al. 2019]. However, due to the heterogeneous nature of the applications, such memories are not guaranteed to be used efficiently.

Although cache memories are a great ally in mitigating the memory-wall problem, some applications do not benefit from this structure. It happens whenever the application does not present temporal or spatial data locality, making cache memory useless. It also negatively impacts the performance due to the extra latency imposed before the main memory access (on high cache misses scenarios). In a scenario where only such applications are running, we could also reduce the leaked energy consumption by disabling the Last-Level Cache (LLC). However, this analysis is out of the scope of this paper.

Moreover, during multi-program workloads, we could identify the applications that pollute a shared cache level with data that is not reused, known as "noisy neighbors" [Kim et al. 2019], and disable the cache access for these applications. Generally, these applications use a large portion of the shared resources. Consequently, identifying and avoiding noisy neighbors can improve general system performance by reducing conflicts and decreasing cache pollution[Egawa et al. 2019].

*This work was supported by the Serrapilheira Institute (grant number Serra-1709-16621) and CAPES (Brazilian government).

Considerable effort has gone into improving the performance of Last-Level Cache (LLC) both in academic and industrial research. However, considering each application may have a distinct memory access behavior, the same cache hierarchy will have different impacts on energy consumption and performance depending on the application.

Therefore, extensive studies try to identify the appropriated cache configuration [Kim et al. 2019, Egawa et al. 2019], using technologies such as Gated-Vdd [Powell et al. 2000] and Cache Allocation Technology [Kim et al. 2019], aiming to get higher energy efficiency and performance for these applications. However, adapting the LLC to individual applications remains a challenge.

Based on the observations above, we propose DyCA, the Dynamically Adaptable Cache Bypassing mechanism. DyCA relies on analyzing the processor's hardware performance counters at the running time during predefined intervals. These analyses allow DyCA to create an application-aware adaptable cache, considering each phase of the running programs. For this, we use a performance prediction to forecast the application's instructions per cycle (IPC) in different cache scenarios and a mechanism that decides what cache configuration is better for each application in a specific execution time. Thus, we can identify whether to adapt as the program behavior changes. We show that adapting the use of LLC during the application execution is possible and improve general system performance.

Our approach can be used in multi-threaded single-program workloads adapting and disabling the access to the LLC when gains in performance are identified and in multi-program workloads. As far as we know, this is the first proposal that supports multi-program executions. Our mechanism can adapt the entire use of LLC for each application individually, thus reducing cache pollution - caused by unfriendly cache applications - nevertheless removing the extra latency faced accessing the LLC.

The paper's organization is as follows. Section 2 presents related work. Section 3 describes our proposed architecture in detail. We describe the experimental environment and results in Sections 4 and 4.2. Finally, Section 5 concludes the paper.

2. Previous work on adaptive caches

Academic and industrial works have presented different approaches for adaptive cache memories by using bypass techniques. Some works adapt the LLC size. For example, Liu, Egawa, and Takizawa [Liu et al. 2020] developed a bypass mechanism in a cache level granularity. Both private and shared levels are bypassed. Private levels are bypassed if they make minimal impacts on performance. LLC, on the contrary, is not bypassed entirely. Instead, only a subset of cache ways is disabled, varying from the total capacity to 1/4 of the LLC capacity. This mechanism improves 26% the energy efficiency, retaining the same performance. Although, this work presents an offline mechanism and does not consider multi-program executions.

Another work from the same group [Liu et al. 2022] developed an online adaptable cache hierarchy application-aware, bypassing and disabling a less-significant cache layer to improve energy efficiency. In this work, the authors partially disable the LLC, adjusting the size and associativity according to the number of conflicts observed.

Another work that tests different LLC sizes was proposed by Mittal, Cao, and Zhang [Mittal and Zhang 2013] and tries to minimize energy consumption. They choose the smallest LLC by analyzing the energy consumption and performance trade-off for every application running, using a cache coloring scheme to allocate different sizes for each. Then, they turn off the unused space.

Some industrial solutions adapt the LLC size, including the one proposed by Samsung's researchers Lee et al. [Yang et al. 2012]. This work uses a power manager that can turn off any core and half of the L2 cache shared memory based on the processing throughput and amount of data required by the applications.

