Where Replacement Algorithms Fail: a Thorough Analysis

Georgios Keramidas
Industrial Systems Institute
Patras, Greece
keramidas@isi.gr

Pavlos Petoumenos
Department of Electrical and Computrer Engineering
University of Patras, Greece
ppetoumenos@ece.upatras.gr

Stefanos Kaxiras
Department of Electrical and Computrer Engineering
University of Patras, Greece
kaxiras@ece.upatras.gr

ABSTRACT

Cache placement and eviction, especially at the last level of the memory hierarchy, have received a flurry of research activity recently. The common perception that LRU is a well-performing algorithm has recently been discredited: many researchers have turned their attention to more sophisticated algorithms that are able to substantially improve cache performance. In this paper, we thoroughly examine four recently proposed replacement policies: the Dynamic Insertion Policy (DIP), the Shepherd Cache (SC), the MLP-aware replacement, and the Instruction-based Reuse Distance Prediction (IbRDP) replacement policy. Our experimental studies show that there is a great *inconsistency* between the number of misses saved by each mechanism and the resulting improvement in IPC. This is particularly true for the DIP and the SC approach and indeed attest to the fact that these algorithms do not take into account the relative cost of each miss (i.e., whether it is an isolated or parallel miss). Their aim is to blindly lower the total number of misses. On the other hand, the MLP-aware replacement, although miss-cost-aware, cannot handle efficiently workloads which display LRU-hostile behavior and thus fails to reduce execution time even when there are ample opportunities to reduce cache misses. The IbRDP replacement policy shows both the ability to deal with non-LRU access patterns and MLP *friendliness* leading to greater consistency between the reduction of misses and the corresponding increase in performance thus the largest IPC improvement among the studied mechanisms. So, what are the appropriate characteristics of a replacement algorithm targeting the lower levels of the memory hierarchy? In this paper we are shedding some light on this question.

Categories and Subject Descriptors

B.3.2 **[Memory Structures]**: Design Styles— *Cache memories*

General Terms

Algorithms, Management, Performance, Design.

Keywords

Replacement/placement Policies/Algorithms, Last-Level Caches, Memory System, Profiling.

1. INTRODUCTION

The rapidly increasing imbalance between the relative speeds of processors and main memory is the predominant problem that computer architects are trying to mitigate. Sophisticated techniques like out-of-order execution, non-blocking caches, run-ahead execution, and kiloprocessors have been extensively explored, in order to hide to the extent possible the long latencies of the slow DRAMs by increasing the off-chip Memory-Level Parallelism (MLP) [4]. Compared to such schemes, classical techniques like cache replacement/placement have received little attention until recently, perhaps due to the implicit assumption that LRU is a well performing algorithm and there is not much margin of improvement for the replacement decisions. This may be the case for the L1 caches, but not for the lower members of the memory hierarchy (L2/L3 caches). LRU replacement algorithm tries to accommodate temporal locality by keeping recently used lines away from replacement, in hope that when they are reused, they will still be in the cache. Two reasons work against LRU in L2 caches.

L2 caches are usually highly associative, since they are not latency sensitive. The high associativity dictates that when a new item is placed into the cache, it has to travel all the way down the LRU stack until it becomes the LRU candidate for replacement. This means that lines with very large reuse distances (which are probably misses) will still occupy useful space in the cache without contributing to the hit rate. Ideally, these lines should be quickly replaced by lines with short temporal reuse patterns, even if such a decision requires a circumvention of the time ordering introduced by the LRU algorithm. The second reason which reveals that LRU is not ideal for L2 caches is the filtering effect of L1 caches. L2 caches are hidden behind the primary level caches and they are accessed only upon an L1 miss. This reality often inverts the temporal reuse patterns of the addresses as they appear in the L2. Consider for example a cache line which exhibits a burst of temporal reuses due to spatial locality, especially with current cache block sizes of 64 or 128 bytes. However, such a bursty pattern typically manifests at the L1 cache only. To the L2 cache, the cache line does not appear to have a temporal reuse. In other words, L1 filters the apparent behavior of the cache line.

The reasons mentioned above pertain to the temporal locality or the "recency" of the cache items and they actually represent just one part of the story that a replacement algorithm should take into account. The rest of the story has to do with the relative cost of the misses. Misses may occur either in isolation or in parallel with other misses. Isolated misses hurt performance the most because the processor is stalled to service just a single miss. In the case of parallel misses waiting to be serviced, the processor's idle time is divided among all those concurrent misses. The notion of servicing multiple outstanding cache misses in parallel is called Memory

Level Parallelism [2,4,9] and should not be confused with the notion of criticality [25] which actually is defined by how much the instruction stream processing continues in the face of a read miss.

In response to all those reasons, many recent approaches prove that more sophisticated algorithms, compared to the monolithic LRU, are able to substantially improve system performance. The Dynamic Insertion Policy (DIP) [22], the Shepherd Cache (SC) [24], the MLP-aware replacement [23], and the Instruction based Reuse Distance Prediction (IbRDP) replacement policy [19] are four recent approaches with very different characteristics and different mentality, but with the same target: increase the real estate of the L2 cache by optimizing its placement/replacement functionality.

Despite the generally good results of all these approaches, an obfuscation in the field still exists. First of all, there is a lack of a tentative comparison of those algorithms using exactly the same evaluation framework. Trace vs. cycle accurate simulations, different processor and/or memory configurations, and different benchmark's configurations (number of skipping instructions and/or different compilation parameters) does not allow someone to extract safe conclusions about the actual strength of the aforementioned policies. To deal with this issue, in this paper we offer a thorough evaluation of the four algorithms under the same framework (both system configuration and benchmark selection).

This experimental analysis enables us to deepen our understanding of those algorithms as well as to discover their pitfalls and their advantages. For example, both SC and DIP are carefully designed to predict/adapt to the temporal characteristic of the cache items (or the data reuse): SC attempts to emulate/mimic the Belady's optimal algorithm, while DIP works in a statistical manner and tries to prevent thrashing for benchmarks whose working set is greater than the cache size. But we observe that while the SC and the DIP approach successfully manage to lower the number of misses for many of the examined benchmarks, there is also a *significant inconsistency between the number of saved misses and the resulting improvement in the benchmark's speedup.* But, what is the source of this inconsistency? Studying these algorithms in more detail, we identify the culprit: Memory Level Parallelism (MLP) [9,23]. Indeed, SC and DIP are only optimized to blindly lower the number of misses without taking into account the cost of each miss (whether the miss is isolated or parallel). Therefore, they can only tackle one side of the problem.

