Instruction-based Reuse-Distance Prediction for Effective Cache Management

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

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

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

***Abstract*-The effect of caching is fully determined by the program locality or the data reuse and several cache management techniques try to base their decisions on the prediction of temporal locality in programs. However, prior work reports only rough techniques which either try to predict when a cache block loses its temporal locality or try to categorize cache items as highly or poorly temporal. In this work, we quantify the temporal characteristics of the cache block at run time by predicting the cache block reuse distances (measured in intervening cache accesses), based on the access patterns of the instructions (PCs) that touch the cache blocks. We show that an instruction-based reused distance predictor is very accurate and allows approximation of optimal replacement decisions, since we can "see" the future. We experimentally evaluate our prediction scheme in various sizes L2 caches using a subset of the most memory intensive SPEC2000 benchmarks. Our proposal obtains a significant improvement in terms of IPC over traditional LRU up to 130.6% (17.2% on average) and it also outperforms the previous state of the art proposal (namely Dynamic Insertion Policy or DIP) by up to 80.7% (15.8% on average).**

I. Introduction

Caching is widely used in almost all computing systems, and cache performance decidedly determines system performance due to the gap between the speed of the processor and main memory. Computer architects attack this problem by offering an increasingly larger portion of the silicon area to cache hierarchies (currently 40-60%) and by devising sophisticated prefetching and replacement mechanisms, but still main memory access latencies are a significant factor in the poor performance of many applications. Furthermore, the shift from frequency scaling to scaling the number of cores, not only does not solve the problem but rather increases the reliance on on-chip caches. As a result, offering improved cache management is a venue to make better use of on-chip transistors, increase the apparent capacity of caches, and reduce the impact of long memory latencies.

The effect of caching is fully determined by the program locality or the data reuse patterns. As the memory hierarchy becomes deeper and deeper its performance increasingly depends on our ability to predict program locality. Previous work discloses mainly two ways of locality analysis: compile-time analysis [33,4,6,5] of the program source code (i.e. loop nests), which is not as effective for dynamic control flow and data indirection, and profiling [2,3,19] which analyzes the program for a given number of selected inputs, but may fail to capture possible changes in other inputs. Ideally, a prediction scheme is needed that can be used at *run-time* and can be both efficient and accurate.

To this end, in this paper, we show that it is possible to directly predict the *reuse distances* of the memory references via *instruction (PC) based prediction* at *run-time*. We introduce a new class of reuse-distance predictors enhanced with the required confidence mechanisms. We explore the structure of the predictors in terms of the required size and the associativity and indicate several ways to increase their effectiveness. We found (using all the benchmarks of the SPEC2000 suite) that measuring the reuse distances in L2 accesses (using a binary distribution granularity for low storage requirements) afford us high accuracy in predicting the temporal characteristics of the memory references (when a memory reference is going to be accessed in the future).

In this work, we demonstrate run-time reuse-distance prediction by applying it to managing the replacement policy of the L2 caches (the last level of the on-chip hierarchy in our experimental environment). While L1 caches are designed with simplicity and fast access time in mind, L2 caches are less sensitive to latency, and focus more on minimizing expensive accesses to memory. Thus, the L2 cache is of paramount importance in all modern computers, since it is the last line of defence before hitting the memory wall and experiencing the long latencies imposed by the main memory and by off-chip busses. But in L2 caches, it is well documented that the widely implemented LRU policy is far from optimal in many applications [22,32,16,23,26].

The reason 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. Cache blocks 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 blocks should be replaced with blocks with short temporal reuse distance, even if such decision requires a circumvention in the time ordering introduced by the LRU algorithm (e.g., [23,26]). 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 decisions and many researchers turn their attention in providing schemes to predict the temporal characteristics of the cache items [18,12,22,32,16,23]. However, all the previous schemes try to

predict the temporal locality of the cache blocks in a binary manner (as temporal or non temporal) or try to predict when a cache block will lose its temporal locality. In contrast to previous approaches (as we will show later), our instruction-based predictor allows us to fully quantify the temporal locality of the memory accesses. The output of the predictor is the reuse distance of the cache blocks using a power-of-two distribution (measured in L2 accesses). As a result, direct comparisons among the temporal locality of the cache blocks are possible, which distinguishes our scheme as a fine-grained approach, in contrast to previously proposed coarse-grain approaches [18,12,22,32,16,23]. The main contributions of this work are the following.

