Cache Replacement Based on Reuse-Distance Prediction

Georgios Keramidas
University of Patras
keramidas@ece.upatras.gr

Pavlos Petoumenos
University of Patras
ppetoumenos@ece.upatras.gr

Stefanos Kaxiras
University of Patras
kaxiras@ece.upatras.gr

Abstract

Several cache management techniques have been proposed that indirectly try to base their decisions on cacheline reuse-distance, like Cache Decay which is a postdiction of reuse-distances: if a cacheline has not been accessed for some "decay interval" we know that its reuse-distance is at least as large as this decay interval. In this work, we propose to directly predict reuse-distances via instruction-based (PC) prediction and use this information for cache level optimizations. In this paper, we choose as our target for optimization the replacement policy of the L2 cache, because the gap between the LRU and the theoretical optimal replacement algorithm is comparatively large for L2 caches. This indicates that, in many situations, there is ample room for improvement. We evaluate our reuse-distance based replacement policy using a subset of the most memory intensive SPEC2000 and our results show significant benefits across the board.

1. Introduction

Despite the dedication of an increasingly larger portion of the chip area to cache hierarchies and the constant improvement of prefetching, main-memory access latencies still represent a significant factor of the performance loss in many applications. The problem is expected to become even more serious in CMPs where, on one hand, off-chip bandwidth becomes a serious bottleneck and, on the other, cache interference between different processes can be catastrophic. As a result, offering improved cache management is a necessity.

Reuse-distance analysis is a powerful tool to characterize the memory behavior of applications [4,13,14,15]. In our work, we measure reuse-distances as the number of memory accesses between two consecutive accesses to the same cacheline. Unlike stack distances, which measure the number of *unique* memory references between two consecutive accesses to the same cacheline, reuse-distances can be easily captured using functionality supported in today's hardware and OS [4].

In this paper, we propose the concept of reuse-distance prediction for run-time optimizations. We show that dynamic, instruction-based, reuse-distance prediction is feasible and we introduce **reuse-distance predictors** enhanced with the necessary confidence mechanisms. Our prediction affords us high accuracy with low complexity and low storage requirements.

We demonstrate run-time reuse-distance prediction by applying it to managing the replacement policy of L2 caches. The L2 is a critical element in the performance of all modern computers. It is the last line of defence before hitting the memory wall and experiencing the long latencies posed by main memory and off-chip busses. Thus, it becomes imperative to manage it for the best possible hit rate. The reason why the well known LRU algorithm is not good enough for the L2 is twofold.

First, L2 caches are typically highly associative which means 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. Lines with very large reuse-distances (which are likely misses) will still occupy useful space in the cache without contributing to the hit rate. Ideally, those lines should be replaced with lines with short temporal reuse-distance, even if such decision requires a circumvention in the time ordering introduced by the LRU algorithm. The second reason why LRU is not ideal for L2 caches is the filtering effect of the L1 caches. L2 caches are hidden behind L1 caches and accessed upon an L1 miss. This often inverts the temporal reuse patterns of the addresses as they are observed by the L2. These reasons indicate that L2 caches require more sophisticated replacement strategies than pure LRU replacement decisions.

2. Reuse-Distance Prediction

Reuse-Distance provides the foundation of our analysis. The reuse-distance of an address is defined as the number of intervening events —a notion of time— between two consecutive references to this address. The wall clock of our analysis is the memory references as they appear in the L1 data cache. Since our target is to use the reuse-distance prediction to take informed decisions for cache management, we consider reuse-distances at a cacheline granularity (assuming 64B block). Finally, we collect reuse-distances not with

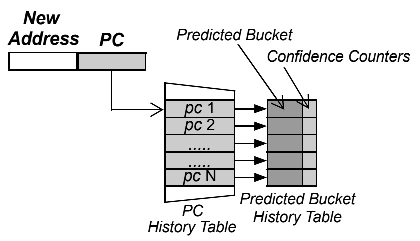

Figure 1. Structure of the Predictor.

their actual scalar value but rather as the log_2 of their value, which we will refer to as buckets (see Section 3).

