Near-optimal replacement policies for shared caches in multicore processors

JAVIER DÍAZ, Aragón Institute of Engineering Research (I3A), University of Zaragoza (Spain), and Hipeac
PABLO IBÁÑEZ, Aragón Institute of Engineering Research (I3A), University of Zaragoza (Spain), and Hipeac
TERESA MONREAL, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya (Spain), and Hipeac
VÍCTOR VIÑALS, Aragón Institute of Engineering Research (I3A), University of Zaragoza (Spain), and Hipeac
JOSÉ M. LLABERÍA, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya (Spain), and Hipeac

Manuscript communications:

imarin@unizar.es

ABSTRACT

An optimal replacement policy that minimizes the miss rate in a private cache was proposed several decades ago. It requires knowing the future access sequence the cache will receive. There is no equivalent for shared caches because replacement decisions alter this future sequence. We present a novel near-optimal policy for minimizing the miss rate in a shared cache that approaches the optimal execution iteratively. During each iteration, the future access sequence is reconstructed on every miss interleaving the future per-core sequences, taken from the previous iteration. This single sequence feeds a classical private-cache optimum replacement policy. Our evaluation on a shared last-level cache shows that our proposal iteratively converges to a near-optimal miss rate that is independent of the initial conditions, within a margin of 0.1%. The best state-of-the-art online policies achieve around 65% of the miss rate reduction obtained by our near-optimal proposal. In a shared cache, miss rate optimization does not imply the optimization of other metrics. Therefore, we also propose a new near-optimal policy to maximize fairness between cores. The best state-of-the-art online policy achieves 60% of the improvement in fairness seen with our near-optimal policy. Our proposals are useful both for setting upper performance bounds and inspiring implementable mechanisms for shared caches.

KEYWORDS

Chip multiprocessor, shared cache, optimal replacement, miss rate, fairness

This is a post-peer-review, pre-copyedit version of an article published in *The Journal of Supercomputing*.
The final authenticated version is available online at: http://dx.doi.org/10.1007/s11227-021-03736-1

1. INTRODUCTION

A critical component of a cache memory is the replacement policy [1]. In its broadest sense the replacement policy is responsible for determining the identity of the incoming and outgoing cache blocks, in order to minimize the number of misses. In the event of a miss, the replacement algorithm first decides whether the block that misses is loaded into the cache or bypassed [2]. Secondly, when a block is loaded, the replacement algorithm will choose a victim block to be evicted. In almost all replacement policies, the bypass decision and victim block selection are based on stay or replacement priorities for each block, which the algorithm manages to estimate future block usefulness.

Nowadays, chip multiprocessor (CMP) systems dominate the market in high-performance servers, desktop or embedded systems, and mobile devices [3]. Their most common design includes a multilevel cache hierarchy, ending with a shared last-level cache (SLLC) [4]. Looking specifically at the current top 10 supercomputers [5], all but one of them include general-purpose CMPs that have a multilevel cache: ARM A64FX [6], IBM Power9 [7], AMD EPYC [8] and Intel Xeon [9]. The contents of each cache are managed by its own replacement policy, implemented in hardware. Although replacement algorithms have been improving for more than 50 years, ongoing changes in capacity, system architecture (single-core, multi-core, multi-threaded) and memory technology (SRAM, eDRAM, STT-RAM, etc.), among other factors, still encourage the development of optimizations or radically different proposals [10].

The replacement policy is important because it is largely responsible for the performance of the cache. In shared caches, which serve several competing cores or threads, it is also partially responsible for fairness and quality of service (QoS). The importance of the replacement policy is testified to by the organization of a cache replacement championship twice in recent years, driven by the community of computer architects [11, 12]. These championships have been focused on policies for SLLCs.

Ideally, a replacement algorithm would make every request to a cache a hit, such that the requested block is serviced as soon as possible. This is evidently not feasible. The OPT cache replacement algorithm minimizes the number of misses for a specific cache organization (size, associativity, etc.) and is therefore considered the optimal policy [13]. OPT requires knowledge of the future and is therefore not implementable on real systems, but can be implemented offline in a simulated environment.

It is worth developing an optimal policy for several reasons:

- To determine the performance gap between proposed replacement policies and the optimum, this being the room for improvement for new replacement policies.
- To identify and analyze specific desirable behaviors of the optimal policy, to inspire real-life implementations that mimic or approach optimal algorithms.

Unfortunately, OPT can only be applied to private caches, those that only serve one core, program, or thread. It cannot be applied to shared caches. This is because it assumes that the future sequence of accesses to the cache is independent of the cache replacement algorithm. That is, it depends only on the processor and the memory hierarchy between the processor and the cache level where the algorithm is executed. This is not true for shared caches.

Therefore, there is no known optimal policy that minimizes misses for shared caches. The maximum performance that can be achieved by improving the replacement policy in a shared cache is unknown.

Our goal in this paper is to propose new algorithms for shared caches that are approximations to algorithms that would optimize certain specific metrics. We call this family of algorithms near-optimal replacement algorithms (NOPTs). These algorithms can be used in the design phase when developing new replacement policies for shared caches, the same way OPT has been used for private caches since it was proposed.

Specifically, our contributions are the following:

- We first propose a NOPT to minimize the miss rate for shared caches. It is an iterative algorithm that is applied to consecutive runs of the same workload. Our experiments show that, after several runs, the miss rate for each of our workloads converges to a minimum. To the best of our knowledge, this is the first study that seeks to know the theoretical minimum miss rate for shared caches and provides a successful approach.
- Secondly, having noticed that an optimal shared cache that minimizes miss rate does not benefit all programs in the workload equally, we propose a NOPT to maximize fairness between threads. Our experiments show that, with this algorithm, we obtain the best fairness of all replacement policies considered.
- Additionally, we compare the performance of current state-of-the-art policies with that of our proposals. We show that the best previous policies achieve 65% of the miss rate reduction obtained by applying our NOPT designed to minimize miss rate (vs. random), and 75% of the increase in instructions executed per cycle (IPC). Further, our results are maintained when doubling the number of cores in the system. Regarding fairness, the best state-of-the-art policy achieves 60% of the improvement obtained

with our near-optimal policy, and the second best only 45%. The gaps increase when doubling the number of cores.

The paper is structured as follows. Section 2 explains the background and motivation for this work. Section 3 describes in detail the NOPT proposed to minimize miss rate. Section 4 details the methodology used in our evaluations, including the experimental environment and the configuration of the simulated systems. Section 5 evaluates the NOPT to minimize miss rate. Section 6 describes our NOPT proposed to maximize fairness between cores and Section 7 evaluates this algorithm. Section 8 presents results for state-of-the-art online policies and compares them to our near-optimal proposals. Section 9 presents an analysis on how sensible our proposals are to the most important variable in a CMP, the number of cores. Finally, in Section 10, we summarize our conclusions.

2. BACKGROUND

2.1 Replacement policies for shared caches in multicore processors

Cache replacement algorithms for processors have been improving for more than 50 years. Most of the published work has focused on SLLCs [10], as on those caches the benefits of a more effective algorithm are larger: it is difficult to manage SLLC contents because both temporal and spatial localities are diminished in the stream of references observed by the SLLC [14, 15].

These replacement policies are based on various mechanisms. Some prioritize blocks based on their recency of access, such as LRU, SRRIP [14] and PDP [16]. Other use the frequency of accesses, like LFU, LFRU [17] and SAR [18]. The re-reference distance is the basis for Timekeeping [19], EHC [20] and Leeway [21]. A mixed approach is used in Modified LRU [22], DIP [15], DRRIP [14], SBAR [23], EVA [24] and ACR [25]. Recent state-of-the-art policies classify blocks in cache-averse or cache-friendly and manage each class differently, such as DBCP [26], AIP/LvP [27], CHAR [28], the Reuse Cache [29], SHiP [30], ReD [31] or Hawkeye [32].

2.2 Optimal policies

The studies of Belady [33] and Mattson et al. [13] are the first to address the issue of obtaining the absolute minimum number of cache misses produced by a known sequence of references, and they do so from different perspectives. The first approach proposed, MIN, determines whether a reference causes a miss or a hit in a cache managed with an optimal algorithm by analyzing the past sequence of references, while the second one, OPT, decides the block to be replaced to obtain the minimum miss rate by analyzing the future sequence of references. A few years later, Belady and Palermo proved that MIN and OPT provide identical results [34]. The optimal algorithm with bypass (OPTb), called pass-through initially, is presented more than a decade later [35]. This is the optimal algorithm when the cache is not required to store an incoming block.

In a recent paper, Michaud develops new ways to describe the OPT and OPTb algorithms that help to improve our understanding of them and mathematically demonstrates some of their properties [36].

Several papers about cache replacement policies have used OPT in recent years. Some of them use it as a reference when presenting results [15, 22, 23, 26, 28, 37]. But, even when designing a policy for shared caches, its usage is limited to single-core comparisons. Authors of a recent study explicitly state their desire to compare with optimum results for SLLCs, but rule it out as impossible [38].

Other authors not only use OPT for presenting results, but also study and seek to mimic its behavior in new replacement policy proposals [27, 32, 39-42]. Again, they sometimes use single-core OPT to guide shared cache designs, because there is no multi-core alternative.

Similarly, other offline policies have been designed to represent the ideal application of a specific mechanism [28, 43]. These are often called “oracle” policies. They are used to guide the development of implementable online policies based on the same principles.

2.3 OPT and OPTb for a private cache

In a set-associative cache, OPT selects the block in the cache set that is going to be referenced the furthest into the future as a victim, and it always inserts the missing block in the cache [13]. Supposing that a 4-way set of a cache contains blocks A, B, C and D, and that the future sequence of references to blocks in this set is FACEDABFC, the OPT algorithm would choose to replace block B to make room for block F. This is because blocks A, C and D are going to be referenced before block B.

The authors of the aforementioned study demonstrated that this algorithm minimizes the miss rate in the cache, and thereby, optimizes conventional performance metrics such as the execution time and the IPC. Though it cannot be implemented on a real cache because it requires knowledge of the future sequence of address references, it can be implemented offline in a simulated environment, when the sequence is known from a previous, identical run.

OPTb is the optimal policy when the cache allows bypassing. In this architecture, a requested block does not need to be inserted into the cache, it can be *bypassed*. It is for example used in non-inclusive L2 or L3 caches [44]. OPTb is similar to OPT but it also considers the requested block as a potential victim. If it is re-referenced the furthest into the future, the incoming block is not inserted into the cache and no block is evicted. In our previous example, OPTb would bypass the incoming block F instead of evicting B, because all the blocks present in the cache are going to be referenced before the next reference for block F.

OPT and OPTb are implemented in two phases. In the first phase, the program is simulated, and the sequence of access requests made to the cache is recorded and stored in a file. In the second phase, the simulation is run again, but this time, replacement decisions are taken using OPT/OPTb, which look for the future references in the file generated in the first phase.