In response to this Qurechi et al. exploit the MLP property to enhance the performance improvements resulting from the replacement decisions [23]. The target of the MLP-aware replacement policy is simple: data that do not exhibit MLP are given preference for staying into the cache over data that do exhibit MLP. This is because accessing the former type of data is much more costly than accessing the latter. However, our experience with this replacement algorithm reveals that its benefits are pronounced only for benchmarks with regular temporal access patterns. In workloads with non-regular access patterns, MLP-aware replacement has a neutral or even a pathological behavior even if there are ample opportunities to reduce the number of misses. This inefficiency comes from the fact that the MLP-aware replacement is not equipped with an explicit mechanism able to orchestrate the replacement decisions according to the temporal locality of the blocks that are inserted into the cache.

Having analyzed the pitfalls as well as the advantages of the three aforementioned mechanisms, we are now able to formulate two distinct characteristics that an ideal[1] replacement/placement algorithms should have. Those are:

- Ability to adapt its replacement/placement decisions based on the temporal behavior of the memory references.
- Ability to assign finite costs to the replacement candidates based on their MLP properties and adjust the replacement/placement strategy accordingly.

Finally, in this work we study another replacement algorithm, called Instruction based Reuse Distance Prediction or IbRDP [12,19]. According to our experimental results IbRDP achieves the largest IPC improvement among the studied algorithms. IbRDP provides an explicit mechanism to directly predict the temporal locality (or the reuse distances) of the memory accesses. This mechanism is triggered by the access pattern of the instructions (PCs) which access the cache lines and it outputs with high accuracy when a specific cache line is going to be reused in the future. Further analyzing this algorithm, we discover that it shows both ability to deal with non-LRU access patterns and a great consistency between the reduction of misses and the corresponding increase in performance (*MLP-friendliness*).

The main contributions of this work are the following:

- We thoroughly evaluate four recently proposed replacement/placement policies (Shepherd Cache [24], Dynamic Insertion Policy [22], MLP-aware replacement [23], and the Instruction based Reuse Distance Prediction [19] for all SPEC2000 benchmarks and for various L2 sizes. To the best of our knowledge this is the first work that provides a quantitative comparison of those recently proposed replacement/placement algorithms.
- We provide a critical view on those replacement algorithms and we further analyze their advantages and disadvantages.
- Based on the above observations, we formulate the exact characteristics that an ideal replacement algorithm should take into account. Those are the temporal patterns of the memory references and the MLP-cost of the misses saved by the examined algorithm.
- We provide analytical profile results for all the interesting benchmarks of the SPEC2000 suite based on the above criteria. More specifically, we compute the distribution of the MLP-cost for all the benchmarks, while we use the notion of the stack distance to quantify the temporal characteristics of the benchmarks.

The remainder of this paper is organized as follows: In Section 2, we briefly discuss related work and we delve more into details about the four replacement algorithms that we examine in our paper. Section 3 describes the simulation methodology and Section 4 presents a thorough profiling analysis of the temporal locality and the distribution of the MLP-cost for each benchmark. Section 5 presents our experimental results in terms of IPC

[1] Do not confuse the term of ideal replacement with the OPT Belady's algorithm [1] which dictates as the optimal candidate for replacement, the one that is going to be accessed farthest in the future. The OPT algorithm is only optimized for predicting the temporal behavior of the memory blocks ignoring their MLP-cost, and as it is proven in [23], it does not the provide the best results in terms of IPC.

Table 1. Parameters for the four Policies

Policy	Parameters
LRU	16-way L2 caches
DIP	64 leader sets (32+32), epsilon=1/32
Shepherd Cache	SC-4, MC-12
MLP-aware	SBAR implementation (1/32 leader sets), lamda=4
IbRDP (with and without Selective Caching)	Predictor: 256 entries organized in 8 ways, 2-bits confidence counters.

improvement and read misses reduction. Section 6 discusses some special cases which are able to reveal the pros and cons of each algorithm. Finally, Section 7 offers our conclusions.

2. BACKGROUND AND RELATED WORK

In this work, we provide a thorough comparison of four recently proposed algorithms (SC, DIP, MLP-aware, and IbRDP) and we claim that the success of a replacement/placement mechanism depends on its ability to both shape its behavior according to the workload's temporal characteristics and victimize the cache items associated with low MLP-cost. We will briefly overview related work in each of these directions.

MLP aware schemes. One of the first works in the area was by Chou et al. [2]. This paper formally defines MLP as "the number of useful long latency off-chip accesses outstanding when there is at least one such access outstanding." The authors also analyze the effectiveness of various techniques like out-of-order execution, runahead execution and value prediction on increasing MLP. Their results indicate that those techniques have a substantial impact on improving MLP. The real impact of MLP was exposed by Karkhanis and Smith [9] in their attempt to provide a first order model of a superscalar core. Their work provides a fundamental insight into how important is the parallelism of L2 misses in shaping the performance of out-of-order execution. Eyerman and Eckhout [3] exploit MLP to optimize the fetch policies for Simultaneous Multi-Threading (SMT) processors and they show that there is a strong correlation between MLP and the instructions (PC) that miss in the L2 cache. Petoumenos et al. [20] expoit this correlation in order to drive a processor window resizing mechanism, which lowers the power consumption of the processor core with minimal impact in performance, by protecting MLP.

Cache Management. Early work studied the limits of cache replacement algorithms using program traces. The first attempt was by Belady [1] who formulated the area by comparing random replacement algorithms, LRU and an optimal algorithm that looks into the future. Since then, many researchers proposed schemes aiming to reduce the cache miss rate by providing better replacement decisions.

In the area of L1 cache management, many techniques categorize, and handle accordingly, the memory references based on their temporal and spatial characteristics [6,8,10,18]. Jeong and Dubois introduce the idea of cost sensitive replacement algorithms [6]. The authors proposed different variations of LRU by assigning finite costs between load and store misses. Another class of replacement techniques [13,26] typically involves the exploitation of the generational behavior of a cache block from the moment that a specific line is inserted into the cache, until the time that is evicted —inspired by the cache decay methodology [11].

Most of the work in the context of the L2 caches focuses either in the identification of dead lines (e.g. last touch prediction), or in providing cache bypassing schemes [7,14,16,21,27,28]. However, since in this paper our main target is to systematically compare four recently proposed replacement/placement algorithms, we will continue by offering a short description for each of them. The configuration parameters of the four algorithms are shown in Table 1.

Dynamic Insertion Policy (DIP) [22]. This approach presented by Qurechi et al. tries to prevent cache blocks that are not expected to exhibit temporal locality from staying in the cache for long. This is done by dynamically adapting the position of the LRU stack at which a cache block is inserted. In order to be able to adapt to different applications/program phases, the DIP approach utilizes a technique called Set Dueling. According to this technique, a small portion of the cache sets, called leader sets, implements either the traditional LRU or the Bimodal Insertion Policy (BIP). BIP probabilistically inserts the incoming cache lines either in the MRU or in the LRU stack position. The policy (LRU or BIP) that has fewer misses in the leader sets is the one that determines which replacement should be applied in the remaining (follower) sets of the cache.