Differently, Kim et al. [Kim et al. 2019] use the Intel Cache Allocation Technology (CAT) in an automated mechanism, through machine learning, to predict the performance of an application in different cache sizes. The paper accurately predicts IPC with a 4.7% error on average.

Other works discuss the use of machine learning models for making bypass decisions. Sato et. al. [Sato et al. 2019] developed a hardware *perceptron* to predict dead blocks. They bypass the dead blocks to guarantee a more efficient cache way adaptation as the dead blocks are not present in the cache, and more ways can be disabled, decreasing energy consumption.

Although some works do not use the bypass, they present a cache adaptation mechanism. Zhu and Zeng [Zhu and Zeng 2021] use hardware counters and a decision tree to predict the best cache associativity. Adapting the L2 for the associativity find. Nevertheless, this decision is time-consuming since five execution phases are performed, each trying a different associativity configuration to make a decision.

Bypass is also widely used at a finer granularity. Köhler and Alves [Köhler and Alves 2019] use a possible misses predictor in the LLC request. If a request was classified as a miss by the predictor, the request bypasses the LLC, saving extra cycles of latency. In their paper, the average precision of the mechanism is 95%. The performance improvement observed is about 9%, although it can be up to 13% depending on the application behavior.

The work from Park, Kim, and Hou [Park and Hou 2021] is an example of an adaptable cache hierarchy mechanism. To develop this approach, they execute programs four times, varying the size of the LLC, and based on the information of each execution. The programs are classified into three types - bypass the entire LLC, use only a quarter of the LLC, and use LLC completely. Even though this work can be applied to a multi-program workload, improving up to 41.7% overall performance, their technique requires a costly offline stage.

Besides, several previous works present their proposals to improve Graphics Processing Units (GPU) architectures [Li et al. 2015, Xie et al. 2015]. These works aim to mitigate the congestion in the GPU cache caused by the vast number of threads using the bypass.

Table 1 summarizes the aspects of the more relevant related works for this paper and compares them with our proposed method.

Proposals	Online	Dynamic	Bypass	Multi-program	Entire LLC	Granularity
[Köhler and Alves 2019]	x	x	x	x		per request
[Liu et al. 2020]		x	x			per associative set
[Park and Hou 2021]			x	x		per application
[Liu et al. 2022]	x	x	x	x		per application
DYCA - Our Proposal	x	x	x	x	x	per application

Table 1. Summary of related work on cache bypassing.

3. DyCA - Dynamically Adaptable Cache Bypassing Mechanism

Our proposal is an application-aware dynamic mechanism using cache bypass to adapt the usage of LLC in the cache hierarchy. Consequently, we need an online mechanism to adjust the cache during run time.

DyCA principles depend on a learning phase and identification and run-time phase. First, in the learning phase, we use metrics from SPEC CPU 2006 [SPEC 2006] to train the models. On the one hand, **wLLC**, which stands for with LLC, predicts the IPC when LLC is used. On the other hand, **woLLC**, which stands for without LLC, predicts IPC when the use of LLC is disabled. Both use metrics for each execution window.

Second, at the run-time stage, DyCa is performed during program execution, deciding the best cache configuration for the next application's execution window based on the IPC predicted by the functions **wLLC** and **woLLC**. This way, we provide an online and dynamic mechanism to adapt to multi-program workloads and detect the different program phases. This process is illustrated in Figure 1.