Evidently, replacement decisions in the second phase may differ from those taken in the first phase. A change in a replacement decision may modify the service time for a future request due to the difference in hit and miss times, and therefore, alter the instruction execution speed of the processor. This, in turn, may change the timing of future accesses to the cache. In a private cache, however, this change will not modify the sequence itself: it will still include the same block references in the same order (assuming the processor executes instructions in program order).

In other words, the access sequence received by a private cache does not depend on its own behavior. Therefore, during the first phase of OPT and OPTb, the cache can use any organization and replacement policy. In fact, simulating the program execution and its interactions with the memory hierarchy between the processor and the studied cache would be enough.

2.4 Why it is not possible to implement OPT and OPTb for shared caches

A shared cache receives several streams of accesses, one per core or thread sharing the cache. The interleaving of these per-core sequences forms the global access sequence the cache receives. To form this global sequence, it is necessary to know not only the individual sequences but also their timings. As in a private cache, the replacement policy of a shared cache cannot modify the content or the order of references in each individual stream, but it can change timings between them, because hits and misses have different service times. This, in turn, can alter the stream interleaving, and therefore the global sequence itself.

Fig. 1 illustrates, with an example, how this affects the execution of OPTb on a system with a shared cache. A cache set contains blocks A, B, X and Y. Two cores are going to access this cache set. The future access sequences are shown as S_0 and S_1 respectively. We execute phase 1 of the OPTb algorithm with two different replacement policies α and β. They result in different timings due to the different replacement decisions, and form two different global streams $S_{\alpha G}$ and $S_{\beta G}$. When running the simulation again in phase 2 of the OPTb algorithm, core 0 first accesses block A, and a replacement decision is required. When working with global stream $S_{\alpha G}$, OPTb evicts block B, while when working with global stream $S_{\beta G}$, the first evicted block is Y. Clearly, it is not possible for both to be the optimal decision.

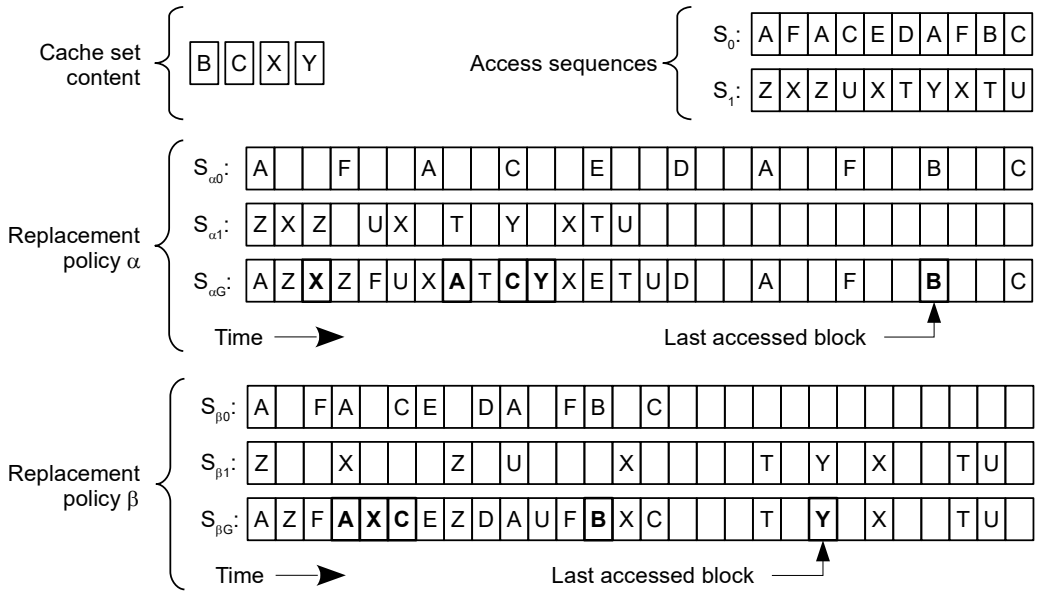


Fig. 1 An example of the OPTb algorithm naively applied to a shared cache. Top: cache set contents and access sequences for a 4-way cache in a 2-core system. Middle: local and global streams resulting from running phase 1 of the OPTb algorithm with replacement policy α, in which B is the last accessed block from the initial cache set contents. Bottom: local and global streams resulting from replacement policy β, in which Y is the last accessed block. The difference in access timings between policy α and β appear because of the different replacement decisions. Highlighted boxes indicate first accesses to blocks considered for replacement or bypass. The last accessed block found in phase 2 of the OPTb algorithm is not the same in both cases.

OPT and OPTb require the global sequence they work with to be the same as the future stream the cache will receive. That is, the global access sequence that is captured in phase 1 must be the same as that obtained when applying the optimal algorithm in phase 2. This can only happen if we already know, during phase 1, the timings that the optimal replacement policy will generate. As they depend on block service times, which depend on whether accesses are hits or misses, and in turn, on the replacement policy, we would need to know the optimal replacement decisions during phase 1. This is impossible. Therefore, OPT and OPTb cannot be used as optimal replacement policies for shared caches.

2.5 Optimal replacement policies for shared caches

There are some specific characteristics of shared caches, not present in private caches, that need to be considered when developing an optimal policy for them.

First, as the global sequence the shared cache receives depends on timings, and those depend on the whole processor and memory organization, optimal replacement decisions for shared caches also depend on the timing behavior of the rest of the system. Therefore, the optimal cache replacement policy may differ between two computer organizations, even when the workload and the physical characteristics of the shared cache are the same.

Additionally, even the definition of optimum is not unique for a shared cache. Minimizing the miss rate of the cache does not automatically imply an optimal result in other metrics such as throughput (instructions executed in the whole system per unit of time), weighted speedup [45] or fairness. Therefore, it is possible to propose various optimization goals for a shared cache.

Our objective in this paper is to propose new algorithms for shared caches that are approximations to algorithms that would optimize specific metrics, calling this family of algorithms NOPTs. Our proposals require knowledge of the future and therefore are offline policies, like OPT and OPTb. We apply them to SLLCs, the most common use of shared caches in CMPs nowadays[1].

The optimal algorithm we seek to approximate is OPTb. We decided to work with this variant because it obtains the minimum miss rate for a given cache organization and size. As mentioned, it requires the cache to be able to bypass incoming blocks. It is straightforward to adapt our proposals to a cache that does not allow bypassing by just taking OPT as the basis instead of OPTb.

[1] Another common use of shared caches appears in simultaneous multithreading processors, where a single core executes several threads in parallel [46].

3. A NEAR-OPTIMAL REPLACEMENT ALGORITHM TO MINIMIZE MISS RATE

In this section, we propose a replacement algorithm whose goal is to achieve a miss rate close to the optimum for an SLLC. We call it NOPTb-miss, standing for near-optimal replacement algorithm with bypass to minimize miss rate.

3.1 NOPTb-miss design

Our proposed method for achieving a result close to the optimum in miss rate is to iteratively approach the optimal execution and the optimal global access sequence. Our hypothesis is that the global sequence obtained in an iteration with NOPTb-miss is closer to what an optimal algorithm would produce than the one obtained in the previous iteration.

We propose the following procedure:

1. Simulate the workload, using any replacement policy in the SLLC.
2. For each execution core, store its individual access stream to the SLLC in files. Include the timing of the accesses with this initial replacement policy, that is, the cycle when each request arrives at the SLLC.
3. Simulate the workload using OPTb, feeding it with a global access sequence that is dynamically constructed using the individual sequences generated and stored in the previous execution.
4. For each execution core, store its individual access stream to the SLLC in files, including the new access timings.
5. If the exit condition is not met, return to step 3 and iterate.

The files generated in steps 2 and 4 store, separated by the core and SLLC set, the list of accesses to the SLLC and the time (in cycles) between two consecutive accesses.

Fig. 2 is a schematic diagram of the simulation of iteration *i* of NOPTb-miss (steps 3 and 4) using a cache shared by two cores.

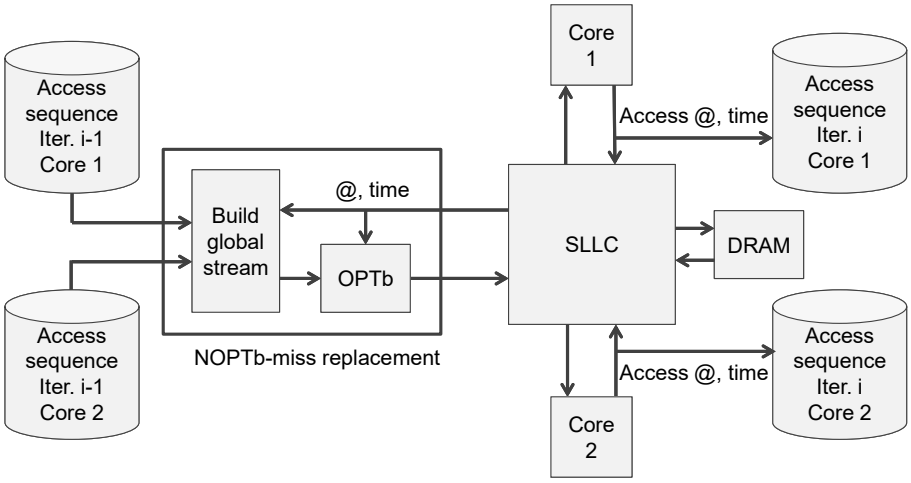


Fig. 2 Schematic diagram of the simulation of iteration *i* of NOPTb-miss

During iteration *i*, the simulation executes the applications running in all cores, including its private caches, and generates requests made to the SLLC. In the event of a miss, the SLLC uses the NOPTb-miss replacement policy. Each time a replacement is required, this policy reads the files created in iteration *i-1* to build, for the set affected, the global access sequence expected in the immediate future. It does so by interleaving the access streams of all cores, using the timing information included[2]. The replacement decision is taken using OPTb, fed by this future global access sequence, calculated for that specific time.

It is important to highlight that it is not enough to store and use a static global access sequence. Each execution changes the relative speed between cores, due to the different optimization possibilities of the various access patterns of the simulated applications. These variations accumulate over time and would quickly invalidate the information stored in a static global access sequence.

NOPTb-miss dynamically adapts to these changes. For each set, it maintains pointers to the individual access sequences indicating the last access the SLLC has received from each core. These pointers show the

[2] If two accesses from different cores arrive to the SLLC at the same time, the access from the lower-ordered core is taken first.

current execution point of each core. When a core generates more accesses per cycle than in the previous iteration due to a better miss rate, its pointers advance faster than before, meaning that the computation for that core is relatively more advanced. The global access sequence is dynamically built interleaving the individual access streams starting from the pointers.