Shepherd Cache (SC) [24]. The SC is an interesting approach that attempts to emulate the Belady's optimal algorithm [1] for a conventional cache. This technique proposes to replace the large, highly associative cache with a cache with lower associativity, Main Cache (MC) and use a secondary cache, Shepherd Cache (SC), to approximate the Belady's optimal algorithm in the MC. The SC approach works in the following way: upon a cache miss, the requested line is initially buffered in the SC, and while it resides there, it gathers information about potential replacement candidates. So when the line is transferred in the MC, it replaces the line which is expected to be used farthest in the future. The authors prove that this gathered information facilitates in making a replacement decision that approximates the Belady's OPT fairly well [1].

MLP-aware Replacement Policy [23]. Minimizing the absolute number of misses, as it is proposed by Belady [1], does not necessarily have a proportionally positive impact on performance. This is because not all the misses are equally costly. Some misses occur in isolation, whereas some misses occur in parallel with other misses. Isolated misses hurt performance the most because the processor is stalled to service just a single miss. In case of parallel misses that are waiting to be serviced, the processor idle time is divided among all those concurrent misses.

In the MLP-aware replacement policy, the authors try to exploit the MLP properties of the cache lines in order to enhance the performance improvements resulting from the replacement decisions. According to this scheme, every time a block is inserted into the cache, the MLP-cost of this cache block is computed. The authors show that for most of the benchmarks the currently computed MLP-cost of a block can be used as a predictor of the next MLP-cost of the same block. This cost metric (associated with an coefficient factor called lamda) is used as an input to a cost function which eventually drives the replacement policy of the cache (called LIN policy). However, the authors show that a hybrid replacement mechanism, able to revert back to LRU, is greatly needed in order to deal with cases where LIN hurts performance. Therefore, they use a sampling technique (called SBAR similar to Set Dueling approach proposed in [22]), to choose the best performing policy (between LIN and LRU). The number of the required leader sets, as well as the lamda value used in rest of this paper, is given in Table 1.

Instruction based Reuse Distance Prediction Replacement Policy (IbRDP) [19]. In this work, the authors argue that it is possible to fully quantify the temporal characteristics of the cache blocks at run-time by predicting the cache block reuse distances (measured in intervening cache accesses), based on the access pattern of the instructions (PCs) that touch the cache block. They carefully design the organization as well as the functionality of the instruction based reuse distance predictors (similar to branch predictors) in order to achieve a good trade-off between the implementation cost and the performance improvement. They show that dedicating a small amount of hardware to these predictors affords high accuracy in predicting when a memory reference is going to be accessed in the future. Having reuse-distance information for each cache block in a set, allows to approximate optimal replacement decision by looking into the future (it is a way to "see" the future). However, lack of prediction information (not all the cache accesses are predictable) requires to look into the past too. As a result, a hybrid replacement algorithm is proposed which automatically reverts to LRU when it is required. Finally, the authors propose an extension of the IbRDP algorithm in which it is possible to victimize the currently fetched block by not caching it at all in the L2 (Selective Caching). This happens when the currently fetched cache block has a prediction for a reuse distance that exceeds the expected lifetime of the blocks residing already in the set. We refer to the latter algorithm as IbRDP+SC (Selective Caching).

3. EXPERIMENTAL METHODOLOGY

Our experiments were performed using a detailed cycle accurate simulator that supports a dynamic superscalar processor model. Our baseline processor is a 4-way out-of-order processor with an 80-entry instruction window. We simulate a 32K, 64 byte block, 4 way, dual-ported, 2 cycle L1 data cache and a 16-way, 13 cycle unified L2 cache of various sizes. The main memory has a 500 cycles latency and is able to deliver 16 bytes every 8 cycles. We use various L2 cache sizes in order to have a good understanding about the L2 cache behavior.

We run all the applications of the SPEC2K benchmarks, but in the interest of clarity we only show results for 15 benchmarks which have more than 1 miss per kilo-instructions (MPKI). Benchmarks with less than 1 MPKI have low cache requirements and do not benefit much from advanced replacement algorithms. The simulations were performed after skipping the initialization phase for all benchmarks. We simulate 200M instructions after skipping 3 billion instructions for *ammp*, 2B for *mcf*, *twolf* and *vpr*, and 1B for the rest of the benchmarks. Finally, in order to take a more representative picture of the actual strength of the replacement algorithms, we warm up the caches for 100M instructions before we start collecting statistics.

4. ANALYZING THE TEMPORAL AND THE MLP CHARACTERISTICS

The premise of this work is to identify the appropriate characteristics of a replacement algorithm targeting the lower levels of the memory hierarchy (L2/L3 caches). Our experience with the four recently proposed replacement/replacement policies (DIP, SC, MLP-aware, and IbRDP) reveals that the success of a policy depends both on its ability to shape its behavior according to the workload temporal characteristics as well as on its ability to victimize cache items with high MLP-cost. Hence, in order to be able to analyze how the four replacement algorithms perform for a specific benchmark (Section 5), we first need to uncover the temporal and MLP characteristics of the benchmarks.

Figure 1 presents the profiles we produced for the baseline LRU case and for the 15 benchmarks that we consider in this work. Due to lack of space, for every benchmark we present only two graphs. The first graph (shown at left) depicts the distribution of the MLP-related cost measured in a 512K (left part of the graph) and 1M cache (right part of the graph). The MLP-costs are measured according to the methodology presented in [23]. The y-axis of this graph corresponds to the measured number of misses (absolute value), while the x-axis corresponds to the quantized MLP-cost, which we produce in our work by dividing absolute costs by 64. Thus, the leftmost bar (in each part of the graph) represents the number of misses that had a MLP-cost between 0 and 64 (fully parallel misses), while the rightmost bar depicts the number of the isolated misses.

The second graph for each benchmark aims at capturing the temporal behavior of the benchmarks and represents the stack distance profiles [15,17] measured in a 512K cache (stack distances profiles for the other cache sizes are just an upscaled/downscaled version of the presented graph). Stack distance analysis is a powerful basis to understand the temporal characteristics of the applications [15,17]. Each stack distance profile collects the histogram of accesses to different LRU stack positions in the cache. For example, if there is an access to a line in the nth MRU position, the nth counter (nth number shown in the x-axis of the graph) is incremented. The stack distance profiles presented in Figure 1 illustrate both the distances of the hits (light blue bars) and misses (dark red bars). We should note that in all graphs presented in Figure 1, we exclude the compulsory misses which can not be avoided by improving the replacement decisions.

As we see in Figure 1, different benchmarks exhibit different characteristics. This is particularly true for the MLP-costs in contrast to the stack distance profiles which rather show a more straightforward distribution over the benchmarks. Before analyzing the graphs in more details, we should mention that rarely *two benchmarks display the same characteristics in both profiles rendering the job of a replacement algorithm an uphill battle*.