- We systematically examine reuse-distance prediction and show that its success depends on both the access stream and the way reuse distances are measured at the cache level of interest (in our case the L2). Specifically, we show that we can obtain great improvements over prior approaches, in both accuracy and coverage, by tracking for the prediction only a *filtered* L2 access stream discounting irrelevant events such as writebacks, and at the same time measuring reuse distances in terms of the intervening *filtered* L2 accesses.
- We study practical predictor organizations, concerning size, associativity and embedded confidence estimation, and show that the PC prediction requirements in terms of hardware are quite small (under 2% of the total L2 size).
- In addition, we study efficient techniques to collect reuse distances at run-time (reuse-distance sampling), something that has been largely glossed over in prior work.
- We apply the improved prediction mechanisms in a hybrid replacement mechanism which automatically reverts to LRU when it is required. Having reuse-distance information for each cacheline in a set would allows us to approximate *optimal* replacement decisions by looking into the future. However, lack of prediction information does lot limit our mechanism since we can also look into the past: the longer a cacheline remains unaccessed the higher the probability that it is useless [15]. Our algorithm dynamically picks one of the two candidates (the one indicated by "past" information and the one indicated by "future" prediction). An extension of the algorithm is to victimize the currently fetched block by not caching it at all in the L2 (*selective caching*).
- Finally, we thoroughly evaluate our approach for all SPEC2000 benchmarks and for various L2 cache sizes. Our results indicate a significant speed up in 8 out of the 26 SPEC2000 applications by up to 130.6% and 17.2% on average, while not slowing down any of the remaining applications by more than 1%. Furthermore, we compare our approach to a recently proposed scheme (namely Dynamic Insertion Policy or DIP [23]) and we show that we surpass it terms of IPC by up to 49.6% (10% on average) and by up to 80.7% (15.8% on average) when the selective caching approach is employed.

The remainder of this paper is organized as follows. Section II presents the motivation of this work and clarifies the correct framework to correlate the reuse distances of the cache blocks with the memory access instructions. Section III provides evidence of this correlation and presents our instruction based reuse distance predictor. Section IV discusses and quantifies the effectiveness of the predictors using practical, low overhead sampling techniques. In Section V, we evaluate our methodology in improving the replacement decisions of the L2 caches. In Section VI, we put this work in context of related work. Finally, Section VII concludes the paper.

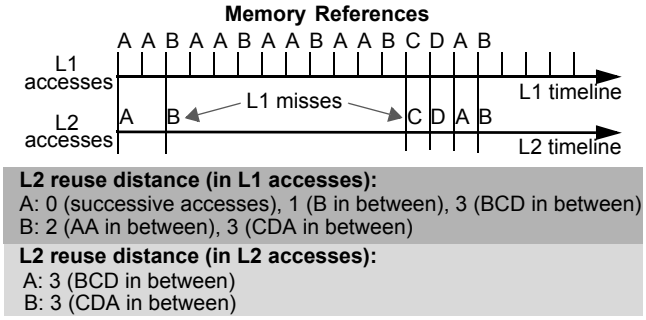

Figure 1. Depending on how we measure reuse distances at different cache levels we can get different behaviors for the same cache blocks

II. Measuring Reuse Distances

In this work, we attempt to relate the reuse distance of cache blocks to memory-access instructions. Our premise is that, due to program locality and the regular nature of loops, certain frequent memory-access instructions tend to access cache blocks that exhibit predictable reuse behavior.

To proceed with this kind of prediction we need to clarify two questions:

- At what cache level do we measure the reuse distances?
- How do we measure reuse distances?

The problem is complicated because of the multilevel cache hierarchy: there are multiple ways to associate instruction (PC) information to access information (in our case reuse distance information) at the different cache levels. For example, one can use the number of intervening (memory-access) instructions as a measure of reuse distances —in other words, one can count as a reuse distance of a cache block the number of loads/stores executed between two successive accesses to the cache block.