Using this mechanism, NOPTb-miss adapts to past variations in core execution speeds and does not allow such variations to accumulate as the execution advances. A valid future global access sequence is fed into OPTb to take its replacement decisions; however, these decisions are sometimes sub-optimal because the timings used to interleave the individual streams come from a previous run. We expect the count of optimal decisions to increase as we iterate, as timings incrementally approach those obtained with the static global access sequence that would correspond to the optimal replacement.

The exit condition in step 5 depends on the goals set to achieve with NOPTb-miss and the precision required for the solution. We suggest iterating until the reduction in miss rate between iteration *i* and *i-1* is lower than a defined threshold. In section 5 we give details about the exit condition applied in our evaluation.

3.2 NOPTb-miss example

In this section, we present an example to illustrate how NOPTb-miss works in a cache shared by two cores. Fig. 3 shows, at the top, the access sequences stored during iteration 1 for both cores and a specific cache set. They include the block address (“A”, “B”, etc.) and the timing of each access, relative to the previous one. For simplicity, we assume all accesses in the example have missed in the SLLC in iteration 1.

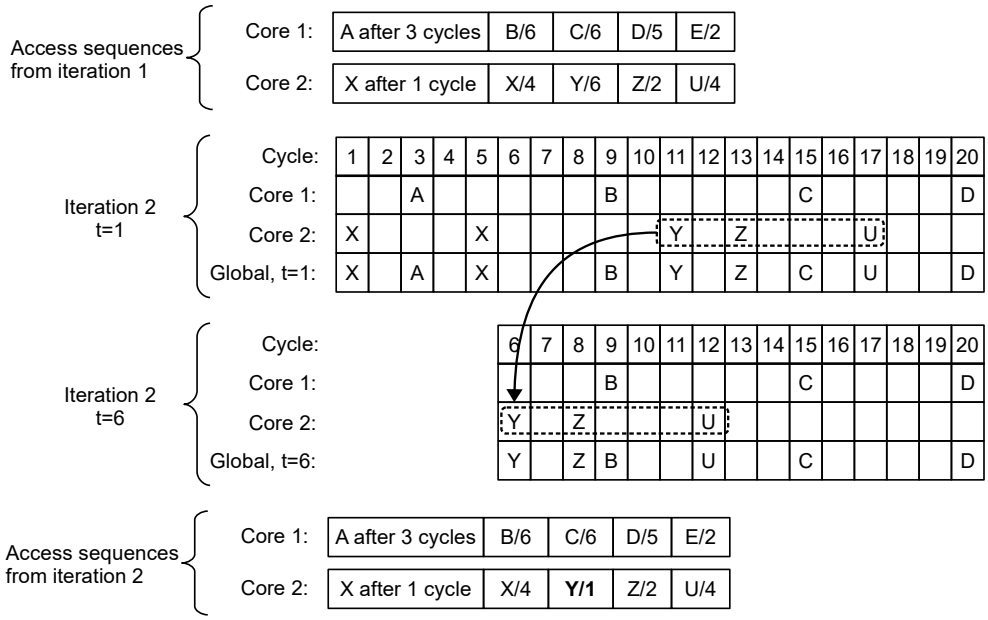


Fig. 3 An example of how NOPTb-miss works with sequences

During the simulation of iteration 2, at time t=1, the access to block X occurs. We assume that this access misses in the cache, and therefore NOPTb-miss is used to decide the replacement. The future global sequence is generated by interleaving the individual access sequences for the corresponding set (see “Iteration 2, t=1” in the figure). OPTb is applied to decide the replaced block based on this sequence (see “Global, t=1”). We suppose that in this case, OPTb decides to store X in the cache, a decision that in the future will convert the second access to X to a hit. As this first access to X misses as in the previous iteration, there are no changes in its service time. Therefore, future accesses will arrive as planned.

At time t=3, an access to A arrives. We assume it also misses. NOPTb-miss regenerates the global sequence from this cycle, which will coincide with the one generated at t=1 but starting in cycle 3, since there have been no changes with respect to the previous iteration.

At time t=5, the second access to X arrives, which in this case is a hit. NOPTb-miss does not apply since no replacement is required. The service time of this access to X is, however, less than that of the previous simulation, because it is served from the SLLC and not from DRAM. Therefore, core 2 experiences less delay and we assume it will be able to launch the next access to Y earlier than in the previous simulation.

At time t=6, an access to Y arrives, five cycles before the previous iteration. We assume it misses in the cache. NOPTb-miss regenerates the global sequence based on the current state (see “Iteration 2, t=6” in the

figure). Note that the order of accesses in the global sequence generated at time t=6 (see “Global, t=6”) differs from that generated at time t=1 (see “Global, t=1”). This is because the speed with which the individual sequence of each core is traversed changes from one iteration to another, when accesses change from misses to hits or vice versa.

During the simulation of iteration 2, new SLLC access sequences are generated and stored in new files. Fig. 3 shows, at the bottom, that in core 2 the distance between the second access to X and the access to Y has changed. These new sequences will be used in iteration 3.

3.3 Computational complexity

The computational time (*T)* required to execute NOPTb-miss depends mainly on the number of accesses of the workload that is being simulated. The order of magnitude of this figure for our simulated workloads (see section 4) ranges from tens of millions to hundredths of millions. It also depends on the number of cores and the cache associativity, but they are several orders of magnitude smaller and do not change unless the architecture changes. The number of iterations required is also a factor, but empirically we have found that it is a small figure (see section 5). We therefore consider the number of accesses of the workload as the input size of our algorithm (*n*).

The part of the algorithm that requires more computation is the construction of the global access sequence that happens on every miss. This task finishes when the first access for every block present in the set and for the incoming block are found in the future access sequences. The best-case scenario is when all accesses are found immediately. As this inexpensive task needs to be repeated on every miss, and the number of misses is proportional to the number of accesses, in this case

$$T_{best}(n) \in O(n) \quad (1)$$

The worst-case scenario happens when all accesses appear at the end of the individual sequences, forcing the construction of the whole global sequence on every miss. In this case, as we need to repeatedly build a list of *n*/2 items on average, we have

$$T_{worst}(n) \in O(n^2) \quad (2)$$

Although there is no known optimal algorithm for our scenario, it is a problem that can theoretically be solved by brute-force searching. On every miss, either one of the blocks present in the set is evicted or the incoming block is bypassed. The brute-force search would need to consider all situations, branching the simulation for every possible decision. At the end of all branches, the miss rate for every branch would be calculated and the minimum taken, which corresponds to the optimal solution. As branches are created on every miss, we have

$$T_{brute}(n) \in O(c^n) \quad (3)$$

where *c* is the associativity of the cache plus one, therefore greater than one.

Comparing (1) and (2) with (3), we can see that NOPTb-miss can be executed in quadratic time whereas the brute-force optimal solution requires exponential time.

4. METHODOLOGY

In this section, we describe the methodology used in the rest of the paper for evaluation purposes. First, we present the experimental framework, in Section 4.1, and details of our baseline system, in Section 4.2. Next, in Section 4.3, we briefly describe the metrics used in our results. Last, in Section 4.4, we offer information to facilitate the reproduction of our results.

4.1 The experimental framework

For our evaluation, we use a simulation framework based on the ChampSim simulator, designed for detailed memory subsystem research. This is the same as that used in the most recent Cache Replacement Championship (CRC2) [12]. It is a cycle-accurate trace-based simulator that models an out-of-order multicore processor and a memory hierarchy with three levels of cache, the last of which is shared.

Note that our proposals do not require this specific framework to work. They can be run on any cycle-accurate simulation engine, either based on the simulated execution of workloads or the reading of executed instructions from program traces.

We run a 4-core multiprogrammed workload set of applications from the SPEC CPU 2006 suite [47]. These are the same applications used in CRC2. The championship defines a “de facto” standard for the evaluation of cache replacement policies that we are reusing.

We have generated a set of 100 mixes, formed by random combinations of 4 benchmarks each, taken from all those in the SPEC CPU 2006 benchmark suite. Each program appears between 8 and 22 times, this corresponding to a mean of 13.8 times with a standard deviation of 4.0.

Traces for each benchmark are also taken from CRC2. For each individual program, the reference input is taken and the highest-weighted SimPoint [48, 49] of 1 billion instructions is traced. If during the simulation the end of a trace is reached, its execution continues from the beginning.

We first run at least 200 million instructions of each instruction trace to warm up the memory system, and then collect data for the next 800 million instructions. At the end of the data collection phase of a thread, the execution continues to put pressure on the shared resources, until data for all cores are collected.

Access streams and timings are captured during the whole execution for all cores. In each execution with NOPTb-miss, the simulation is extended beyond the data collection phase by up to 1000 million instructions, so that the access stream captured is long enough for all cores to not reach the end of it during the next execution.

4.2 Configuration of the baseline system

We model a base system of four superscalar processors with speculative out-of-order execution. Each processor has a 6-wide pipeline. A maximum of two loads and a maximum of one store can be issued every cycle. The reorder buffer has 256 entries and no scheduling restrictions. Branch prediction uses a simple gshare predictor [50]. Hardware prefetchers are disabled. All instructions have one-cycle latency except for memory accesses.

Each processor core has a two-level private cache hierarchy, the third and last level cache being shared by all the cores. The SLLC has a total size of 8 MB and is non-inclusive. Writeback bypassing for dirty blocks is not allowed, but read bypassing is (its usage depending on the policy).

The main memory is partially modelled including data bus contention, bank contention and write-to-read bus turnaround delays. The memory read queue is out-of-order and uses a modified Open Row FR-FCFS policy. Table 1 lists all the details of the memory hierarchy.

Table 1: Memory hierarchy parameters

ITLB / DTLB	4 KB,16-set, 8-way
2nd level TLB unified	96 KB, 128-set, 12-way
Private cache L1 I/D	32 KB, 8-way, LRU replacement, block size of 64 B, 4 cycles of access latency
Private cache L2 unified	256 KB, includes all L1 contents, 8-way, LRU replacement, block size of 64 B, 8 cycles of access latency
Shared cache L3 (SLLC)	8 MB (2 MB for single-core), non-inclusive, 16-way, block size of 64 B, 20 cycles of access latency, 32 demand MSHR
DRAM	DRAM core access latency: 13.5 ns on row hit, 40.5 ns on row miss Two 64-bit DRAM channels (one for single-core)

All previously mentioned parameters mirror the setup used in CRC2, and are representative of current CMP architectures.

Some metrics are normalized to single-core execution results. In such cases, we run each of the benchmarks in the same simulation environment, changing the core count to a single core and reducing the SLLC size to a quarter of its original size.

4.3 Metrics

To evaluate performance, we report SLLC misses per kilo instruction (MPKI) and number of instructions executed by all cores per cycle (system IPC). The former is a performance metric for the SLLC alone and the

latter is for the whole system. Unless stated otherwise, figures show the (arithmetic) mean of the results obtained for each of the 100 workloads.