Considering the MLP-cost distributions, our results show that there is clear non-uniformity in MLP-cost which signifies that this characteristic can be exploited by a replacement algorithm (replicating the work in [23]). In more detail, for 9 out of 15 benchmarks, the number of isolated misses is far less than 15% of

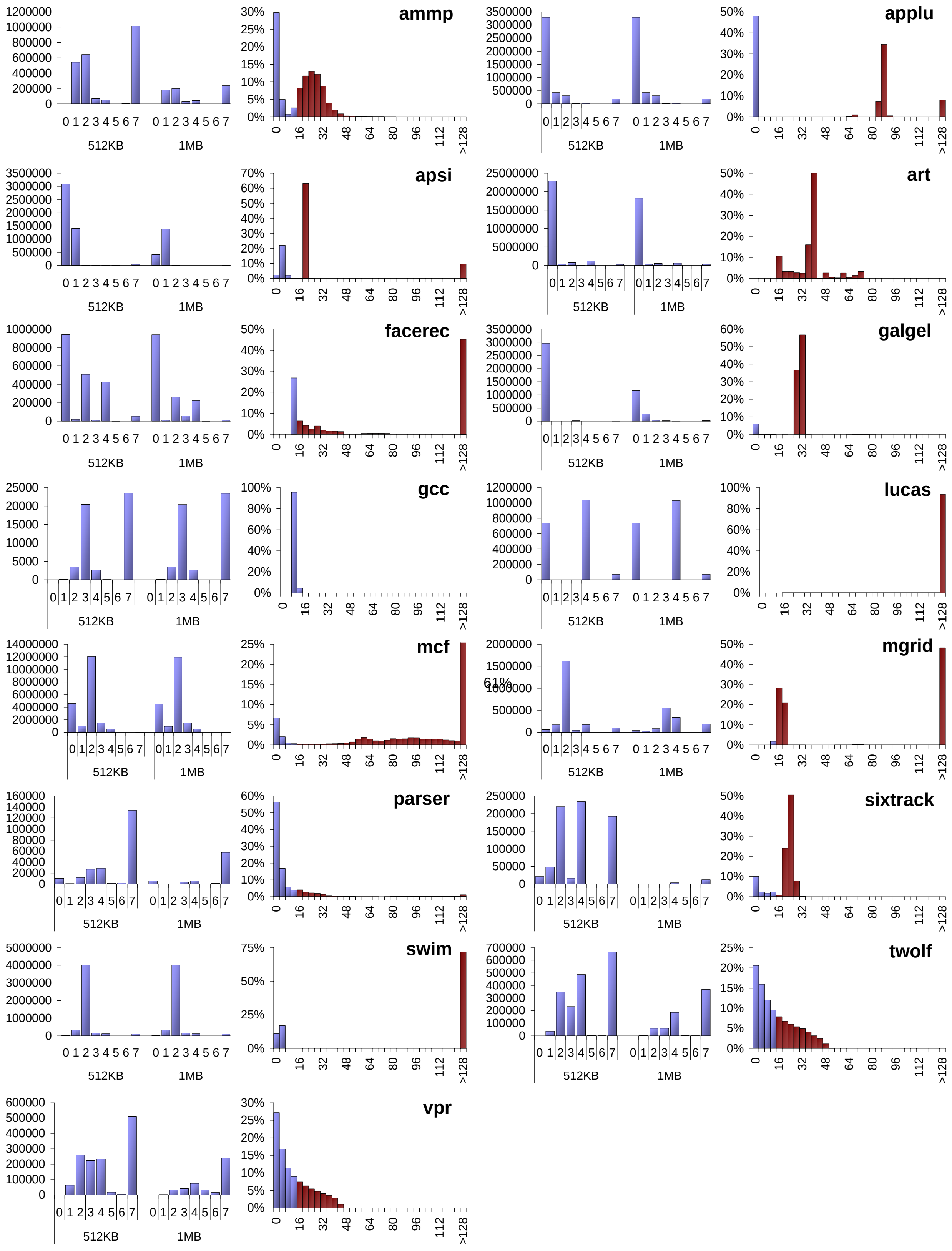

Figure 1. MLP-cost and stack distances profiles for 15 SPEC2K benchmarks with more than one MPKI.