In the following subsection, we describe a **new predictor** which is able to predict the next reuse-distance of a cacheline. The key idea is to relate a cacheline to the load/store instructions (PC) that access the relevant cacheline and make a prediction according to the instruction's previous behavior.

2.1. A New Predictor

Figure 1 depicts the structure of the instruction-based predictor. The predictor is composed of a Context Addressable Memory (CAM), called PC History Table (PCHT), that is responsible for storing active PCs. Non-active PCs are available for replacement. LRU or Decay can be used to manage refill. The second array (SRAM), indexed by the CAM, is the Predicted Bucket History Table (PBHT) which stores the predicted reuse-distance "bucket" for the corresponding PC. Attached to each predicted bucket is a field which contains information concerning the confidence for each prediction.

Given such a structure, the operation of the instruction-based predictor consists of two basic functions: *lookup* and *update*:

- *Lookup*: Lookup is the operation that predicts reuse-distances (buckets) for each PC. It is a straightforward procedure: the instruction (PC) that brings the new address is used as an input to the PCHT. A PCHT hit means that this PC has already appeared in the program execution while a miss indicates that this PC is encountered for the first time so no prediction can be made. In case of a hit, the PCHT selects the corresponding predicted bucket as well as the confidence of the prediction. Predictions are issued only if the value stored in the confidence counters is above a given threshold.
- *Update*: Update is the operation to refresh the PCHT and the PBHT when a reuse-distance of a previous address has been collected, so the history information of these tables is up-to-date to the extent possible. The inputs to this mode of operation are a PC and a newly measured reuse-distance. This information is traced by a hardware structure, called ***sampler***. Once the sampler relates a reuse-distance to a PC, the predictor is accessed for update. If no entry exists for the given PC, a new entry is added, the measured reuse-distance bucket is stored in the PBHT, and the corresponding confidence counter is cleared. If the predictor already contains information about the particular PC, we check if the new measured bucket and the bucket stored in the PBHT match. The match operation determines how should the confidence counters be updated (incremented or decremented). Finally, if the value stored in the confidence counters is zero and a mismatch occurs, then the bucket stored in the PBHT is considered as wrong and is replaced with the new recorded bucket.

The confidence counters are a crucial parameter of our design, since they affect the effectiveness of the predictor in terms of accuracy and coverage. In order to identify the appropriate parameters, we conducted an experiment varying their size as well as their threshold (we call this *safe limit*) above which the predictions can be considered as valid. Values below the safe limit indicate that the predictor is unable to make a prediction. In all cases, a predicted bucket is replaced only if the confidence counter is zero.

2.2. Effectiveness of the New Predictor

To show the predictable nature of the reuse-distances as well as the effectiveness of the proposed predictors, we select the 9 most memory intensive benchmarks of the SPEC2000 suite. Details such as benchmark initialization, phase skipping and processor configuration are given in Section 5.

Figure 2 depicts the results of our analysis. Every set of bars corresponds to a specific benchmark (shown at the bottom of the set) and every pair of bars (black and grey bars) reflects a run with different counter size and safe limit. The numbers above the benchmark names indicate the maximum value of the confidence counter and the values juxtaposed to the x-axis show the *safe limit* that we use in each run. Finally, the black bars show the measured accuracy normalized to the total number of the predictions issued in the whole trace execution, while the grey bars indicate the coverage of the predictions.

As we can see, different benchmarks exhibit different behavior when the size and the safe limit of the counters vary. *art* and *galgel* have a very stable behavior, reporting more than 85% accuracy and coverage irrespective of the counter configuration. *mcf, facerec* and *apsi* are also highly predictable achieving almost 70% of accuracy and coverage in most configurations. On the other hand, *ammp, gcc, twolf* and *vpr* are less predictable compared to the previous benchmarks and they are very sensitive to the counter's configuration.