To evaluate fairness, we use the M_1 (un)fairness metric defined by Kim et al. [51]. This is a pure (un)fairness metric that is independent of performance level, unlike mixed metrics such as harmonic IPC [52]. For each mix, the unfairness is calculated as

$$M_1 = \sum_i \sum_j \left(\left| \frac{Miss\_shr_i}{Miss\_ded_i} - \frac{Miss\_shr_j}{Miss\_ded_j} \right| \right) \quad (4)$$

for any pair i and j of cores executing in the workload (six combinations for our four cores). $Miss\_shr$ is the miss count in the shared cache scenario and $Miss\_ded$ is the miss count in the dedicated scenario, in which a program runs in a single-core setup with a proportionally smaller SLLC (a quarter of the original size). Notice that M_1=0 means complete fairness.

4.4 Other replacement policies

Throughout the evaluation we use other replacement policies, either as part of the discussion or in our comparisons:

- The RANDOM policy evicts a block randomly selected among those stored in the set
- The MISSES policy forces all blocks coming from DRAM to bypass the SLLC, and all write-backs to be allocated to one specific way. Its goal is to force as much misses in the SLLC as possible.
- The SRRIP policy [14] (Static Re-Reference Interval Prediction) utilizes the concept of a re-reference interval prediction value (RRPV) for a cache block. The version we use stores this value in two bits. On insertion, a block gets assigned a *long* re-reference interval, with RRPV=2. If the block receives a hit, it is promoted to a *near-immediate* re-reference interval, with RRPV=0. In case an eviction is required, the lowest-order block with *distant* re-reference interval, RRPV=3, is selected. If there is no block like that in the cache set, the RRPV of all blocks is incremented, and the victim selection restarts. In summary, SRRIP assigns lower RRPV, and therefore less likelihood of replacement, to blocks that have been recently re-referenced.
- The LRU policy (Least Recently Used) maintains the stack of blocks in a set ordered by its recency of usage. It selects the least recently requested block for eviction.
- The ReD policy [31, 53] (Reuse Detector) decides, based on the detection of a previous reuse, whether a block coming from main memory is inserted, or not, in the SLLC. Blocks bypass the cache unless their expected reuse behavior matches specific requirements. In general, they are only stored in the SLLC the second time they are requested in a limited time window. To detect this repetition, it remembers the addresses of previously requested blocks in a Reuse Detector, also called Address Reuse Table. Secondarily, some blocks enter the cache on the first request if their requesting instruction has shown to request highly reused blocks in the past. This information is stored in the Program Counter Reuse Table. Once the decision to enter the cache is taken, the policy mirrors SRRIP.
- The SHiP++ policy [54] (Signature-based Hit Prediction plus plus) is an evolution of SHiP-PC [30]. SHiP-PC tries to predict the reuse characteristics of a line based on a signature of the line, that is defined by the Program Counter (PC) of the instruction that caused the miss. A data structure called the Signature Hit Predictor stores SLLC hit/miss data correlated to this signature. SHiP leverages the RRPV concept of SRRIP but it assigns values on insertion in a different way: a block is inserted with *long* re-reference interval only if other blocks associated with its same signature have previously hit in the SLLC, and with *distant* interval if not. The policy mirrors SRRIP in other areas. SHiP++ extends SHiP by improving the management of the Signature Hit Predictor and making the policy prefetch and writeback-aware.
- The Hawkeye policy [32, 55] learns from Belady's MIN algorithm [33] by applying it to past cache accesses to inform future cache replacement decisions. It introduces a new method of efficiently simulating Belady's behavior on past accesses and stores the expected hit or miss behavior of a block correlated with the PC of the instruction that requested the access. Thereby PCs are classified as cache-friendly or cache-averse and this information is used to assign the replacement priority in the SLLC.

4.5 Reproducibility

All resources required to reproduce this work are available in a public repository [56]. These include:

- The version of the ChampSim simulator we use (the original version is being evolved [57])
- The composition of our workload set

- Pseudo-code for NOPTb-miss
- The source code for NOPTb-miss, NOPTb-fair and the other policies with which they are compared
- Sample execution scripts.

Additionally, the traces we use are also publicly available [58].

5. NOPTB-MISS EVALUATION

In this section, we first study NOPTb-miss convergence, showing the evolution of the SLLC miss rate as more iterations are run. Next, we analyze how near our proposal is to the optimum. Finally, we present results for each application in our workload.

5.1 Convergence analysis

In step 1, NOPTb-miss can model an SLLC with any replacement policy. To validate our design and hypothesis in different circumstances, we select three different initial SLLC policies: SRRIP [14], RANDOM and a policy that misses in the SLLC as much as possible, which we call “MISSES”. These three initial policies are used in three isolated sets of experiments.

Our proposal is based on the hypothesis that the global sequence obtained in an iteration with NOPTb-miss is closer to what an optimal algorithm would produce than the one obtained in the previous iteration. This implies that the miss rate is reduced in each iteration. This hypothesis can be verified by measuring the miss rate, which should decrease asymptotically in each iteration, heading towards the optimum value. This value is expected to be independent of the starting replacement policy.

Fig. 4 shows the mean SLLC miss rate for successive iterations of the simulation of NOPTb-miss, for all three initial replacement policies. Iteration 0 corresponds to step 1 of the algorithm, while the rest correspond to step 3. The chart on the right is a zoom of the chart on the left: iteration 0 has been removed from the X-axis and the range of the Y-axis decreased (the results being the same as those in the corresponding region of the chart on the left). As shown on the left, miss rates converge rapidly, as soon as the first iteration, to a very similar value regardless of the replacement policy used in iteration 0. On the right, we can see how successive iterations improve the miss rate convergently. The miss rates in iteration 4 differ by less than 0.1% between the policies.

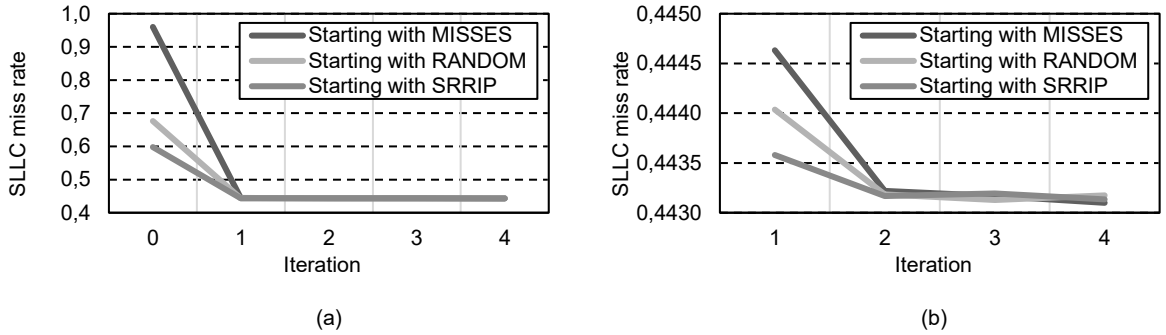


Fig. 4 Mean SLLC miss rate for step 1 of NOPTb-miss (labelled as iteration 0) and several iterations of step 3. The two graphs show the same values for iterations 1-4 but using different scales; (a) shows all iterations while (b) is a zoom showing the iterations in which the convergence can be seen most clearly

The aforementioned values are means of the rates with the different 100 workloads. Fig. 5 shows, for each workload, the maximum difference in miss rate obtained in iteration 4 between any two of the three experiments. In the worst case, the miss rates differ by 0.50%. In 96 of the 100 workloads, the difference is less than 0.1%.

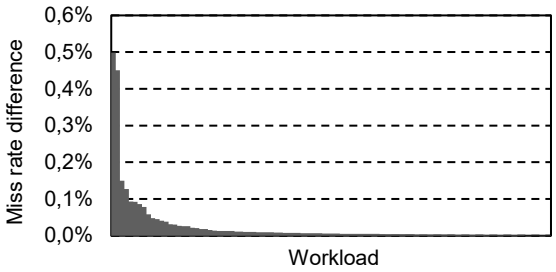


Fig. 5 Maximum relative difference in miss rate in iteration 4 of NOPTb-miss when starting with MISSES, RANDOM and SRRIP as replacement policies in step 1 (iteration 0)

In our experiments, we have stopped our simulations in iteration 4 since the results are very similar to those obtained in iteration 3. In 55% of workloads and experiments, the miss rate is lower than in iteration 3, and in the rest it is higher. In iteration 4 and later (not shown), the miss rate does not converge any more, but fluctuates slightly. According to our analysis, the main reason for this is that the simulation framework models an out-of-order processor. The unordered execution sometimes results in accesses from the same core in successive iterations not occurring in strictly the same order. The most common case is that two consecutive accesses exchange their order in the sequence. These accesses must be very close in time. Since the individual access sequence is not completely reliable, the OPTb policy generates sub-optimal replacements, which cause small variations in the result.

In summary, NOPTb-miss converges quickly, after a few iterations, to a miss rate that differs less than 0.1% for all the various initial replacement policies selected for step 1. Where not stated otherwise, in the rest of this paper, we present the results obtained in iteration 4 when using SRRIP as the initial policy.

Although our experimental results show that the algorithm converges for all 100 mixes in our workload, more theoretical analysis of our proposal would be required to mathematically guarantee convergence in all cases. This interesting analysis is outside of the scope of this paper.

5.2 How close is NOPTb-miss to the optimum?

Since there is no optimal policy for miss rate for shared caches, we believe it is not possible to demonstrate mathematically that NOPTb-miss provides the same results as those that would be obtained with such a policy. That said, there is strong evidence that NOPTb-miss obtains results that, within the indicated margins, are indistinguishable from such a theoretical policy.

On the one hand, as we mentioned at the beginning, a theoretical way to minimize the miss rate in the SLLC would be to apply OPTb using the global sequence resulting from the execution of the workload using OPTb as a replacement algorithm. From this point of view, the optimal algorithm can be understood as the one that generates results and a global sequence such that, when applying OPTb again, these results are not modified. As we have seen, after several iterations of NOPTb-miss the results show very little variation, which in our opinion indicates that it is very close to the optimum.

On the other hand, it could be argued that with NOPTb-miss, in some cases, local minima are reached from which the algorithm is not able to escape by continuing to iterate. We believe that the convergence towards a very similar miss rate starting with three very different replacement policies makes this very unlikely. In addition, we have carried out experiments introducing variations in the timing of the access sequences. With this, the goal is to introduce disturbances in input data in different directions, to force the algorithm out of a hypothetical local minimum. Specifically, for a subset of the workloads, we run several different experiments in iteration 5, where the times between consecutive addresses in three of the cores during the whole simulation were multiplied by 0.5, 1, 3 or 9, while those of the other core were left unchanged. All 175 combinations were tested for each workload in the subset. The results in MPKI and IPC worsened for that iteration, as replacement decisions were misinformed and therefore not optimal, but quickly returned in iteration 6 to values within the same ranges as those obtained before the disturbance. These results provide strong evidence that the minima reached are not local.