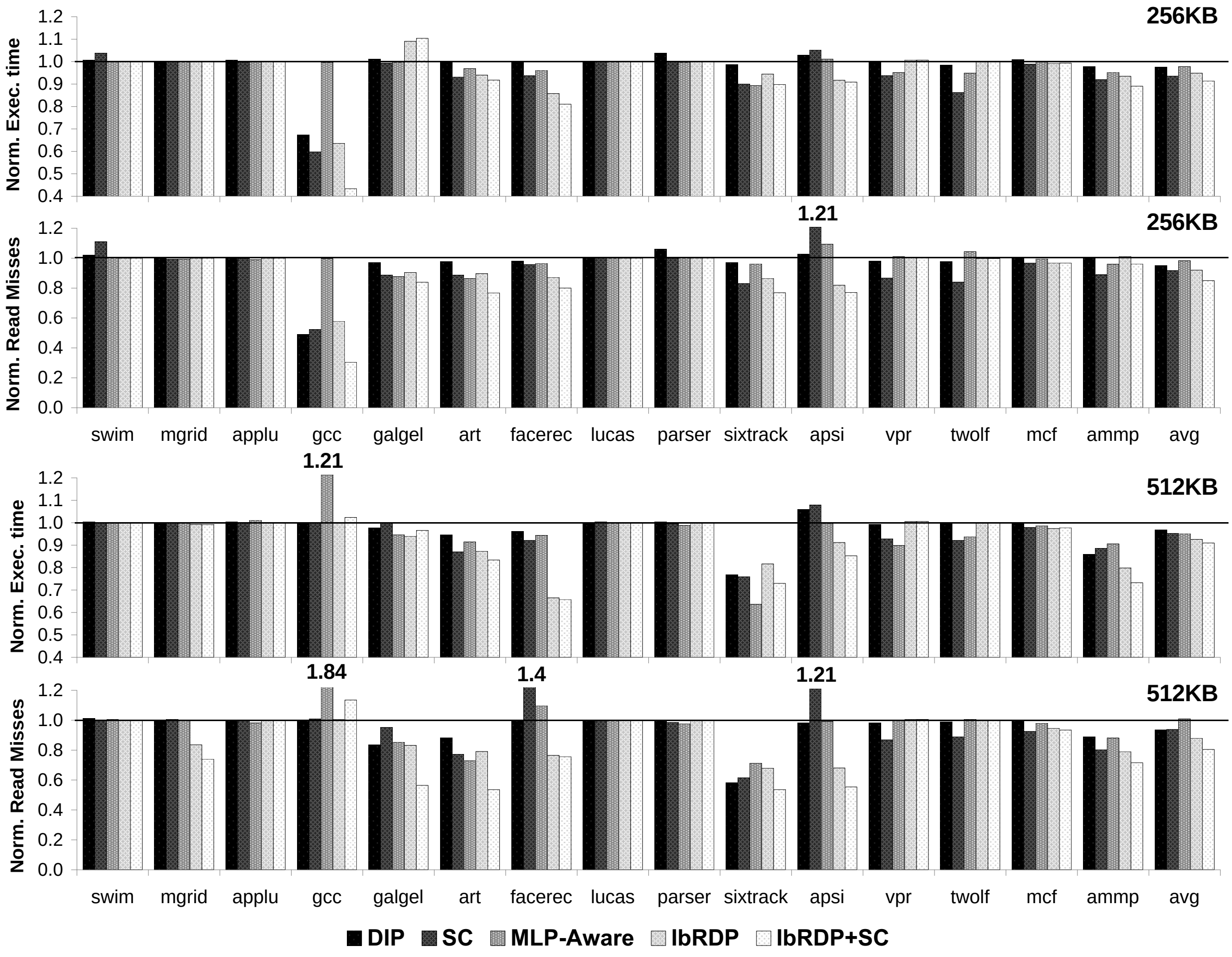

Figure 2. Normalized execution time and normalized reduction of misses results for the DIP, SC, MLP-aware, IbRDP, and IbRDP+SC policies and for 256K (top) and 512K (bottom) L2 cache sizes.

the overall misses indicating a high memory level parallelism. A clear exception is *parser* with 61% (512K) and 75% (1M) of its misses being isolated misses. Other benchmarks with a noticeable number of isolated misses are *sixtrack* (26% and 65% in the 512K and 1M cache respectively), *gcc* (46% and 46%), *vpr* (38% and 55%), *twolf* (37% and 54%), and *ammp* (43% and 34%). Incidentally, in these benchmarks (except *gcc*), there is a great difference in the percentage of the isolated misses in the 512K and the 1M cache. On the other end, benchmarks like *applu*, *art*, *galgel*, *mgrid*, *swim* and *aspi* have a high percentage of parallel misses. For these cases, a replacement algorithm targeting to lower the MLP-cost of the misses is likely to fail, since there are not many opportunities for improvement.

For the stack distance profiles, the distributions illustrated are much more clear. There are two distinct categories in this case. Benchmarks for which the stack distances of misses are distributed right after the cache limit (stack distance 16), and benchmarks for which the total number of misses is stacked at very large distances (> 128). While there is little one can do to deal with the misses belonging to the latter category, there is ample room for improvement in the first category. Benchmarks like *ammp*, *galgel*, and *apsi* can easily benefit by a better replacement algorithm since a great number of misses will turn into hits. However, as it is further explained in Section 6, for algorithms that work in a statistical fashion, like DIP, the important issue is how uniformly the number of misses is spread out across the stack distances.

Having described the MLP and locality characteristics of the applications, let us now present our experimental results of the quantitative comparison of the four replacement/placement mechanisms considered in this paper.

5. COMPARING THE FOUR POLICIES

To the best of our knowledge this is the first work that provides a tentative comparison of the four recently proposed best performing replacement/placement mechanisms. One common obstacle in implementing previously proposed mechanisms is how one can validate the implementation. This was not a problem for the DIP, since we just ported the code distributed by the authors [22] in our simulation framework. For the rest of the mechanisms, we carefully implemented the required structures by reverse engineering the corresponding articles. In all cases, our results match the results provided initially by the authors by a factor of 70-80%. We consider this as safe limit given that i) different numbers of skipping instructions were used for many benchmarks and ii) we chose a processor/memory configuration which will not downscale —to the extent possible— the strength of any of the four mechanisms. Details about the system configuration are given

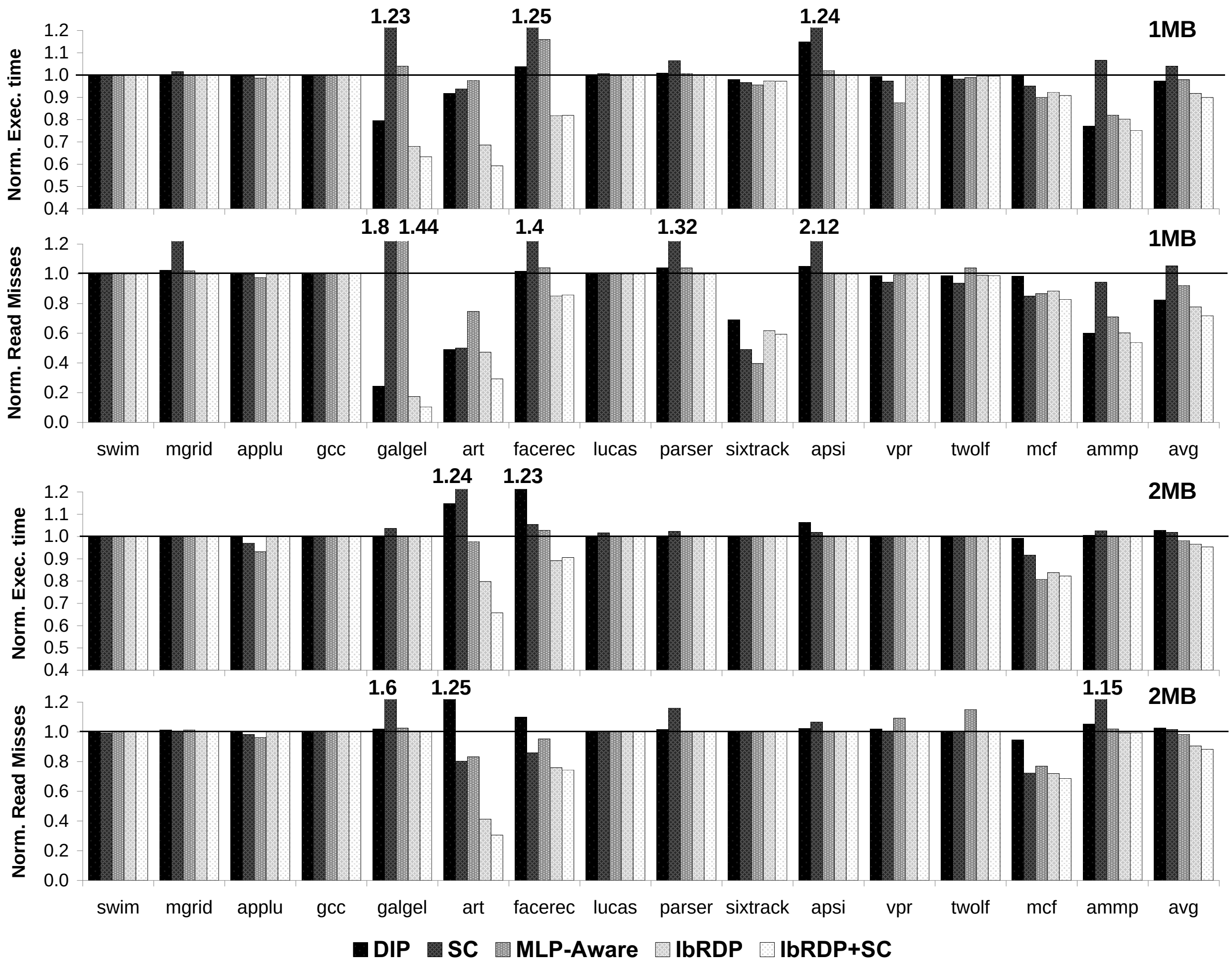

Figure 3. Normalized execution time and normalized reduction of misses results for the DIP, SC, MLP-aware, IbRDP, and IbRDP+SC policies and for 1M (top) and 2M (bottom) L2 cache sizes.

in Section 3, whereas the parameters used in the implementation of the four algorithms can be found in Table 1.