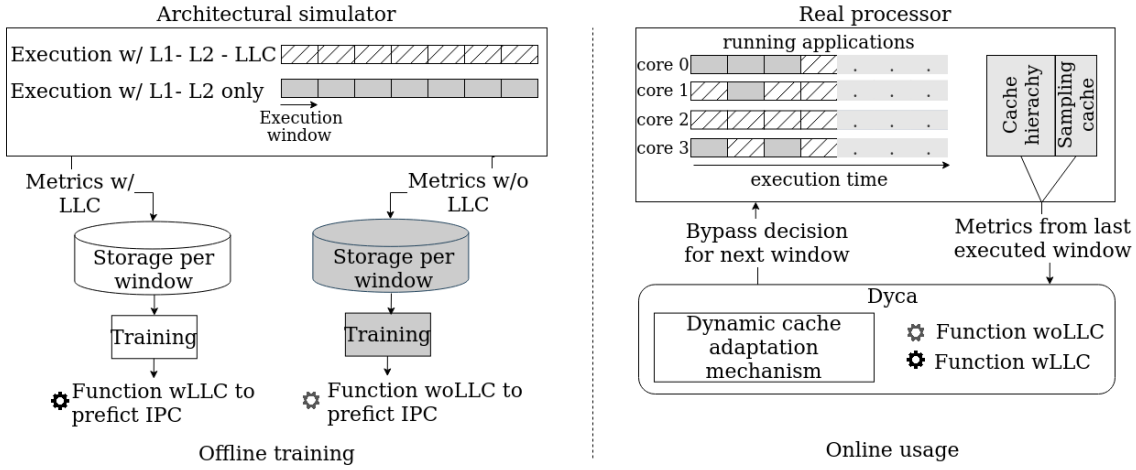

Figure 1. Diagram of the whole process of core adaptation using bypass.

The following sections will detail the mechanism. We first start with a theoretical analysis. After, we describe the sampling cache method used to keep the LLC counters [Qureshi et al. 2006]. Next, the linear regression model and the selected hardware counters are described. Section 3.4 explain the architecture proposal. To conclude, we discuss the performance and hardware overhead of our mechanism DyCa.

3.1. Theoretical Analysis

To motivate our proposal, we present an oracle mechanism to evaluate the maximum possible gains. Here, we have run simulations using an "oracle" predictor of the best usage of the last level cache (LLC). In other words, we analyze both executions with and without LLC, and for every 200 million cycles, we choose the configuration with the higher instructions per cycle (IPC).

With these results, we can obtain the best improvement in performance possible for a 200 million cycle execution window, which enables us to identify possible gains in performance and evaluate if our proposed mechanism is near the best scenario.

The decision to use a 200 million cycle execution window size considers that enabling or disabling the use of a cache level may cause overheads in performance. These overheads come from the need to invalidate data in the cache hierarchy and cold cache effects whenever we adapt the cache hierarchy. Furthermore, an execution phase can last several hundreds of instructions, making the observation in short periods useless. Additionally, a 200 million cycles slice is close to the average interval between the OS context switches [Alves 2014], changing cache configuration combined with the OS context switch can save the performance overhead discussed in Subsection 3.5.

We apply the oracle mechanism in SPEC CPU 2017[SPEC 2017], and the resulting speedup is illustrated in Figure 2.

We can observe that no application suffered performance degradation (speedup lower than one). Speedup equal to 1 represents steady performance, while values greater than 1 represent applications that benefit when the use of LLC is disabled.

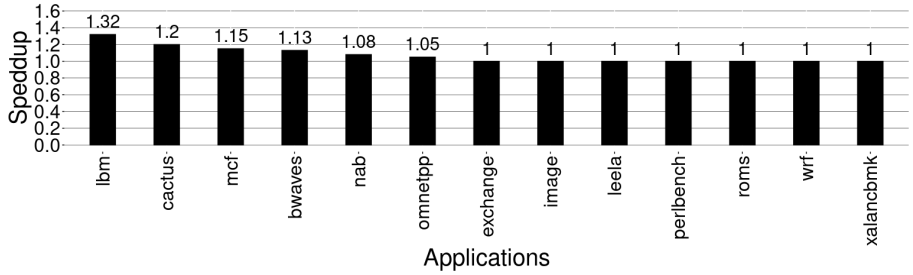

Figure 2. Oracle results for SPEC-CPU 2017 applications.

In this oracle, the overhead of disabling the LLC is not present because this overhead would be insignificant during the context switch. Thus, we are capable of comparing our mechanism to the perfect case.