5.3 Results per application

In this section, we compare the results obtained by each of the applications when using NOPTb-miss as the replacement policy. Fig. 6 shows, on the top chart, the normalized MPKI obtained with NOPTb-miss for all individual benchmarks. Values are normalized using the MPKI obtained when executing each application in a single-core system using OPTb as the replacement policy, with an LLC with a size proportionally reduced

(from 8 MB for 4 cores to 2 MB for a single core). The vertical bar shows the range of results for all instances where the application appears in our workload set and the horizontal bar crossing the vertical one indicates the mean. Applications are ranked by the number of requests for accesses per kilocycle (APKC) they make to the SLLC, shown on the bottom chart.

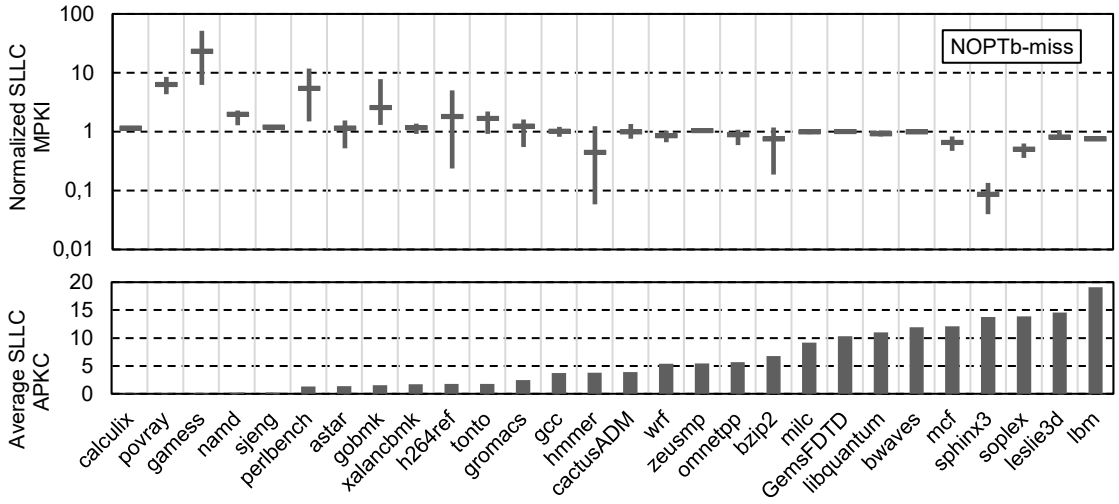


Fig. 6 Bottom: Mean number of accesses to the SLLC per kilocycle (APKC) for the SPEC CPU [47] applications that compose our workload set, ranked by value. Top: MPKI at the SLLC for NOPTb-miss, normalized to the value obtained when executing each application in a single-core system with a proportionally reduced SLLC size and OPTb. The vertical bar shows the range of results for all instances where the application appears in our workload set. The horizontal bar shows the mean

The figure shows that the applications on the right, those with the highest rate of access to the SLLC, generally achieve MPKI reductions when running alongside others in a shared cache. Conversely, the applications on the left, those with lower access rates, experience an increase in miss rate compared to that in their optimal solo execution. This result can be explained by the characteristics of the policy. To minimize miss rate, NOPTb-miss attempts to reconstruct the global sequence of references as it will occur in the future. Cores that launch more requests per cycle place relatively more blocks in that global sequence than those that launch fewer. As an optimal global miss rate is achieved retaining blocks accessed soonest in the future, and those are mostly from applications with high access rate, cores that access the cache less frequently have less priority, this meaning that NOPTb-miss does not favor them.

A fair replacement for a shared cache should provide similar MPKI reduction to all applications. Therefore, we can conclude that NOPTb-miss does not treat applications fairly. Specifically, it seems that NOPTb-miss at least partially sacrifices fair service between cores to minimize the global miss rate of the shared cache.

6. A NEAR-OPTIMAL REPLACEMENT ALGORITHM TO MAXIMIZE FAIRNESS

As mentioned before, there is no single definition of an optimal replacement policy in a shared cache because the optimization of the total number of cache misses does not imply the optimization of other metrics such as global throughput (total number of instructions executed) or fairness between cores.

We can, therefore, propose the design of a NOPT with a different objective, seeking to approximate a different optimum. Considering the analysis presented in the previous section, our new objective is to treat all cores fairly, improving fairness among them. We call our proposed algorithm NOPTb-fair, standing for near-optimal replacement algorithm with bypass to maximize fairness.

Fairness itself is a desirable property but it is not a common main goal in real-world design. Our goals with this design and its analysis are to explore how fairness affects other metrics such as throughput, to show the potential shortcomings that may appear with a policy only looking to minimize miss rate and to ultimately help achieve a better balance between throughput and fairness in real-life policies.

6.1 NOPTb-fair design

NOPTb-miss decides the replacement based on the time it will take to reference each block in the global sequence, regardless of which core will reference it. Nonetheless, for a particular core, only the blocks that it

will reference in the future and their order of reference are relevant. That is, its own sequence of future references is the relevant one, regardless of what the other cores do.

Let us suppose that, at some point, the first references that appear in the future sequences of cores C1 and C2 are for blocks B1 and B2 respectively. The most important block for core C1 is B1, and, similarly, that for C2 is B2, and this does not depend on whether B1 is accessed later than B2 or vice versa. Our proposal is based on respecting these individual priorities of each core.

We define the future reuse distance (FRD) of a block B, used by a particular core, as the number of accesses that appear in the individual access sequence of that core from the current time until the next reference to block B. FRD indicates the relative importance of each block to the core that uses it: the lower the FRD, the more important it is to keep the block in the SLLC for the core that uses it. Our proposal for NOPTb-fair replaces the block with the largest FRD of any of the cores. Therefore, it determines the priority of each block from the point of view of the core that uses it, instead of from that of the shared cache, and replaces the least important of them. It is interesting to note that OPTb for a single core also replaces the block with the largest FRD.

We propose the following procedure for NOPTb-fair:

1. Simulate the workload, using any replacement policy in the SLLC.
2. For each execution core, store its individual access stream to the SLLC in files. Timing information is not required.
3. Simulate the workload using a replacement policy that evicts the block with the largest FRD, also considering the possibility of a bypass. This policy is fed with the individual access sequences captured in the previous step. It also uses pointers in each sequence to track the last access received in the SLLC from each core.

The result does not depend on the policy used in the initial step, within the limitations caused by the simulation of out-of-order processors (see Section 5.1); and nor is it necessary to perform an iterative process. This is because the SLLC replacement policy affects the timing, but not the list or order of accesses of each core, which is the only information used by NOPTb-fair.

A complementary view on NOPTb-fair appears when we realize that its results are equivalent to applying OPTb on a global sequence constructed with the (unrealistic) assumption that the SLLC will receive the same number of accesses per cycle from each core, instead of using the actual expected timings as NOPTb-miss does.

6.2 NOPTb-fair example

In this section, we illustrate the operation of NOPTb-fair with an example of a cache shared by two cores. Fig. 7 shows, for a specific time, the content of a cache set and the sequences of future accesses of the two cores to that cache set. These sequences have been obtained in a previous simulation and have no temporal information. Therefore, they could have been generated using any replacement algorithm in the SLLC.

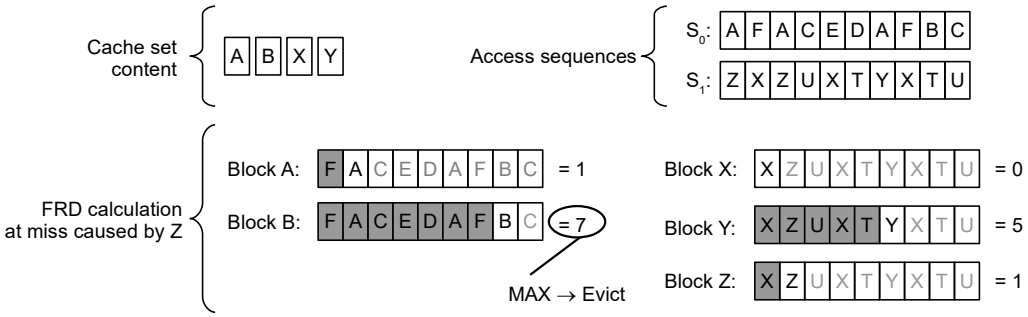


Fig. 7 An example of the victim selection procedure of NOPTb-fair. Top: cache set contents and access sequences for a 4-way cache in a 2-core system. Bottom: FRD calculation at miss caused by Z, assuming A hits before. Each FRD calculation searches only in the sequence of the affected core. Block B is evicted because it has the largest FRD.

Suppose that the first access to reach the SLLC is to block A from core 0. It is a hit and therefore there is no replacement action. Then, the access to Z from core 1 arrives at the SLLC, this causing a miss and therefore requiring a replacement action. At that time, NOPTb-fair calculates FRD for each of the blocks in the set and the new block Z. FRD for block B is 7 since core 0 will access blocks F, A, C, E, D, A and F before block B. Similarly, the FRD is 1 for A (block F), 0 for X, 5 for Y (blocks X, Z, U, X and T), and finally, 1 for Z (block X). The largest FRD among the five candidate blocks is that of block B, which will therefore be evicted to make room for Z.

7. NOPTb-FAIR EVALUATION

In this section, we first present the per-application performance results for NOPTb-fair. Next, we analyze how near our proposal is to the optimum. Finally, we compare results for NOPTb-fair and NOPTb-miss per workload.

7.1 Results per application

Fig. 8 shows the normalized MPKI obtained with NOPTb-fair for all individual benchmarks. The chart organization is the same as in Fig. 6 (above). Comparing the figures, we can see that for NOPTb-fair there is no obvious correlation between the normalized SLLC MPKI and the SLLC APKC. With NOPTb-fair, only two applications (perlbench and gobmk) show a significantly higher mean MPKI than the single-core optimal execution, and the increase is much smaller than that observed with NOPTb-miss. Therefore, we conclude that NOPTb-fair does produce a fairer result than NOPTb-miss.