For the rest of this paper, we consider a constant confidence mechanism (3 as the maximum value and 2 for the safe limit). Although this is not the optimal configuration over all the benchmarks, it is a compromise offering relatively high accuracy with a reasonable coverage.

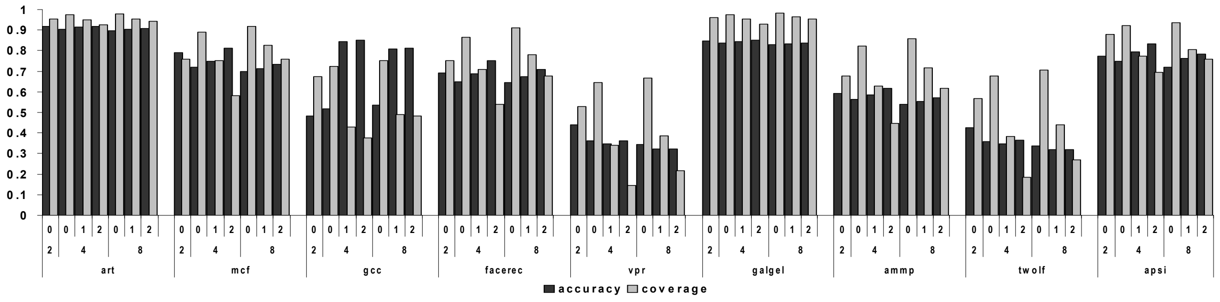

Figure 2. Accuracy and Coverage for various confidence mechanisms

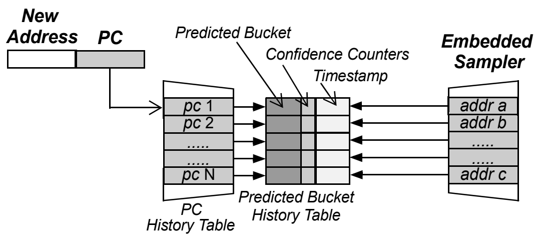

Figure 3. Embedded sampler in the predictor

3. Practical Implementations

A practical implementation of the proposed prediction must address a number of issues. In this section we detail realistic run-time implementations.

Predictor Size. The results presented in Figure 2 were collected using infinitely large prediction tables. We performed simulations to determine how a limited size predictor influences the accuracy and the coverage of the predictions. Our experiments have shown that, in all benchmarks (except from a slight drop in *apsi*), a 512-entry predictor is sufficient to provide the accuracy and the coverage presented in Figure 2 (where infinitely large prediction tables were used), but due to lack of space we do not present simulation results.

Under this scenario, the memory overhead introduced by our predictor is the following: 512 entries x 39 bits per entry (32 bits for each PC + 5 bits for the predicted bucket information + 2 bits for the saturating counters) = 2.5 KB, which is a reasonable overhead compared to a 512KB or 1MB L2 cache. Also, since the table is small, it is not latency sensitive and can be located on-chip.

Finally, the per-cacheline storage overhead is reasonable as well. Every cacheline should be 5 bits wider to accommodate its predicted bucket if available. Assuming an L2 cache with 64B cachelines, this overhead is equivalent to 1% additional memory.

Sampling. At first sight, the collection of the reuse-distances seems impractical for run-time implementations. Many addresses (corresponding to unverified predictions) must be traced, waiting for a match (next access) in order to identify their reuse-distance and associate it with an instruction (PC). This requires very large tables to store the addresses and additional hash tables to speed up the update operations of the predictors. Fortunately, previous work proved that there is no need to collect reuse-distances for all the addresses. By selecting a few addresses at random and trace only those addresses for their reuse-distance, it is possible to capture the memory behavior of the executable program [4,13]. Sampling allows us to keep only a small number of watchpoint registers and still be able to measure very large reuse-distances.