3.2. Sampling the LLC

To keep track of LLC usefulness in the windows where we perform LLC bypass, we use the LLC sampling technique [Qureshi et al. 2006]. The main idea is to simulate the behavior of LLC in a small mechanism with just a few sampled sets. This mechanism reproduces all the access and changes in the LLC. However, it keeps only the tag-store from these sets - providing the number of misses and hits that are our interest - de-

creasing the overhead. Moreover, since this mechanism is not in the execution critical path, the latency increased by it is equal to zero.

The sets reproduced in the sampling cache are named leader sets. The number of leader sets can vary depending on the implementation. Our method to select leader sets is the following: assume N to be the number of sets in the LLC and the leader sets in sampling by K. We logically divide the LLC into equally K-sized regions, including N/K sets. In each region, we choose one leader set. Our best results derive from a simple-static policy [Qureshi et al. 2006]. In this policy, set 0 is selected from the first region, 1 from the second region, 2 from the third, and so on. In a multi-banked LLC, this process is done for each cache bank [Abad et al. 2015]. We illustrate the policy in Figure 3, where set 0 of region 1, set 1 from region 2, until set N from the last region is mapped to the sampling cache.

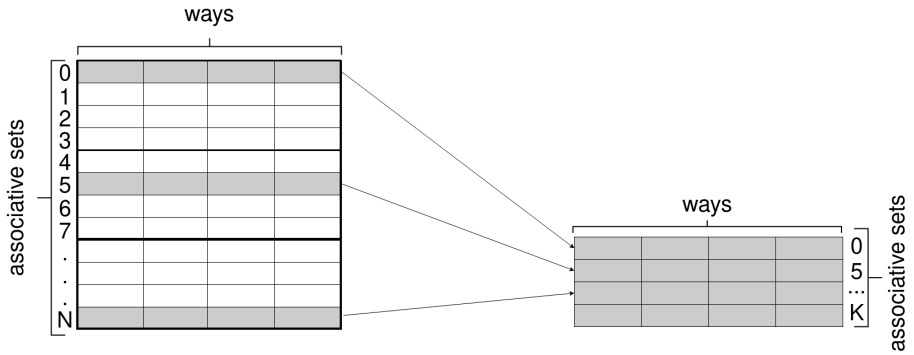

Figure 3. Mapping of leader sets.

In this work, we use 64 sets for the sampling size. Although this decision was taken by analyzing the average error for 16, 32, 64, 128, and 256 sets, minimizing the error is essential because it could cause wrong decisions in the decision model due to the values presented by the sampling cache. Therefore, the average was obtained from the average error -measured by the difference observed between the LLC hardware counters and sampling cache hardware counters- observed in each application of SPEC CPU 2017 [SPEC 2017] and SPEC CPU 2016 benchmark suites. Figure 4 show the average error and standard variation of each sampling size. We impose a limit of 0 in the graphic representation of the average error minus the standard variation, as errors below zero are impossible.

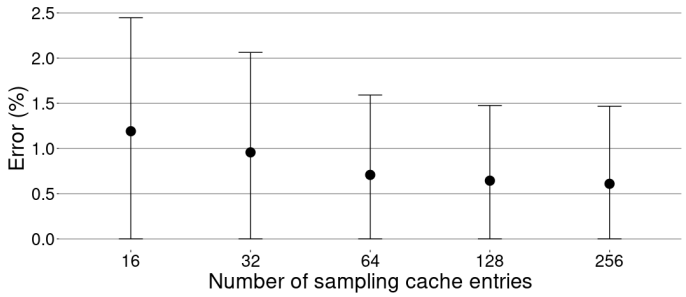

Figure 4. Average error observed for the different number of sampling sets.

As shown in Figure 4, the average error of 64 samples is smaller than 16 and 32. Also, the difference observed from 64 to bigger sizes is not significant, so as accurate

to the standard deviation. To summarize, we use 64 samplings to simulate LLC behavior since the hardware overhead is not high and presents a lower error and standard variation compared to a smaller number of sampling cache entries.

When we use sampling to collect LLC hardware counters, we need an adaptation to correctly predict metrics that involve several fetch instructions (i.e., HPKI and Misses Per Kilo-Instructions (MPKI)). Since sampling represents only a tiny portion of LLC, we multiply the absolute number of misses, hits, and access of the sampling by the N/K factor. N is the number of LLC sets, and K is the number of sampling sets (i.e., multiplying the sampling by the number of regions from LLC that it represents).