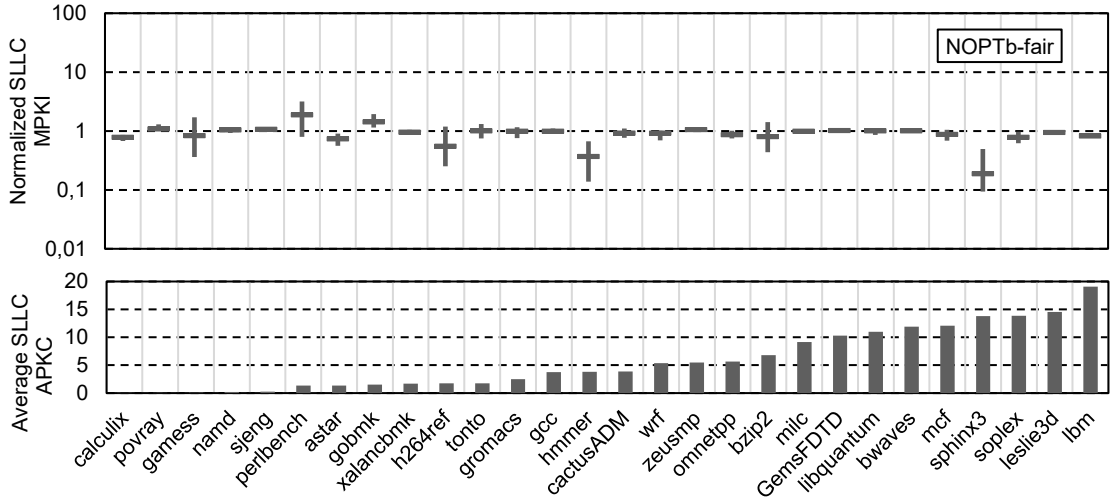


Fig. 8 Bottom: Mean number of accesses to the SLLC per kilocycle (APKC) for all applications in our workload set, ranked by value. Top: MPKI at the SLLC for NOPTb-fair for the same applications, normalized to the value obtained when executing each application in a single-core system with a proportionally reduced SLLC size. The vertical bar shows the range of results for all instances where the application appears in our workload set. The horizontal bar shows the mean

7.2 How close is NOPTb-fair to the optimum?

To the best of our knowledge, there is no previous proposal for a replacement algorithm that optimizes fairness between cores in an SLLC. Therefore, we cannot compare the NOPTb-fair design or results against other optimal or near-optimal proposals.

In Section 8, we compare NOPTb-fair fairness results with those from other replacement policies. NOPTb-fair obtains a better fairness metric than any other policy. Beyond this, and unlike the case of NOPTb-miss, we are not able to provide strong evidence that its results are optimal or near optimal. We consider NOPTb-fair as best approximation to the theoretical optimal fairness policy to date.

7.3 Comparison between NOPTb-miss and NOPTb-fair at workload level

Having two policies whose optimization goals are different allows us to compare them and study how the goals affect the final performance of the system. In this section, we compare NOPTb-miss and NOPTb-fair, and analyze their performance for each workload.

Fig. 9 shows a comparison of NOPTb-miss and NOPTb-fair at workload level. The chart on the left shows the differences in normalized SLLC MPKI between the two policies for each of the workloads executed, ranked by this difference in descending order. Positive values mean that NOPTb-miss misses less often (performs better) than NOPTb-fair. As expected, NOPTb-miss reduces the SLLC MPKI more than NOPTb-fair for all workloads in our set.

The chart on the right shows the differences in normalized IPC, with workloads ranked by this difference in descending order. Positive values mean that NOPTb-miss increases IPC more (performs better) than NOPTb-fair. We can see that the higher reductions in MPKI observed with NOPTb-miss do not always translate into a better throughput: 34 out of the 100 multiprogrammed workloads show better normalized IPC for NOPTb-fair than for NOPTb-miss.

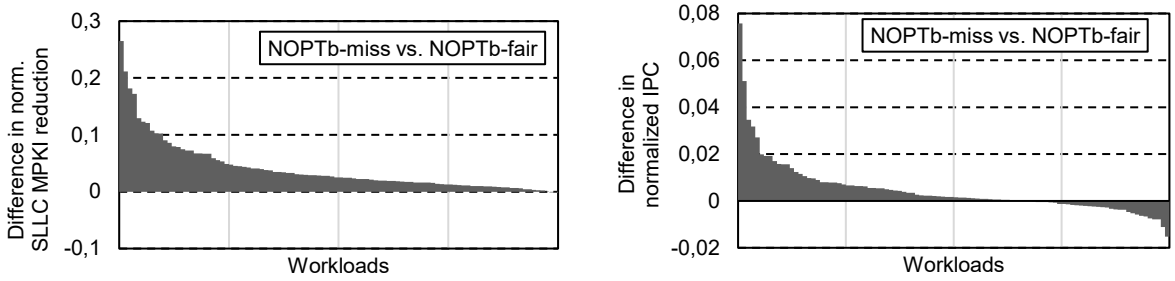


Fig. 9 (a) Difference in normalized SLLC MPKI reduction and (b) difference in normalized IPC between NOPTb-miss and NOPTb-fair for all workloads. A positive value indicates that NOPTb-miss has a higher score for the base metric. Workloads are ranked by value in descendent order (i.e., the workload order differs between the two charts)

It seems counterintuitive that the policy optimized for fairness should achieve better IPC than the one that yields a better miss rate, as is observed in several workloads. Although we have already noted in our introduction that, in shared caches, optimizing miss rate does not imply optimizing other performance metrics, it is not obvious why greater fairness can, in some cases, lead to this effect.

Applications with fewer accesses to the SLLC tend to have higher IPC than applications that have more accesses. The former access less data or they do it more from the private caches, and those are faster than the SLLC and much faster than main memory. Although the penalty for an SLLC miss is similar in all cases when measured in cycles, the equivalent number of instructions that cannot be executed during the miss penalty is higher in applications with low SLLC APKC. In other words, eliminating a miss in a low-APKC/high-IPC application increases the number of instructions executed per cycle more than eliminating a miss in a high-APKC/low-IPC application.

As NOPTb-miss favors applications with higher APKC and lower IPC, it reduces miss rate but does not increase throughput by the same proportion. This mechanism explains why, in certain workloads, the balanced behavior of NOPTb-fair achieves better global IPC than NOPTb-miss.

8. COMPARISON WITH STATE-OF-THE-ART POLICIES

One of the utilities of having an optimal or near-optimal policy is to compare it against the best existing online policies, which do not require knowledge of the future. In Sections 8.1 and 8.2, we compare the results of various online replacement policies, in terms of performance and fairness respectively, against the two offline near-optimal algorithms presented previously. Following this analysis, in Section 8.3, we propose new design guidelines for future replacement algorithms.

8.1 MPKI and throughput comparison with state-of-the-art policies

The left chart of Fig. 10 shows the percentage of MPKI reduction achieved by several replacement policies when taking random replacement as the baseline. We consider two basic policies, LRU and SRRIP [14], the three best ones from the CRC2, namely, ReD [31, 53], SHiP++ [30, 54], and Hawkeye [32, 36], and finally, NOPTb-miss and NOPTb-fair.

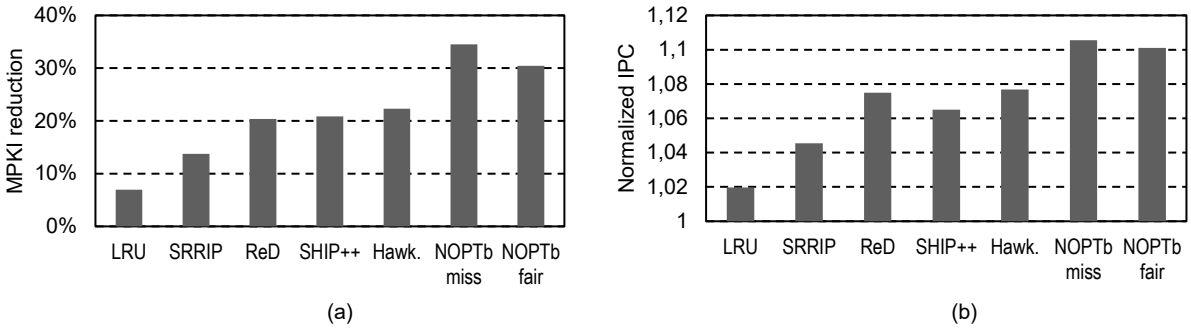


Fig. 10 (a) MPKI reduction in the SLLC and (b) normalized IPC, relative to random replacement

On average, NOPTb-miss results in an MPKI 35% lower than that obtained with random replacement. The winning algorithms from the CRC2 provide an MPKI between 20 and 22% lower than that obtained with random replacement. It is interesting to note that the differences between them are small, although their designs are very different. The reduction obtained by the three CRC2 winners corresponds to around 65% of the reduction obtained by NOPTb-miss.

Fig. 10 also shows, on the right, the normalized IPC (versus random) with those same policies. The result is similar to that obtained for MPKI. On average, with NOPTb-miss, the IPC is 10.6% higher than with random replacement, while the winning algorithms of the last replacement competition achieved an IPC between 6.5 and 7.7% higher than that observed with random replacement.

This means that the best policies have already achieved around 75% of the maximum improvement possible in IPC. With this average figure we do not want to discourage replacement research since, on the one hand, significant improvements may be obtained for some applications and, on the other, the relative importance of cache memory tends to increase over time [59].

8.2 Fairness comparison with state-of-the-art policies

In this section, we present the results of state-of-the-art policies, together with those of NOPTb-miss and NOPTb-fair, focusing on fairness. Fig. 11 shows the unfairness score (see Section 4.3) for the same set of SLLC replacement policies as before.

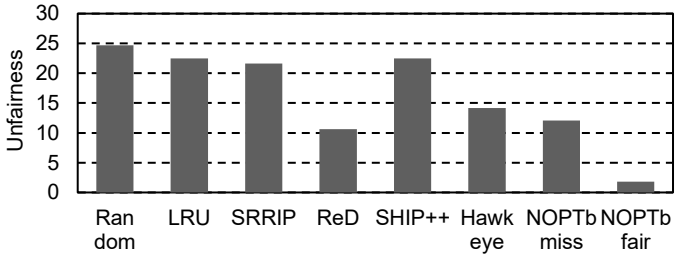


Fig. 11 Unfairness for various SLLC replacement policies. A lower score means better fairness

As expected, NOPTb-fair achieves the lowest and therefore best value, specifically, an unfairness 85% lower than that achieved with NOPTb-miss.

Among the online policies, Random and LRU result in the worst fairness. This is because they do not use any mechanism to prevent an application with a high access rate from taking up excessive space in the SLLC. The worst case occurs with applications that heavily access with a scanning or thrashing pattern [14], since these do not take advantage of the SLLC and prevent other applications from doing so.

The next policies, in the ranking by unfairness, are SRRIP and SHiP++, which show intermediate values. Both have mechanisms to avoid problems with scanning or thrashing patterns; specifically, they do not allow blocks to have high priorities unless they detect reuse in the SLLC. Nonetheless, they do not prevent some applications that access heavily from taking up excessive space. Next in the ranking is Hawkeye which, imitating OPT, avoids thrashing and scanning, and respects the space of each application if that proves useful in reducing the overall miss rate.

The best result among online policies is obtained by ReD. On the one hand, it protects against scanning and thrashing because it bypasses blocks without reuse. On the other, it protects applications with a relatively lower access rate because it uses a private reuse detector for each core, separated from the SLLC.