This would be fine as long as our target cache is the L1. However, this type of measuring of the reuse distances would not be appropriate for managing the L2 cache. This is because such reuse distances correspond to how frequently a cache block is accessed by the instruction stream and not how they are accessed in the L2. The L2 access stream is the stream of misses and writebacks coming out of the L1 and this gives to the (L2) cache blocks a totally different reuse distance behavior than what we can infer from the processor accesses [16,23].

Consider the example in Figure 1 which shows the accesses to a single 2-way set in the L1 and the corresponding accesses to the L2. Cache blocks A, B, C, and D are accessed by the processor resulting in the eventual eviction of A and B from the L1 set (because of C and D) and their re-fetch. Measuring the reuse distance of A using processor accesses we end up with three different reuse distances (which might actually hamper prediction). In any case it seems that the behavior of A is different than the behavior of B. In contrast, measuring their

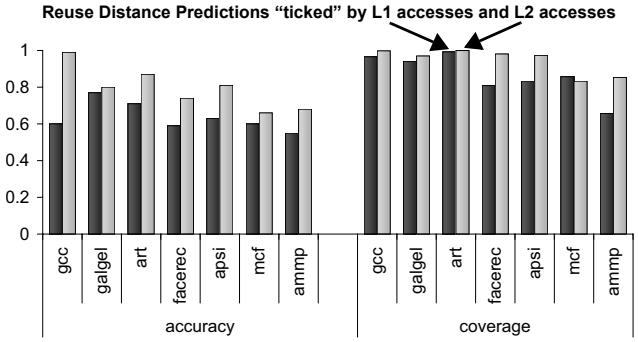

Figure 2. Accuracy and coverage compared to previous work

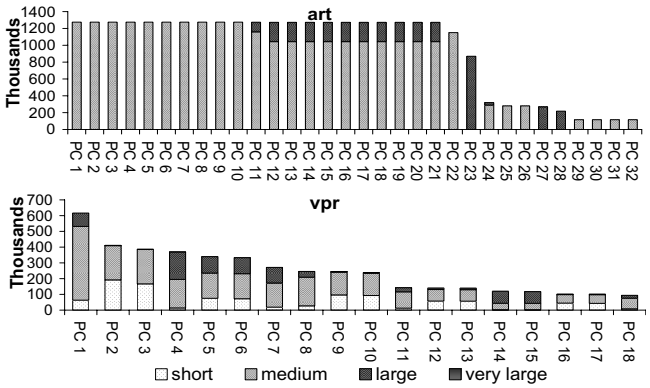

Figure 3. Collected reuse-distances for *art* and *vpr* (corresponding to 98.5% of the total accesses to the L2)

behavior using L2 accesses the two blocks seem equivalent. For the purpose of managing the L2 it is the latter that matters.

While previous work made no distinction which access stream is used to count reuse distances, here we argue that for managing a specific cache level, one must rely on the access stream of this level to measure the reuse distances. To prove this point, Figure 2 compares accuracy and coverage for making reuse-distance predictions in the L2, but measuring the reuse distances in two different ways: with processor accesses (L1 accesses), and with L2 accesses. Without getting into the details of the prediction mechanisms, reuse distances are associated with the instructions (PCs) that access the L2 and predictions are performed based on history. It is evident from Figure 2 that markedly improved results can be expected by choosing the right "counter" for reuse distances.

A. Filtered access streams

While for the processor access stream one can expect that each access is genuinely related to program behavior, this is *not so* for all the accesses below the L1 in the cache hierarchy.The reason is that an eviction from a cache level can appear as an *access* at a lower level if it is a writeback (i.e., appears as a write at the lower level). An eviction, however, is not related to the instruction that caused it in the first place. In other words, accesses corresponding to writebacks cannot be associated with any specific instruction. Failure to see this effect can pollute reuse-distance prediction with random information and seriously degrade the results. It is therefore necessary to filter the access stream from such irrelevant events. *Prediction can be performed only for the read accesses in the levels below the L1 and not for the writes*. Our improved results in Figure 2 assume such filtering.