3.3. Linear regression model

To create a model capable of generalization, we used regression models in our approach to application performance prediction. More than this, a regression model can decide without using a threshold value. Regression models are statistical techniques that enable us to assess the impact of explanatory variables over response variables by estimating quantities that measure this effect. If these quantities are significantly different from zero, there is evidence of a significant effect of the explanatory variable over the response. It is also possible to make response predictions based on the observed values of the explanatory variable.

Dyca considers the hardware counters of the last executed window to predict the next one, as shown in Figure 1, and based on the observation that a program phase can last several hundreds of instructions, we expect similar behavior in subsequent execution windows. Furthermore, to train the model, the counters of the last windows are associated with the IPC of the next one. This way, training the model with the same configuration as the real processor has.

We trained two models, using Generalized Additive Models (GAM)[Hastie 2017], the first model, named **wLLC**, to predict the performance using the LLC and the second, named **woLLC**, to predict when bypassing the LLC. The hardware counters selected for wLLC are presented in Equation 1. GAM relaxes the restriction that the relationship must be a simple weighted sum and instead assume that the outcome can be modeled by a sum of arbitrary functions of each feature.

$$IPC = s_0 L1\ HPKI + s_1 L1\ hit\ ratio + s_2 L2\ MPKI + s_3 LLC\ accesses + s_4 Loads \quad (1)$$

Equation 2 show the hardware counters for woLLC. The equations are similar. Only two counters are different, L1 accesses - in Equation 1 L1 hit ratio is used and L2 HPKI, corresponding to L2 MPKI.

$$IPC = s_0 L1\ HPKI + s_1 L1\ acesses + s_2 L2\ HPKI + s_3 LLC\ acesses + s_4 Loads \quad (2)$$

These hardware counters were not empirically selected. Instead, we collected a list of twenty possible candidates for each cache layer, i.e., L1, L2, and LLC, and analyzed the correlation between each counter and the IPC value. From these analyses, we could select the best ones to compose the models.

3.4. The Mechanism Architecture

We develop two possible architectures using the sampling cache. **SingleSC** has only one sampling shared cache, accessed by all the cores. Moreover, **MultiSC** architecture has the sampling shared cache and a private sampling cache for each core. In Figure 5 both architectures are shown.

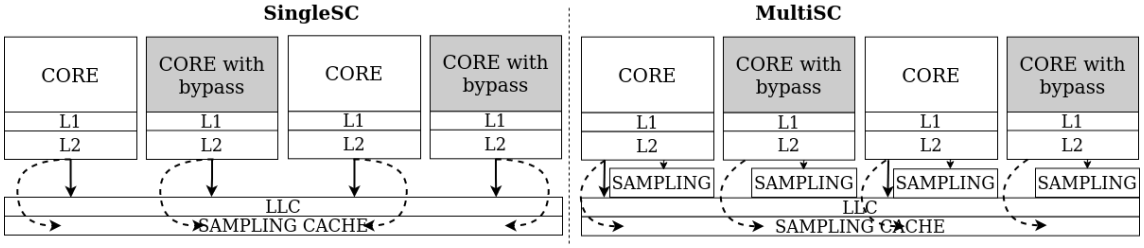

Figure 5. SingleSC and MultiSC architectures.

The sampling cache receives all the access from the L2 cache. When a core is not using LLC, illustrated in grey in Figure 5, the access from L2 remains to be sent to the sampling cache even though they are not sent to the LLC.

Both **SingleSC** and **MultiSC** architectures are tested and analyzed in the regression model to evaluate which one presents the best performance. Architecture **MultiSC** had a 2% higher accuracy for a multi-program workload.

Adding an extra bit is enough to store the behavior decided by the mechanism. Since it is a binary decision, the bit is added to the Translation Lookaside Buffer (TLB).

3.5. Performance and hardware overhead

When analyzing the hardware and performance overhead of adding a sampling cache, an important aspect to consider is that this cache stores only the tag, requiring a small area. For one sampling cache, the hardware overhead is equal to 6Kb, considering the choice to use 64 sets and the **SingleSC** architecture. For a **MultiSC** architecture with four cores, the overhead is equal to 30Kb. We also added a simple decision logic.