8.3 Balancing miss-rate reduction and fairness in SLLC replacement policies

Any design that attempts to minimize miss rate may resort to favoring specific applications or application types that are “easy targets”, getting unfair results. As we have seen in Section 7.3, a fair replacement policy like NOPTb-fair manages to convert the reductions in miss ratios to IPC increases better than an unfair policy like NOPTb-miss. On the other hand, we have seen in Fig. 11 that state-of-the-art replacement policies achieve even worse results in terms of fairness than NOPTb-miss. In view of these findings, we conjecture that fairness also affects throughput for state-of-the-art policies. If this is the case, the same effect of an advantage in IPC associated with better fairness should be also noticeable comparing two of these policies.

Fig. 10 shows that MPKI reduction is 20.4% for ReD and 20.9% for SHiP++, higher for SHiP++, while the normalized IPC is 1.075 for ReD and 1.065 for SHiP++, higher for ReD. This is in line with the worse unfairness obtained by SHiP++ shown in Fig. 11: 22.5 with SHiP++ and 10.6 with ReD. Similar trends are seen comparing other policies, like Hawkeye with SHiP++ and with ReD.

We conclude that, in shared caches running heterogeneous workloads, higher system IPC values are obtained when considering fairness together with miss rate reduction when designing the replacement policy. As inter-core fairness is a concept that has no meaning in single-core systems, this conclusion suggests a replacement policy designed for private caches should not be transferred directly to a shared cache, even if the private cache is a last-level cache. Such an approach might result in a policy with relatively high unfairness between cores, that benefits only a subset of applications and does not achieve, on average, an increase in throughput proportional to the decrease in the number of misses. A quantitative analysis of its fairness and per-application performance, using NOPTb-fair as a reference, may provide hints on how to improve the design to avoid such problems.

9. SENSITIVITY ANALYSIS

As the algorithms presented above are for shared caches in multicore processors, the number of cores is one of the most important parameters of the setup. In this section, we evaluate the sensitivity of these algorithms to the number of cores in the system by using 8 cores, twice the number used before. The size of the SLLC has also been doubled to 16 MB. Our workload consists of 10 mixes, a smaller set due to computing power constraints.

First, we analyze the convergence of NOPTb-miss using this new setup. Then, we present performance results for NOPTb-miss and NOPTb-fair and compare them with online policies. Finally, we report fairness results.

9.1 Convergence analysis of NOPTb-miss

Fig. 12 shows the mean SLLC miss rate after several iterations of NOPTb-miss, with the 8-core setup. Comparing these results with the 4-core results in Fig. 4, we see that the convergence is slightly slower in iteration 1, but results are similar after iteration 3 (<0.1% variation).

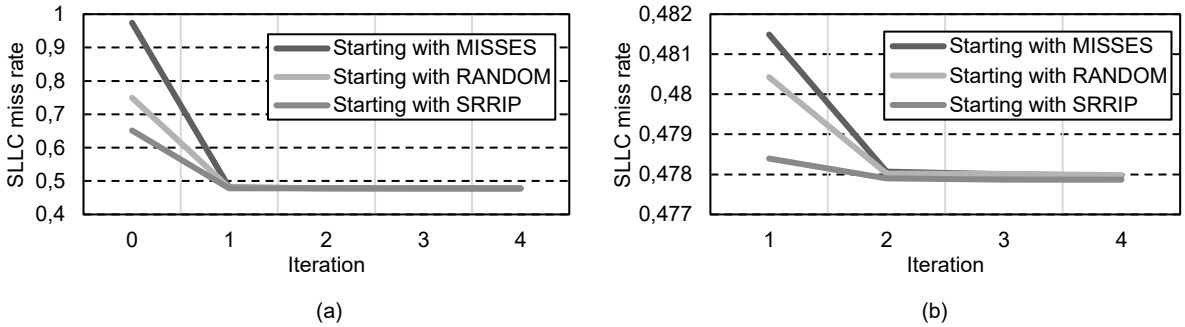
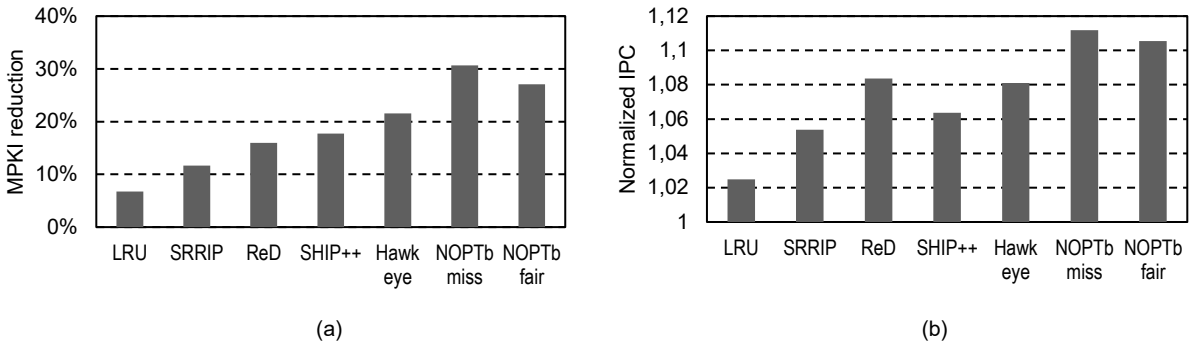


Fig. 12 Mean SLLC miss rate for step 1 of NOPTb-miss (called iteration 0) and several iterations of step 3, using an 8-core setup. The two charts show the same values for iterations 1-4 but using different scales; (a) shows all iterations while (b) is a zoom showing the iterations in which the convergence can be seen most clearly

We conclude that the number of cores does not significantly affect the convergence of NOPTb-miss.

9.2 MPKI and throughput

Fig. 13 shows the percentage of MPKI reduction and the normalized IPC achieved by our selected set of replacement policies when taking random replacement as the baseline, using the 8-core setup.



(a) (b)

Fig. 13 (a) MPKI reduction in the SLLC, and (b) normalized IPC, relative to random replacement, using an 8-core setup

NOPTb-miss also outperforms all other policies in both metrics with 8 cores, as expected. Comparing these results with the 4-core results in Fig. 10, we can see that the gap with respect to the best online policies is similar. The best online policy is Hawkeye in terms of MPKI reduction, as with 4 cores. Nonetheless, when considering global IPC, the best online policy with 8 cores is ReD.

9.3 Fairness

Fig. 14 shows the unfairness score for the same set of SLLC replacement policies as before, using the 8-core setup.

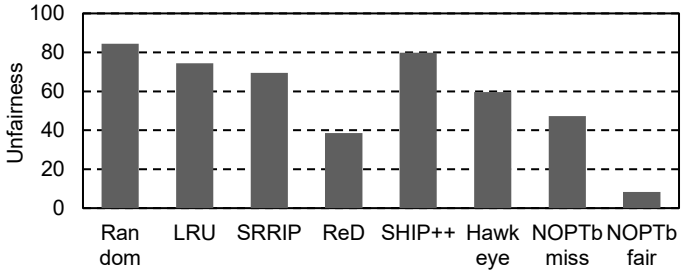


Fig. 14 Unfairness for various SLLC replacement policies, using an 8-core setup. A lower score means better fairness

NOPTb-fair also achieves the best fairness with 8 cores, as expected. Comparing these results with the 4-core results in Fig. 11, we can see that the gap with respect to the best online policies is larger, and the differences between online policies are also larger, especially between ReD and Hawkeye. This larger fairness advantage for ReD explains why this policy obtains the best IPC results with 8 cores, outperforming Hawkeye despite its MPKI disadvantage. This is consistent with our analysis in Section 8.3.

10. CONCLUSIONS

The replacement algorithm is a key piece in the design of cache memories. An optimal algorithm that minimizes the miss rate in a private cache has been available for several decades. This algorithm can be run offline on a simulator, to provide reference values and insights into design problems. There is no equivalent algorithm, however, for shared caches, which are accessed by several processors or cores.

This paper presents a novel near-optimal offline algorithm for minimizing the miss rate in a shared cache. We propose approaching the optimal execution and the optimal global access sequence iteratively. During each execution, the future access sequence that the shared cache will receive is reconstructed on every miss, interleaving the future individual per-core access sequences, taken from the previous execution. This reconstructed future access sequence feeds an OPTb policy, which decides to bypass the requested block or evict the block to be accessed further into the future. We evaluate NOPTb-miss running on an SLLC and show

that it converges after a few iterations to a near-optimal miss rate that is independent of the initial conditions, within a margin of 0.1%.

In a shared cache, the optimization of the total number of cache misses does not imply the optimization of other metrics like fairness or throughput. We show that running a multiprogrammed workload, NOPTb-miss benefits applications that access the SLLC more than others and is therefore not optimal in that regard. We then propose a new near-optimal offline algorithm whose goal is to maximize fairness while maintaining a low miss rate. NOPTb-fair achieves the best fairness among all replacement policies considered. It even achieves a better overall throughput than NOPTb-miss in 34% of the workloads, due to its better fairness, despite always having a worse miss rate.

Our near-optimal proposals can be used to gain insights into state-of-the-art online policies. Comparing performance results, we show that the best online policies achieve around 65% of the MPKI reduction obtained by NOPTb-miss and 75% of the throughput improvement (vs. random). We demonstrate that our results are maintained when doubling the number of cores in the system. Comparing fairness, the best state-of-the-art policy achieves 60% of the improvement seen with our near-optimal policy, and the second-best only 45%. The gaps increase when doubling the number of cores. Analyzing performance and fairness together we show that, when designing a replacement policy for shared caches, higher system IPC values are obtained when seeking to achieve fairness in addition to miss rate reduction.

ACKNOWLEDGEMENTS

We thank the anonymous referees for their valuable comments to improve our paper. The authors acknowledge support from grants (1) PID2019-105660RB-C21 and PID2019-107255GB-C22 from Agencia Estatal de Investigación (AEI) from Spain and European Regional Development Fund (ERDF); (2) gaZ: T58_20R research group from Aragón Government and European Social Fund (ESF); and (3) 2014-2020 "Construyendo Europa desde Aragón" from European Regional Development Fund (ERDF). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

REFERENCES

[1] Smith AJ (1982) Cache memories. ACM Comput Surv (CSUR) 14(3):473-530

[2] Mittal S (2016) A survey of cache bypassing techniques. J of Low Power Electron and Appl 6(2):5

[3] Baer J (2009) Microprocessor architecture: from simple pipelines to chip multiprocessors. Cambridge University Press.