In addition to the four aforementioned policies (DIP, SC, MLP-aware, and IbRDP), we chose to implement a variation of IbRDP, called IbRDP+SC (Selective Caching). This was done because IbRDP is the only algorithm (among the four algorithms examined in this work) which easily supports a selective caching mechanism.

The results of the comparison are presented in Figure 2 (256K and 512K cache) and Figure 3 (1M and 2M cache). Studying the impact of the five policies in either smaller or larger caches will not be useful, since the working sets of the benchmarks either do not "fit" at all in the cache or "fit" in their entirety. For every cache size, we present two graphs. The first graph illustrates the performance improvements —reduction in execution time normalized to the LRU case— and the second graph depicts the reduction in the performance-critical read misses over the LRU. Finally, the first, second and third bar in each set of bars represent the examined metric for DIP, SC, and the MLP-aware replacement algorithm respectively, while the fourth and the fifth bar illustrate the results for the IbRDP and the IbRDP+SC. As Figure 2 and Figure 3 indicate, IbRDP+SC is clearly the best performing mechanism[1] over all the cache sizes. IbRDP+SC outperforms DIP on average by 6.4% (6.1%, 7.5%, 7.2%), SC by 2.4% (4.5%, 13.5%, 6.3%), MLP-aware by 6.6% (4.3%, 8.2%, 2.9%), and IbRDP by 3.7% (1.8%, 2%, 1.3%) in the 256K cache (512K, 1M, and 2M respectively). Trying to rank the algorithms using the average values over all the cache sizes, the order goes as follows (from the best to the worse): IbRDP+SC, IbRDP, MLP-aware, SC, and DIP. However, IbRDP+SC is not always superior. For five cases, the competitors perform better than IbRDP+SC (*sixtract*-512K: SC and MLP-aware, *vpr*-1M: SC and MLP-aware, *mcf*-2M: MLP-aware). One other issue is that IbRDP+SC always performs better than its predecessor (IbRDP) indicating that the ability to victimize the currently fetched block by not caching it at all (Selective Caching) is applied with great success on top of IbRDP. We should note here that no one of the other mechanisms has the potential to support such a "victimization" technique.

In general (further details will be given in Section 6), there are two more issues (except the performance benefits) that make the IbRDP+SC (and IbRDP) to look more concrete compared to other algorithms. The first one is that it degrades performance only in

[1] In this work we do not take into account the hardware cost associated with each algorithm, but we examine only the performance benefits provided by each of them. However, we should mention that in terms of hardware cost as well as simplicity the clear winner is DIP [22].

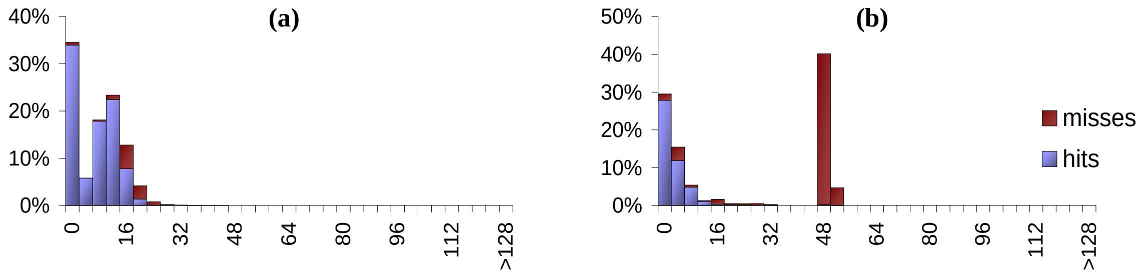

Figure 4. Stack Distances profiles using DIP for (a) *ammp* (1MB) and (b) *facerec* (2MB).

one case (*galgel*-256K). In contrast, a large number of noticeable increases in execution times are observed for the other algorithms. DIP degrades performance in *aspi*-512K (6%), *aspi*-1M (15%), *art*-2M (15%), *facerec*-2M (23%), and *aspi*-2M (6.4%). SC reports performance degradations in *aspi*-512K (7.8%), *galgel*-1M (23%), *facerec*-1M (25%), *parser*-1M (6.5%), *apsi*-1M (24%), *ammp*-1M (6.6%), *art*-2M (24%), and *facerec*-2M (5.5%), while MLP-aware replacement reduces the performance of *gcc*-512K (21%) and *facerec*-1M (23%).

The second and more important issue that advocates the use of the IbRDP mechanism is that it yields a great consistency between the reduction of the misses (compared to the LRU) and the corresponding performance gains. Indeed, for almost all cases the number of the misses saved by the IbRDP algorithm translates into a proportional increase in performance. This is an inherent property of the IbRDP approach and it does not appear for DIP and SC (the other two algorithms that try to shape their decisions according to the temporal characteristics of the workloads). As it will be further explained in Section 6, *this happens because IbRDP works in a MLP-friendly fashion, in contrast to DIP and SC which rather exhibit a MLP-hostile behavior.*

6. INTERESTING CASES: WHERE THE PERFORMANCE IS LOST?

In this Section, we provide an in depth analysis of the four polices we consider in this paper and we discuss some special cases in order to reveal some of their negative points.

Dynamic Insertion Policy. DIP while simple is far from being the best replacement mechanism. Its best cases are when the benchmark starts to fit in the cache. In this case it is both easy to eliminate a miss by keeping a line a little bit longer in the cache and the part of the working set that stays in the cache is likely to be useful in the future. The experiments where DIP achieves significant speedups, fit this profile, with *ammp* using an 1MB cache as a characteristic example. As we see in Figure 4.a, *ammp* almost fits in the 1M as most accesses have stack distances below 16 and the rest have stack distances between 16 and 23. Additionally, there are no longer stack distances, so if DIP chooses to keep a line in the cache (by inserting it in the MRU position), future accesses to this cache line have an extremely high probability to be hits.