Regarding its latency, since the sampling cache is not in the execution critical path, the latency provided by it is equal to zero, not increasing the total latency of the cache accesses.

Furthermore, when a change in the use of LLC is executed, some performance overhead is added. First, it is necessary to empty the pipeline before fetching instructions in the new LLC use configuration. Secondly, some changes in the cache hierarchy are made for the correct maintenance of the coherence protocol. Therefore, we adopted the worst-case scenario to simulate this overhead by invalidating all cache levels.

In conclusion, this overhead simulation will affect the proposal, made impossible to obtain the same performance as the oracle experiment. Although, this overhead needs to be reasonable to keep the mechanism feasible. The results are discussed in Section 4.2.

4. Methods and Results

This section describes the architectural aspects of the simulation, and the benchmarks used and explains the simulation decisions and mechanisms applied in this proposal.

4.1. Simulation Environment and Benchmark Suites

To perform our experiments, we used the Ordinary Computer Simulator (OrCS), an in-house cycle-accurate and trace-driven simulator based on SiNUCA [Alves et al. 2015], to perform our experiments. First, we train the regression model with 24 benchmarks from the SPEC CPU 2006 suite. Furthermore, we test the model with 14 benchmarks from the SPEC CPU 2017 benchmark suite to simulate the mechanism's performance when we observe the application evolution over the years. The simulations used 2 billion most representative instructions from the benchmark extracted with the Pin-Points [Patil et al. 2004] tool for both suites.

4.2. Results and Discussion

We executed some experiments to understand and evaluate our approach. We observe the proposal's performance improvement in a single and multi-program workload, single-program using **SingleSC** architecture, and multi-program using **MultiSC** architecture since we observe a higher accuracy in this configuration.

4.2.1. Single core execution

In order to evaluate the mechanism performance in each application, we evaluate DyCa when executing a sequential and single application per time (i.e., SPEC CPU 2017). The result is shown in Figure 6.

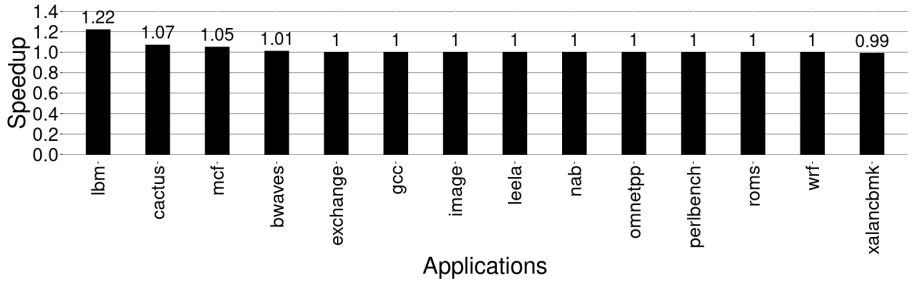

Figure 6. DyCa result for SPEC CPU 2017 with only one application.

For a single workload, performance degradation of 1% is observed in one application (*xalancbmk*). In Figure 2, we can observe that for this application, no improvement in performance is reached when adapting the use of the LLC. Because of that, the degradation is attributable to a model miss-prediction, causing a wrong adaptation of using LLC, which hurt performance.

In addition, some applications - *lbm*, *cactus*, *mcf* and *bwaves* - shows improvements in performance up to 22%, these are between the applications observed as the ones that gains performance when the use of LLC is disable.

The other applications maintain the same performance (i.e., do not improve performance by adapting the use of LLC). In preview experiments, most of the appli-

cations did not present an increase in performance when the use of LLC was disabled, as illustrated in Figure 2.

To sum up, DyCa correctly identified the applications that should have the use of LLC disabled, showing a performance improvement. Moreover, this correctness extends to applications that should not disable the use of LLC. The mechanism's efficiency is directly related to the accuracy of the prediction. In this experiment, only one false positive case was observed. Although, we observe a performance overhead of 4% comparing the oracle version with DyCa, associated with all the control discussed in Subsection 3.5.