In our case we sample reuse-distances by tracing *only one* unverified prediction per PC even if further predictions are made for those PCs. Figure 3 shows an embedded sampler. An address CAM is coupled with the predictor. The address CAM matches addresses to unverified predictors. The sampler overhead besides the address CAM includes a "timestamp" field that is used to compute the reuse-distance as the difference of the current "time" to the timestamp. This raises the storage overhead of our base 512-entry predictor to 6.5 KB, which is still quite reasonable.

Quantization. The second fundamental technique that allows a practical implementation is the quantization of the reuse-distances to buckets. It would be impractical to store very large reuse-distances since the predicted reuse-distance should be stored in every line of the managed caches. This will lead to an unacceptable increase of the cache size, but quantizing the reuse-distances to buckets comprising power-of-two ranges, only a few additional bits per cacheline are required. We choose to use the following 20 buckets for quantization:

$[0][1:2][3:6]...[2^n-1:2^{n+1}-2]...[0.5M:1M][1M\text{-}inf]$

We refer the reader to [4] for more information about the sampling and the quantization techniques.

4. Managing Replacements

Having reuse-distance information for each cacheline allows us to approximate an optimal replacement algorithm because we can *"see"* the future. The idea is to replace the cacheline that is going to be used farthest in the future according to the reuse-distance information. There are a few caveats, however. These will be discussed below where we first explain what we are trying to accomplish in simple terms and leave the actual implementation details last.

Assume each time we access a cacheline we tag it with a timestamp and its reuse-distance prediction if available. On a miss we search the cache set. We look for the cacheline that is going to be accessed farthest in the future by computing its Estimated Time of Access *(ETA)*. The ETA of a line is the time it was last accessed plus its predicted reuse-distance minus the current time. In other words, a cacheline's ETA is the (predicted) number of accesses that separate the present moment from its next access (Figure 4). But what happens if we do not have reliable reuse-distance information about a cacheline, or no prediction at all? In this case we are conservative and since we have very low visibility into the future we prefer to look in the past. Thus, cachelines without reuse-distance prediction or whose ETA is negative —the predicted access time has passed— receive an ETA of 0 as if they were going to be accessed immediately.

Assume now that we select the line with the ETA farthest in the future. Is this enough for making a replacement decision? The answer would be yes if we had *full* information about *all* the cachelines. Since we are dealing only with *partial* information, the line with the largest *known* ETA is not necessarily the best candidate for replacement. Thus, we need to also look in the past. For this we use the notion of Decay to compare the importance of cachelines for replacement. The difference of the current time and a line's last access time is the *Decay time* of a line, i.e., how long the line has remained unaccessed (Figure 4). The longer a line remains unaccessed the higher the probability that it is useless. Cache Decay [7] is based on this principle. This line of reasoning is also consistent with LRU: the line with the longest Decay time is the LRU.

We now have two candidates, quantified by exactly the same metric: time measured in accesses. One candidate is the line with the largest ETA (the ETA line), and the other is the line with the largest Decay time (the LRU line). We pick the largest of the two for replacement (Figure 4). This guarantees that if we have reliable information about large reuse-distances, the replacement decision will be based on those; otherwise we revert to LRU replacement.

The *conceptual* description of our replacement algorithm relied on timestamps, calculations (to find ETA and Decay times), and comparisons; rather complicated to implement. In fact, a *realistic* and efficient implementation does not need to use timestamps or perform calculations. The implementation is based on *hierarchical decay counters* that measure time automatically as in [7]. The global counter is incremented by processor accesses. An important change we introduce is that our local cacheline counters count in exponential steps. This is accomplished by triggering each successive transition of the local cacheline counters from a successively more significant bit of the global counter. To reduce overhead we can truncate a few low-order bits (e.g., 6 to 8) from the local counters reducing the resolution of time for small reuse-distances and decay times without a significant impact on results.