A significant drawback of DIP, which is observed even in this case, is that because of its statistical nature it treats every newly allocated line as equal, without taking into account the difficulty of keeping the line in the cache until it is accessed again. Lines with short stack distances will be inserted in the LRU position for a number of accesses resulting in misses, while some lines with stack distances above 20 will be chosen for MRU insertion over lines which fit in the cache. While this behavior improves performance in comparison to LRU, it makes DIP behave worse than policies which “remember” the past like SC and IbRDP.

Figure 4.b illustrates a case where DIP's behavior results not only in sub-optimal management but also in significant performance loss. Half the accesses of *facerec* (2M) have stack distances below the cache associativity (cache hits), while most of the other accesses are difficult to change into hits, since they are characterized by medium length stack distances. A closer inspection of the data access patterns also reveals that stack distances do not correlate well with data: accesses to the same line display both short and long stack distances intermingled. The end result is to keep data in the cache which are not going to be used soon (just shortly reused on their last access), while at the same time inserting in the LRU position useful cache lines. Even though Bimodal Insertion does not perform well, one would expect that the hybrid nature of DIP and set duelling would at least bring the results at the same level with LRU, as it happens for other experiments. Unfortunately, the fact that the follower sets do not use Bimodal Insertion always, as the leader sets do, cause them to try to keep in the cache qualitatively different parts of the working set than the leader sets. So, while the follower sets display 10% more misses in comparison to LRU, the BIP leader sets decrease their misses by 15% causing DIP to choose Bimodal Insertion Policy as the preferable policy. This behavior is not specific to *facerec*, as it was also encountered in the experiments for *art* in the 2MB cache and is not caused by the leader sets selection algorithm, as nothing changed when the roles of the LRU leader sets and BIP leader sets were reversed. While, this behavior could be described as a “freak accident,” it does signify that the results that DIP produces are very sensitive on the configuration of the leader sets and the inherent randomness of DIP.

Shepherd Cache. The Shepherd Cache mechanism tries to get advantage of the repeating nature of the access patterns (i.e. loops). Hence, by gathering information about the relative order of the accessed blocks, it is possible to approximate the Belady's optimal algorithm [1]. The success of this approximation depends on the regularity of the access patterns and on the ability to accommodate the gathered information (size of the shepherd ways).

As we see in Figure 2, SC is the best performing policy for *twolf* (256K). In this case, IbRDP fails to make a difference because *twolf's* reuse distances do not show a good correlation with the instructions which generate the accesses. DIP shows marginal benefits due to the fact that the appeared access patterns are very random and the observed stack distances do not follow a regular ordering. Finally, MLP provide some benefits in terms of IPC

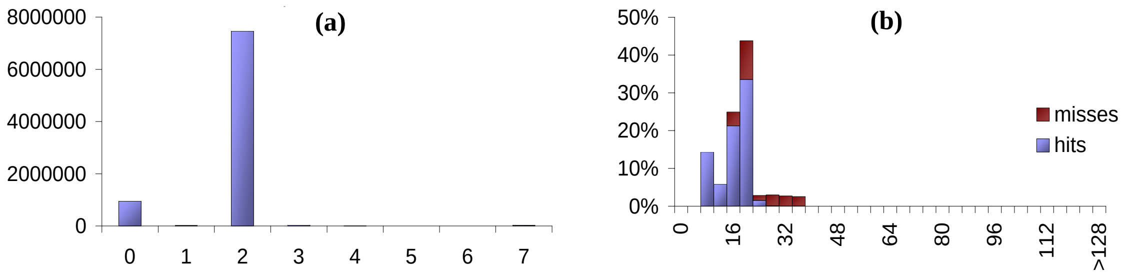

Figure 5. (a) MLP-cost distribution for *gcc* (256KB), (b) Stack distances for *art* (1MB) using IbRDP.

because it manages to replaces costly misses with "cheap" misses although the absolute number of misses is increased.

In general, SC manages to decrease misses because it can remember past behavior. A small subset of the frequently used blocks can stay in the cache almost permanently despite their occasional long stack distance. This happens because SC frequently succeeds in calculating their relative order and these blocks are placed near the top of the LRU stack. Less frequently accessed lines will either don't get accessed while in the SC and will be quickly evicted (LRU insertion in the Main Cache) or they will quickly lose their ordering information and start traveling down the LRU stack resembling a MRU insertion in DIP.

SC and DIP attempt to achieve the same target: keep a subset of the working set in the cache. The difference is that using SC the selected cache lines can stay more in the cache (SC associativity + 1 replacement) until it is decided whether they are worth keeping compared to 1 replacement that LRU insertion offers (DIP case). Therefore, SC is able to always produce hits for very small stack distances, which makes it less random in choosing what eventually will stay in the cache. In other words, it tends to keep more aggressively useful data than DIP does.

On the other hand, SC has a major weak point: if frequently used data are reused with stack distances greater than SC associativity + 1, then SC will always treat them as not useful and will evict them before they are reused. A clear example is *facerec* (1M). For some parts of *facerec's* execution, the access pattern seen in each set consists of exactly 8 blocks read sequentially for many times. Since none of these block were previously present in the cache, each one in turn is brought into the Shepherd Cache. After 4 misses, each of these blocks is transferred in the LRU position of the Main Cache and after the 5th miss the block is evicted. This makes the cache to appear almost 3 times smaller and less associative, resulting in an increase of the execution time by 25% and an increase in the number of misses by 40% compared to LRU. Similar behavior is witnessed in all experiments where SC performs much worse than LRU: *apsi* (512KB and 1MB), *galgel* (1MB), and *art* (2MB).

MLP-aware Replacement Policy. MLP-aware replacement uses a version of LRU, skewed in a way that victimizes the blocks which are not expected to cause isolated misses. Its success depends on the distribution of MLP-cost as well as the MLP-cost predictability.

A case where this policy outperforms the other schemes is for *sixtrack* (512K). As we see in Figure 1, most accesses of *sixtrack* almost fit in the cache, so there are ample opportunities for management (easily attainable by all mechanisms). But MLP-aware goes one step further as it reduces execution time 10% more than the second best policy. The reason for that is apparent in the MLP-cost distribution graph: almost one third of *sixtrack's* misses are isolated, so there is a great potential for a scheme which prefers to eliminate costly misses. Additionally, MLP-cost is easily predictable: 95% of the misses incur the same MLP-cost as the previous miss which brought the same line into the cache.

Exactly the opposite happens for *gcc* (256K). In Figure 5.a, we see the distribution of MLP-cost of *gcc* for this cache. Almost all misses of *gcc* are parallel and have the same MLP-cost, so for most replacement decisions MLP-aware replacement will behave exactly like LRU. But even for the relatively few isolated misses (less than 0.3% of the misses) MLP-aware replacement does not help much: most of them are cold misses and as for the other misses MLP-aware replacement manages to predict right and eliminate only 40% of them. All these result in less than 1% decrease of the execution time, whereas the second worst policy achieves a 33% decrease.