4.2.2. Multi-program workload

In order to understand the performance of a real system where multi-applications are sharing the LLC, we perform multi-program workload analyses.

Four application bundles were produced to create the multi-program workloads. The first is an LLC-compatible-apps bundle, created with only applications that gain performance when LLC is disabled. The LLC-compatible (a) consist of *lbm*, *cactus*, *mcf* and *bwzves* applications and LLC-compatible (b) contain the applications *mcf*, *bwaves*, *nab* and *omnetpp*. The second bundle is LLC-incompatible apps, including only applications that lost performance. In this case, containing *wrf*, *roms*, *perlbench* and *xalancbmk* applications.

Besides, a Mixed-apps bundle was created with half applications that gain performance, and half that lose. The first bundle (i.e. Mixed (a)) represent the execution of *lbm*, *cactus*, *xalancbmk* and *wrf*. The second (i.e. Mixed (b)) considered applications *mcf*, *bwaves*. *roms* and *perlbench*.

Finally, a Random-apps bundle with randomly chosen applications, in our experiment the chosen ones are *lbm*, *bwaves*, *image* and *perlbench*. Figure 7 describes the result for SPEC-CPU 2017 for the four workloads identified at the bottom of each column.

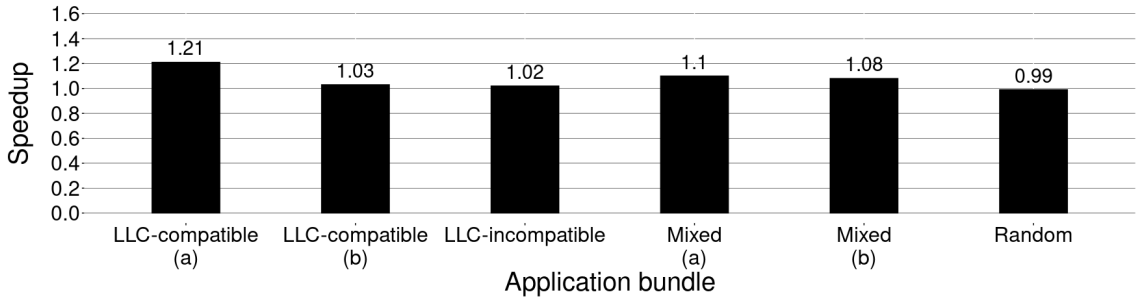

Figure 7. DyCa result for SPEC CPU 2017 with four applications.

It is possible to notice that even considering the overhead associated with each LLC use adaptation DyCa still gains performance in almost every configuration. On the other hand, for the Random-apps bundle, 1% of performance is lost. Furthermore, it is essential to notice that for this configuration, no gain in performance is observed when the use of LLC is disabled for the entire execution. Thus, such performance degrada-

tion is associated with a miss-prediction of the model.

Regarding the results, for both Mixed-apps bundles, we observe gains in performance that could be associated with the cache pollution and conflicts avoided by disabling the LLC use for the unfriendly cache applications.

It should be noted that every time DyCa switches between using or bypassing the LLC, it poses a time overhead arising from the mechanism to maintain cache coherence. Although, on average, these bundles have observed a loss of only $1,4\%$ performance.

5. Conclusion and future work

In this work, we proposed a new model to dynamically bypass the Last Level Cache (LLC) based on the performance prediction emulating different cache hierarchy conditions, with L1-L2-L3 or L1-L2 only. Our mechanism presents a low hardware overhead, relying only on a small sampling cache and a simple decision logic.

Our linear regression model can generalize application behavior, predicting the Instructions Per Cycle (IPC) for applications it has never seen before. Thus, it does not need stabilizing thresholds. DyCa can be used with only one application running, showing performance gains, and if LLC is turned off when it is not used, we could reduce the leaked energy. DyCa reduces global pollution for multiple applications, increases average system performance, and reduces cache conflicts.

In future works, we understand that our mechanism could be expanded to multi-thread applications. We also envision a study about classification models to make the bypass decisions.

References

Abad, P., Prieto, P., Puente, V., and Gregorio, J. A. (2015). Improving last level shared cache performance through mobile insertion policies (mip). *Parallel Computing*, 49:13–27.