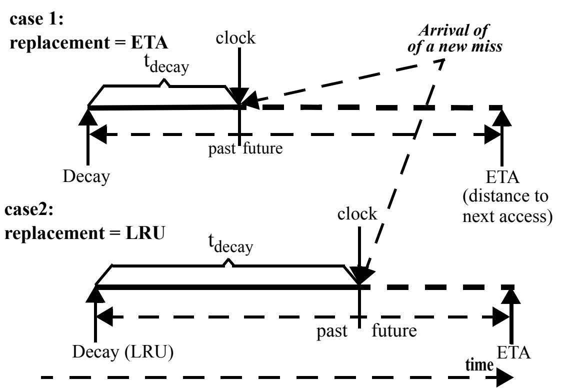

Figure 4. The Proposed Replacement Algorithm

Both the Decay time and the ETA of a line are local cacheline counters. Decay counters start counting when a line is accessed and are read when are needed for replacements. The ETA counters are set whenever a reuse-distance prediction is made for a cacheline and count backwards towards 0 (where they stop). It is evident that at replacement time the ETA counters will contain the time *left* for the next access and the Decay counters the time that *passed* since the last access. The remaining functionality consists of selecting the largest ETA and Decay in the set and then selecting between them.

5. Evaluation

We performed our experiments using detailed cycle accurate simulations. Our baseline processor is a dynamic 8-way superscalar processor. We simulate a 16K, 64B block, 4 way, dual-ported, 1 cycle L1 data cache and 8-way, 15 cycle, unified L2 cache of various sizes. The main memory has a 250 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. The instruction-based predictor utilizes 2-bit confidence counters and a prediction is considered safe if the value stored in the confidence counters is greater or equal to two.

For this study, we select a subset of the SPEC2000 suite for which Belady's optimal algorithm [1] exhibits more than 5% miss rate reduction over the LRU. As a result the applications under optimization are *art*, *mcf*, *gcc*, *facerec*, *vpr*, *galgel*, *ammp*, *twolf*, and *apsi*. These applications are also the most memory intensive applications in the SPEC2000 suite. We simulate 200M instructions for every benchmark after skipping 2B instructions for *mcf* and *vpr*, 3B for *ammp*, and 1B for the rest of the benchmarks.

Figure 5 shows the results of our experiments in terms of miss rate reduction compared to the LRU for

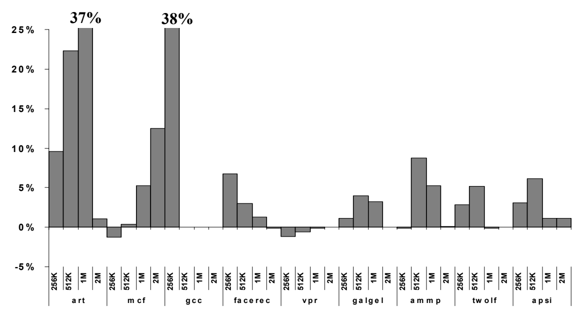

Figure 5. Miss rate Reduction

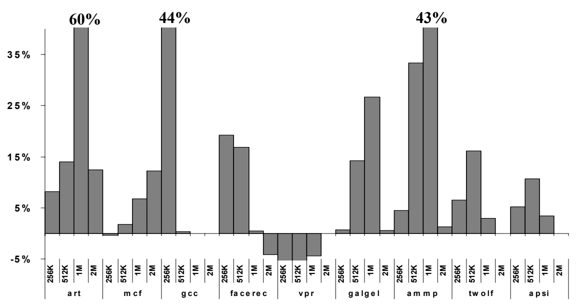

Figure 6. IPC improvement

various L2 cache sizes. Our replacement policy reports significant improvements for the majority of the benchmarks and cache sizes. More specifically, in *art,* our proposal reduces the misses by 37% in the 1M cache, 22% in the 512K cache, and 9.6% in the 256K cache. However, there is no performance benefit for the 2M cache. *art* is an interesting benchmark for cache management. According to [13], *art* is characterized by large working sets, reporting very high miss rates, but this trend is inverted in caches larger than 2M, since art fits in these caches and so there is no space to enforce better replacement decisions. On the other hand, our algorithm does very well in reducing the number of misses in the 1M cache. In this case, we manage to lower the miss rate to 25%, while the LRU reports 62% miss rate. The 1M case offers the best improvement ratio, because the possibility for a cacheline to be replaced promptly before it is used is maximized. Hence, the potential benefit of better replacement decisions is significant (many misses turn into hits).