Instruction-based Reuse Distance Prediction. Instruction-based Reuse Distance Prediction takes replacement decisions based on the assumption that the blocks accessed by the same instruction will be accessed again after an almost equal number of accesses. When this is the case, IbRDP is practically identical to Belady's algorithm [1].

One such case is *art*, where our experiments show that the correlation between instructions and reuse distances is correct for 85% of the accesses. Figure 5.b illustrates the stack distance distribution of *art* (1M). IbRDP introduces almost no extra miss below the associativity limit, while managing to turn most accesses with stack distances up to 21 into hits. That translates into 71% less misses and 41% decrease of the execution time. When the basic assumption of IbRDP is not correct, the replacement policy produces results which vary from slightly worse to slightly better than LRU, as already mentioned for *twolf* (256K). But curiously IbRDP really fails only for *galgel* (256K), where it correctly predicts the reuse distances for 78% of the accesses. Virtually all of *galgel's* misses at this cache size are completely parallel, so even though three of the four algorithms manage to reduce the misses by more than 10% (up to 16% for IbRDP), no significant benefit results from this reduction. At the same time, IbRDP introduces a relatively small number of extra misses due to mispredictions, but because many of these misses are isolated, they have a disproportionate impact on execution time (10% increase).

7. CONCLUSIONS

In this paper, we systematically examine five recently proposed replacement/placement policies. Our elaboration deals not only with which is the best performing algorithm, but it also helps us to deepen our understanding on how these algorithms work. We find out that while Shepherd Cache and Dynamic Insertion Policy manage to improve the performance of workloads with LRU hostile behavior, they fail to handle efficiently workloads whose misses exhibit a great variation in their MLP-properties. In contrast, MLP-aware replacement, although able to exploit MLP with great success, it cannot recognize opportunities to improve performance when they involve eliminating parallel misses. Finally, Instruction based Reuse Distance Prediction replacement policy (with and without Selective Caching) shows both ability to deal with non-LRU access patterns as well as to perform in an MLP-friendly fashion and as a result outperforms the studied policies. Overall, the results presented in this paper stress the fact that both paths to improving cache behavior are equally important as targets of a truly efficient replacement policy.

8. ACKNOWLEDGEMENTS

This work is supported by the EU-FP6 Integrated Project, Scalable computer ARChitecture (SARC), Contract No. 27648 and the EU-FP7 ICT Projects, "A highly efficient adaptive multi-processor framework (HEAP)," Contract No. 247615, and "Embedded Reconfigurable Architecture (ERA)," Contract No. 249059.

9. REFERENCES

[1] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 1966.

[2] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploring memory-level parallelism. *Proc. of the International Symposium on Computer Architecture*, 2004.

[3] S. Eyerman and L. Eckkhout. A MLP-Aware Fetch Policy for SMT Processors. *Proc. of the International Symposium on High Performance Computer Architecture*, 2007.

[4] A. Glew. MLP yes! ILP no! *In Wild and Crazy Ideas Session, 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[5] A. González, C. Aliagas, and M. Valero. A Data Cache with Multiple Caching Strategies tuned to Different Types of Locality. *Proc. of the International Conference on Supercomputing*, 1995.

[6] J. Jeong and M. Dubois. Cost-sensitive cache replacement algorithms. *Proc. of the International Symposium on High Performance Computer Architecture*, 2003.

[7] T. Johnson, D. Connors, M. Merten, and W. Hwu. Run-Time Cache Bypassing. *IEEE Transactions on Computers*, 1999.

[8] M. Kampe and F. Dahlgren. Exploration of the Spatial Locality on Emerging Applications and the Consequences for Cache Performance. *Proc. of the International Parallel and Distributed Computing Symposium*, 2000.

[9] T.S. Karkhanis and J.E. Smith. A first-order superscalar processor model. *Proc. of the International Symposium on Computer Architecture*, 2004.

[10] M. Karlsson and E. Hagersten. Timestamp-Based Selective Cache Allocation. *In the Workshop on Memory Performance Issues*, 2001.

[11] S. Kaxiras, Z. Hu, M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. *Proc. of the International Symposium on Computer Architecture*, 2001.

[12] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache Replacement Based on Reuse-Distance Prediction. *Proc. of the International Conference on Computer Design,* 2007.

[13] M. Kharbutli and Y. Solihin. Counter-Based Cache Replacement Algorithms. *Proc. of the International Conference on Computer Design*, 2005.

[14] A. C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. *Proc. of the International Symposium on Computer Architecture*, 2000.

[15] J. Lee, Y. Solihin and J. Torellas. Automatically mapping code on an intelligent memory architecture. *Proc. of the International Symposium on High-Performance Computer Architecture*, 2001.

[16] W. F. Lin and S. K. Reinhardt. Predicting last-touch references under optimal replacement. *University of Michigan Technical Report*, 2002.

[17] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 1970.

[18] V. Milutinovic, B. Markovic, M. Tomasevic, and M. Tremblay. The Split Temporal/Spatial Cache: Initial Performance analysis. *Journal of Systems Architecture: the EUROMICRO Journal*, 1996.

[19] P. Petoumenos, G. Keramidas, and S. Kaxiras. Instruction based Reuse Distance Prediction for Effective Cache Management. *Proc. of the International Symposium on Systems, Architectures, Modeling, and Simulation*, 2009.

[20] P. Petoumenos, G. Psychou, S. Kaxiras, J.M. Cebrian Gonzalez and J.L. Aragon. MLP-Aware Instruction Queue Resizing: the Key to Power-efficient Performance. *Proc. of the International Conference on Architecture of Computing Systems, 2010.*

[21] T. Piquet, O. Rochecouste, and A. Seznec. Exploiting Single-Usage for Effective Memory Management. *Proc. of the Asia-Pacific Computer Systems Architecture Conference*, 2007.

[22] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. Emer. Adaptive insertion policies for high-performance caching. *Proc. of the International Symposium on Computer Architecture*, 2007.

[23] M. K. Qureshi, D. Lynch, O. Mutlu, and Y. N. Patt. A Case for MLP-Aware Cache Replacement. *Proc. of the International Symposium on Computer Architecture*, 2006.

[24] K. Rajan and R. Govindarajan. Emulating optimal replacement with a shepherd cache. *Proc. of the International Symposium on Microarchitecture*, 2007.

[25] S. T. Srinivasan and A.R. Lebeck. Load latency tolerance in dynamically scheduled processors. *Proc. of the International Symposium on Microarchitecture*, 1998.

[26] M. Takagi and K. Hiraki. Inter-Reference Gap Distribution Replacement: an Improved Replacement Algorithm for Set-Associative Caches. *Proc. of the International Conference on Supercomputing*, 2004.

[27] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. *Proc. of the International Symposium on Microarchitecture*, 1995.

[28] W. A. Wong and J. L. Baer. Modified LRU policies for improving second-level cache behavior. *Proc. of the International Symposium on High-Performance Computer Architecture*, 2000.