Alves, M. A. Z. (2014). Increasing energy efficiency of processor caches via line usage predictors.

Alves, M. A. Z., Villavieja, C., Diener, M., Moreira, F. B., and Navaux, P. O. A. (2015). Sinuca: A validated micro-architecture simulator. In *17th International Conference On High Performance Computing And Communications (HPCC)*, pages 605–610.

Egawa, R., Saito, R., Sato, M., and Kobayashi, H. (2019). A layer-adaptable cache hierarchy by a multiple-layer bypass mechanism. In *Proceedings of the 10th Int. Symp. on Highly-Efficient Accelerators and Reconfigurable Technologies*, pages 1–6.

Hastie, T. J. (2017). Generalized additive models. In *Statistical models in S*, pages 249–307. Routledge.

Kim, Y., More, A., Shriver, E., and Rosing, T. (2019). Application performance prediction and optimization under cache allocation technology. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1285–1288.

Köhler, R. and Alves, M. A. Z. (2019). Acelerando requisições de prováveis cache misses com requisições em paralelo cache/dram. In *Anais Estendidos do IX Simpósio Brasileiro de Engenharia de Sistemas Computacionais*, pages 101–106. SBC.

Li, C., Song, S. L., Dai, H., Sidelnik, A., Hari, S. K. S., and Zhou, H. (2015). Locality-driven dynamic gpu cache bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 67–77.

Liu, J., Egawa, R., Agung, M., and Takizawa, H. (2020). A conflict-aware capacity control mechanism for last-level cache. In *2020 Eighth International Symposium on Computing and Networking Workshops (CANDARW)*, pages 416–420. IEEE.

Liu, J., EGAWA, R., and TAKIZAWA, H. (2022). A conflict-aware capacity control mechanism for deep cache hierarchy. *IEICE Transactions on Information and Systems*, 105(6):1150–1163.

Mittal, S., C. Y. and Zhang, Z. (2013). Master: A multicore cache energy-saving technique using dynamic cache reconfiguration. *Transactions on very large scale integration (VLSI) systems*, 22(8):1653–1665.

Park, J., K. S. and Hou, J. U. (2021). An l2 cache architecture supporting bypassing for low energy and high performance. *Electronics*, 10(11):1328.

Patil, H., Cohn, R., Charney, M., Kapoor, R., Sun, A., and Karunanidhi, A. (2004). Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation. In *Int. Symp. on Microarchitecture (MICRO-37'04)*, pages 81–92.

Powell, M., Yang, S.-H., Falsafi, B., Roy, K., and Vijaykumar, T. (2000). Gated-vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the 2000 international symposium on Low power electronics and design*, pages 90–95.

Qureshi, M. K., Lynch, D. N., Mutlu, O., and Patt, Y. N. (2006). A case for mlp-aware cache replacement. In *33rd Int. Symp. on Computer Architecture (ISCA'06)*, pages 167–178.

Sato, M., Chen, Y., Kikuchi, H., Komatsu, K., and Kobayashi, H. (2019). Perceptron-based cache bypassing for way-adaptable caches. In *2019 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*, pages 1–3. IEEE.

SPEC (2006). SPEC CPU 2006. `https://www.spec.org/cpu2006`. Online; accessed 08 November 2021.

SPEC (2017). SPEC CPU 2017. `https://www.spec.org/cpu2017`. Online; accessed 08 November 2021.

Xie, X., Liang, Y., Wang, Y., Sun, G., and Wang, T. (2015). Coordinated static and dynamic cache bypassing for gpus. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 76–88. IEEE.

Yang, S.-H., Lee, S., Lee, J. Y., Cho, J., Lee, H.-J., Cho, D., Heo, J., Cho, S., Shin, Y., Yun, S., et al. (2012). A 32nm high-k metal gate application processor with ghz multi-core cpu. In *2012 IEEE Int. Solid-State Circuits Conference*, pages 214–216.

Zhu, W. and Zeng, X. (2021). Decision tree-based adaptive reconfigurable cache scheme. *Algorithms*, 14(6):176.