gcc follows a similar behavior although in smaller caches. A 512K cache is able to accommodate its memory requirements, so the 256K cache is the best candidate for management. In fact, *gcc* experiences the greatest miss rate reduction (almost 40%). Actually, the miss rate reduction is totally correlated with the application's working set size relative to the cache size. Going to larger caches, the improvement decreases because most of the benchmark's working sets fit in the cache. Under this scenario, *ammp, galgel, twolf,* and *apsi* show the best improvements in the 512K cache (8.8%, 4%, 5.1%, 6.1% respectively), while *facerec* offers the higher reduction in the 256K cache (6.8%). However, the situation differs in *mcf. mcf* is a very memory intensive application and a 2M cache is still not able to accommodate its working set. As a result, the best improvements are achieved in larger caches (10% miss rate reduction in the 2M cache).

Finally, *vpr* is the only benchmark where our management lags behind the LRU. The reason for this reduction —albeit very small— is that the specific confidence mechanism used in those experiments reports only 14% coverage and 34% accuracy. Of course, there is always the opportunity of increasing accuracy and coverage by choosing a different predictor configuration or creating more sophisticated predictors.

Figure 6 shows the percentage of IPC improvement over the LRU, for various cache sizes and for the applications that we consider in this analysis. Our reuse-distance based replacement policy manages to speedup all applications (except *vpr*). *art*, *gcc*, and *ammp* show the greatest performance gains —for specific cache sizes. A 60% speedup is achieved in *art,* 43% in *ammp* (in the 1M cache), and 44% in *gcc (*in the 256K cache). Our proposal reports considerable speedups in the other benchmarks as well, albeit in different cache sizes. As we have already mentioned, the improvement in each benchmark is related to the cache size. The best improvement (over all cache sizes) is 19% in *facerec*, 26.7% in *galgel*, 16% in *twolf*, and 10.6% in *apsi*. On the other hand, the LRU algorithm performs better than our proposal in *vpr* which shows an almost 6% slowdown as a result of selecting the specific confidence counter configuration.

6. Previous Work

Replacement Algorithms. The limits of the cache replacement algorithms were initially studied by Belady [1] who formulated the area by comparing random replacement algorithm, LRU and an optimal replacement algorithm that looks into the future. In this work, program traces were used and it was more of a way to provide a better understanding of the cache behavior rather than to implement a real cache and related replacement algorithms. Since then many researchers proposed schemes aiming to provide better replacement decisions. A very popular scheme is the LRU-K algorithm [2] which is observes past reuse distances. Jeong and Dubois introduce the idea of cost sensitive replacement algorithms [5]. The authors proposed different variations of the LRU by assigning finite costs between load and store misses. A different approach is presented by Karlsson and Hagersten [3]. Their method reduces the miss rate by carefully selecting L1 blocks.

Another class of replacement techniques involves the exploitation of the generational behavior of a cache block [7]. According to the authors, it is possible to identify dead blocks in the cache as well as to predict the time at which the next reference to a cacheline will occur. Takagi and Hiraki further explore the predictable nature of the time between two consecutive uses of the same cacheline and provide a replacement algorithm which collects access interval distributions [10]. However, their algorithm requires very large per-cacheline counters and complex computational effort. Abella et al. [8] proved that the original cache decay methodology, although very successful in the context of the L1, is not directly applicable to the L2. The authors propose flexible event counters to adapt the decay interval.