[4] Balasubramonian R, Jouppi NP, Muralimanohar N (2011) Multi-core cache hierarchies. Synth Lect on Comput Archit 6(3):1-153

[5] Strohmaier E, Dongarra J, Simon H, Meuer M, Meuer H (2021). Top 500, the list. https://www.top500.org. Accessed 25 February 2021

[6] Yoshida T (2018) Fujitsu High Performance CPU for the Post-K Computer. IEEE hot chips 30

[7] Sadasivam SK, Thompto BW, Kalla R, Starke WJ (2017) IBM Power9 processor architecture. IEEE Micro, 37(2):40-51

[8] Lepak K, Talbot G, White S, Beck N, Naffziger S, Fellow S (2017) The next generation AMD enterprise server product architecture. IEEE hot chips 29.

[9] Arafa M, Fahim B, Kottapalli S, Akhilesh K, Looi LP, Mandava S, Rudoff A, Steiner IM, Valentine B, Vedaraman G, Vora S (2019) Cascade lake: Next generation Intel Xeon scalable processor. IEEE Micro 39(2):29-36

[10] Jain A, Lin C (2019) Cache Replacement Policies. Synth Lect on Comput Archit 14(1):1-87

[11] Various authors (2010) Proc of the 1st JILP Workshop on Comput Archit Compet

[12] Various authors (2017) 2nd Cache Replacement Championship. http://crc2.ece.tamu.edu. Accessed 3 March 2021

[13] Mattson RL, Gecsei J, Slutz DR, Traiger IL (1970) Evaluation techniques for storage hierarchies. IBM Syst J 9(2):78-117

[14] Jaleel KB, Theobald SC, Steely Jr A, Emer J (2010). High performance cache replacement using re-reference interval prediction (RRIP). In: Proc of the 37th Int Symp on Comput Archit (ISCA), 2010. IEEE, pp. 60–71

[15] Qureshi MK, Jaleel A, Patt YN, Steely S, Emer J (2007) Adaptive insertion policies for high performance caching. In: Proc of the 34th Ann Int Symp On Comput Archit (ISCA), 2007. IEEE, pp. 381–391

[16] Duong N, Zhao D, Kim T, Cammarota R, Valero M, Veidenbaum A (2012) Improving cache management policies using dynamic reuse distances. In: 45th Annu IEEE/ACM Int Symp on Microarchit. IEEE, pp. 389–400

[17] Lee D, Choi J, Kim J H (2001) LRFU: A Spectrum of policies that subsumes the least recently used and least frequently used policies. IEEE Trans on Comput 50(12):1352-1361

[18] Lim H, Kim J, Chong J (2010) A cache replacement policy to reduce cache miss rate for multiprocessor architecture. IEICE Electron Express 7(12):850-855

[19] Hu Z, Kaxiras S, Martonosi M (2002) Timekeeping in the memory system: Predicting and optimizing memory behavior. In: Proc of the 29th Annu Int Symp on Comput Archit. IEEE, pp. 209–220

[20] Vakil-Ghahani A, Mahdizadeh-Shahri S, Lotfi-Namin M, Bakhshalipour M, Lotfi-Kamran P, Sarbazi-Azad H (2018) Cache Replacement Policy Based on Expected Hit Count. IEEE Comp Arch Letters 17(1): 64-67

[21] Faldu P, Grot B (2017) Leeway: Addressing variability in dead-block prediction for last-level caches. In: Proc of the 26th Int Conf on Parallel Archit and Compilation Tech (PACT). IEEE, pp. 180–193

[22] Wong W, Baer JL (2000) Modified LRU Policies for Improving Second-Level Cache Behavior. In: Proc of the Sixth Int Symp on High-Perform Comput Archit (HPCA), 2000. IEEE, pp. 49-60.

[23] Qureshi MK, Lynch DN, Mutlu O, Patt YN (2006) A case for MLP-aware cache replacement. In: 33rd Int Symp on Comput Archit (ISCA), 2006. IEEE, pp. 167-178

[24] Beckmann N, Sanchez D (2017) Maximizing cache performance under uncertainty. In: Proc of the Sixth Int Symp on High-Perform Comput Archit (HPCA). IEEE, pp. 109–120.

[25] Warrier T, Anupama B, Mutyam M (2013). An application-aware cache replacement policy for last-level caches. In: Int Conf on Arch of Comp Sys. Springer, Berlin, Heidelberg, pp. 207-219.

[26] Khan, Manabi S, Tian Y, Jimenez DA (2010) Sampling dead block prediction for last-level caches. In: Proc of the 2010 43rd Annu IEEE/ACM Int Symp on Microarchit. IEEE, pp. 175-186

[27] Kharbutli M, Solihin Y (2008) Counter-based cache replacement and bypassing algorithms. IEEE Trans on Comput 57(4):433-447

[28] Chaudhuri M, Gaur J, Bashyam N, Subramoney S, Nuzman J (2012) Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In: Proc of the 21st Int Conf on Parallel Archit and Compilation Tech (PACT), 2012. IEEE, pp. 293–304.

[29] Albericio J, Ibañez P, Viñals V, Llaberia J (2013) The reuse cache: downsizing the shared last-level cache. In: 46th Annu Int Symp on Microarchit (MICRO). IEEE/ACM, pp. 310–321.

[30] Wu CJ, Jaleel A, Hasenplaugh W, Martonosi M, Steely Jr SC, Emer J (2011) SHiP: Signature-based hit predictor for high performance caching. In: Proc of the 44th Annu IEEE/ACM Int Symp on Microarchit, 2011. IEEE/ACM, pp. 430–441

[31] Diaz J, Monreal T, Ibañez P, Llaberia J, Viñals V (2019). ReD: A reuse detector for content selection in exclusive shared last-level caches. J of Parallel and Distributed Comput 125:106–120

[32] Jain A, Lin C. (2016) Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In: 43rd Annu Int Symp on Comput Archit (ISCA), Seoul, 2016. ACM/IEEE, pp. 78-89.

[33] Belady LA (1966) A study of replacement algorithms for a virtual-storage computer. IBM Syst J 5(2):78–101

[34] Belady LA, Palermo FP (1974) On-line measurement of paging behavior by the multivalued MIN algorithm. IBM J of Res and Dev 18:2–19

[35] McFarling S (1991) Program Analysis and Optimization for Machines with Instruction Cache. Dissertation, Stanford University. Tech Rep No. CSL-TR-91-493

[36] Michaud P (2016) Some mathematical facts about optimal cache replacement. ACM Trans on Archit and Code Optimization (TACO) 13(4):50

[37] Qureshi MK, Moinuddin K, Thompson D, Patt YN (2005) The V-Way cache: demand-based associativity via global replacement. In: 32nd Int Symp on Comput Archit (ISCA), 2005. IEEE, pp. 544-555

[38] Jain A, Lin C (2018) Rethinking Belady's Algorithm to Accommodate Prefetching. In: 45th Annu Int Symp on Comput Archit (ISCA), 2018. ACM/IEEE, pp. 110-123.

[39] Lin WF, Reinhardt S (2002) Predicting Last-Touch References under Optimal Replacement. Tech Rep CSE-TR-447-02, Univ of Michigan.

[40] Jeong J, Dubois M (2006) Cache replacement algorithms with nonuniform miss costs. IEEE Trans on Comput 55(4):353-365

[41] Rajan K, Govindarajan R (2007) Emulating optimal replacement with a shepherd cache. In: 40th Annu Int Symp on Microarchit (MICRO), 2007. IEEE, pp. 445-454

[42] Gaur J, Chaudhuri M, Subramoney S (2011) Bypass and insertion algorithms for exclusive last-level caches. ACM SIGARCH Comput Archit News. 39(3):81-92

[43] Liu W, Yeung D (2009) Using Aggressor Thread Information to Improve Shared Cache Management for CMPs. In: Proc. of the 18th Int Conf on Parallel Archit and Compilation Tech (PACT), 2009. IEEE, pp. 372-383

[44] Zahran M, Albayraktaroglu K, Franklin M (2007) Non-inclusion property in multi-level caches revisited. Int J of Comp and Their Appl 14(2):99-108

[45] Snavely A, Tullsen DM (2000) Symbiotic job scheduling for simultaneous multithreading processor. In: Proc of the Int Conf on Archit Support for Program Lang and Oper Syst (ASPLOS), 2000. ACM, pp. 234–244

[46] Eggers SJ, Emer JS, Levy HM, Lo JL, Stamm RL, Tullsen DM (1997) Simultaneous multithreading: a platform for next-generation processors. IEEE Micro 17(5):12-19

[47] Henning JL (2006) SPEC CPU2006 benchmark descriptions. SIGARCH Comput Archit News 35(4):1–17

[48] Perelman E, Hamerly G, Van Biesbrouck M, Sherwood T, Calder B (2003) Using SimPoint for accurate and efficient simulation. In: Int Conf on Meas and Model of Comput Syst (SIGMETRICS), 2003. ACM, pp. 318–319

[49] Hamerly G, Perelman E, Lau J, Calder B (2005) SimPoint 3.0: Faster and more flexible program phase analysis. J of Instr Level Parallelism 7(4):1–28

[50] McFarling S (1993) Combining branch predictors. Vol. 49, Tech Rep TN-36, Digital West Res Lab

[51] Kim S, Chandra D, Solihin Y (2004) Fair cache sharing and partitioning in a chip multiprocessor architecture. In: Proc of the 13th Int Conf on Parallel Archit and Compilation Tech (PACT), 2004. IEEE, pp.111-122

[52] Luo K, Gummaraju J, Franklin M (2001) Balancing throughput and fairness in SMT processors. In: Proc of the IEEE Int Symp on Perform Anal of Syst and Softw (ISPASS), 2001. IEEE, pp. 164–171

[53] Diaz J, Ibañez P, Monreal T, Viñals V and Llaberia J. (2017) ReD: A Policy Based on Reuse Detection for a Demanding Block Selection in Last-Level Caches. The 2nd Cache Replacement Championship, June 25, 2017, Toronto, Canada. https://crc2.ece.tamu.edu/?page_id=53. Accessed 3 March 2021

[54] Young V, Chou C, Jaleel A, Qureshi M (2017) SHiP++: Enhancing Signature-Based Hit Predictor for Improved Cache Performance. The 2nd Cache Replacement Championship, June 25, 2017. Toronto, Canada. https://crc2.ece.tamu.edu/?page_id=53. Accessed 3 March 2021

[55] Jain A, Lin C (2017) Hawkeye Cache Replacement: Leveraging Belady's Algorithm for Improved Cache Replacement.The 2nd Cache Replacement Championship, June 25, 2017. Toronto, Canada. https://crc2.ece.tamu.edu/?page_id=53. Accessed 3 March 2021

[56] Diaz J. NOPT repository. https://github.com/jdmaag73/NOPT. Accessed 5 March 2021

[57] Various authors. ChampSim code repository. https://github.com/ChampSim/ChampSim. Accessed 3 March 2021

[58] Various authors. CRC2 trace repository. https://crc2.ece.tamu.edu/?page_id=41. Accessed 3 March 2021

[59] Wulf WA, McKee SA (1995) Hitting the Memory Wall: Implications of the Obvious. ACM SIGARCH Comput Archit News 23(1):20–24.