Most of the work in the context of the L2 caches focuses either in the identification of dead lines (called last touch prediction) or in cache bypassing schemes. In these studies, decisions are made either based on the program counter (PC) of the instruction that generates the cache access [9], or on the memory address of the access [12]. The later techniques suffer from low accuracy, long training times and require significant storage to yield replacement benefits. The former techniques were inspired by the work of Lai and Falsafi [11] who used the PC-based last-touch prediction information in the context of multiprocessor cache coherent protocols. Compared to these approaches, our instruction-based (PC) prediction predicts the reuse-distance of the memory references which allows us to directly decide which cache block is going to be accessed farthest in the future.

Predictability of Reuse Distances. Reuse-distance analysis has proven to be a good mechanism to predict the memory behavior of the programs. Previous work has shown that both whole program [14] and instruction-based [15] reuse-distances can be predicted accurately across program inputs using a few profiling runs. Recently, Fang et al. [15] use profiling and then statically predict reuse-distances across various data sets in order to estimate whole-program miss rates for L1 and L2 caches. While this paper is the first to collect reuse-distance information at run time and not at compile time, the proposed predictors are trained by profiling runs and used statically for analysis. In contrast, we propose dynamic, instruction-based, reuse-distance prediction that directly applies its predictions to optimize the execution of the (running) program.

7. Conclusions

In this work we demonstrate that the instruction-based reuse-distance prediction can be used for cache optimizations. We propose a simple, small and highly accurate reuse-distance predictor which are triggered by the instructions that touch cachelines and we evaluate the accuracy and the coverage of the predictors. In this paper, we choose to apply the proposed approach in improving the replacement decisions in the L2 cache. Our experiments, using cycle accurate simulations, report a significant increase of the overall performance in the most memory intensive applications of the SPEC2000 suite. For specific cache sizes, we obtained a 60% speedup in art, a 44% in *gcc* and 43% in *ammp*. A modest, although noticeable, speedup is observed in the rest of the applications. The storage overhead of the predictor is negligible compared to a large L2 cache (less than 2% in total). We believe the concepts presented here can be the source for many cache-level optimizations.

Acknowledgments. This work is supported by EU FP6 IP-FET SARC No. 27648, and the EU FP6 HiPEAC NoE IST-004408. Georgios Keramidas is supported by a PENED 2003 grant; Pavlos Petoumenos is supported by a “K. Karatheodoris” grant.

8. References

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

[2] E. J. O’Neil, P. E. O’Neil, and G. Weikum. An Optimality Proof of the LRU-K Page Replacement Algorithm, Journal of the ACM, 1999.

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

[4] P. Petoumenos, G. Keramidas, H. Zeffer, S. Kaxiras, and E. Hagersten. Modeling Cache Sharing on Chip Multiprocessor Architectures. Proc. of the International Symposium on Workload Characterization, 2006.

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

[6] W. A. Wong and J. L. Baer. Modified LRU policies for improving second-level cache behavior. Int. Symposium on High-Performance Computer Architecture, 2000.

[7] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. Proc. of the 29th International Symposium on Computer Architecture, 2002.

[8] J. Abella, A. Gonzalez, X. Vera, and M. O’Boyle. IATAC: a Smart Predictor to Turn-Off L2 Cache Lines. ACM Transactions on Architecture and Code Optimization, 2005.

[9] W. F. Lin and S. K. Reinhardt. Predicting last-touch references under optimal replacement. University of Michigan Technical Report (CSE-TR-447-02), 2002.

[10] 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.

[11] 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.

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

[13] E. Berg and E. Hagersten. Fast Data-Locality Profiling of Native Execution. Proc. of the ACM SIGMETRICS, 2005.

[14] Y. Zhong, S. Dropsho, and C. Ding. Miss rate prediction across all program inputs. Proc. of the Conference on Parallel Architectures and Compilation Techniques, 2003.

[15] C. Fang, S. Carr, S. Onder, and Z. Wang. Instruction Based Memory Distance Analysis and its Application to Optimization. Proc. of the International Conference on Supercomputing, 2006.