III. PREDICTION REQUIREMENTS

One of the motivations for correlating cache block reuse distances to the instructions (PCs) that access them is that the number of instructions involved is relatively small. To show this we examined all the programs of the SPEC2000 suite. For illustration, we show the collected reuse distances for two representative benchmarks —*art* and *vpr*. We specifically selected these benchmarks, because they exhibit different reuse-distance behavior.

We run each program for 250M instructions after the necessary skipping (see Section V for more details). We concentrate on the L2 and collect the reuse distances for each instruction (PC) that accesses the L2. Reuse distances are measured in L2 accesses using a filtered access stream, as described in Section II.A. We consider reuse distances up to 1M intervening accesses, which means that a reuse distance has a storage requirement of 20 bits. Reuse distances larger than this are represented with the maximum value. In many cases, we can use a reduced resolution for the reuse distances, discarding a few low-order bits as we will see in Section V.

Figure 3 plots the results of this experiment. For reasons of clarity, we present the PCs that are responsible for the 98.5% of the total accesses to the L2. In addition, we categorize the reuse distances in four groups. The group tagged as "short" (see Figure 3) contains the number of the reuse distances with values between 2^0 and 2^{10}. Addresses in this group are characterized with short temporal reuse patterns and are expected to experience more cache hits. In contrast, the group tagged as "very large" represents the number of cache blocks with very large reuse distances (2^{19} to 2^{20}). These addresses are primary candidates for management (in the case of creating a replacement algorithm, these addresses should be replaced even if they reside in the top of the LRU stack). The "medium" and the "large" group correspond to the number of the cache blocks with reuse distances between 2^{11} to 2^{14} and 2^{15} to 2^{18}, respectively.

As we can see, for both programs, there is a need to capture the reuse distances of only a few instructions in order to take a representative picture of the benchmark's memory behavior. *art* needs 32 instructions, while in *vpr*, 18 PCs are capable to capture its memory reuse-distance patterns. In any case, the limited number of the required PCs reveals that our proposal can be integrated in a real system with low overhead.

art shows a straightforward per-instruction reuse distance histogram contributing to our initial argue about the predictable nature of the reuse distances via instruction based prediction. The bulk of the PCs touch cache blocks with "medium" reuse distances. However, some PCs generate cache blocks that are either "medium" or "large". The "large" cache blocks are primary candidates for replacement, because it is expected to be cache misses, so their early replacement can free up useful space in the cache. The situation in *vpr* is not as clear. Unique PCs are associated with a variety of reuse distances. However, optimization hints still exist (since there are cache blocks with very large reuse distances). The problem

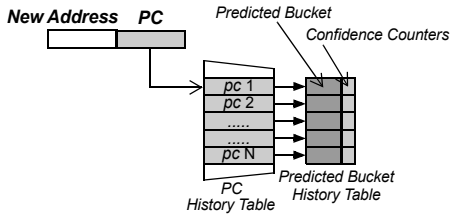

Figure 4. Structure of the Instruction Based Predictor

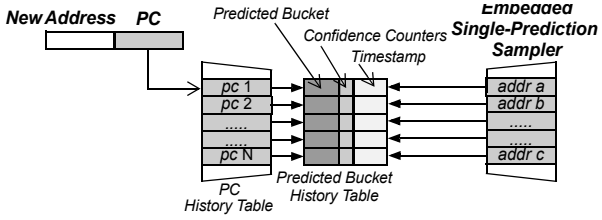

Figure 5. SPS reuse distance sampling mechanism

is to identify these addresses, but this subject is covered in the following subsections.

The results of our analysis of the SPEC2000 benchmark suite indicate that a simple structure, such as the one shown in Figure 4, is well suited to handle the instruction-based prediction of reuse distances. An instruction accessing a cache-block at the target cache level initiates an access to a history table indexed by the PC of the instruction. The history table provides the recorded reuse-distance for this PC, along with an estimated confidence for the prediction. Reuse distances are collected with a ***sampler***, which is the subject of the next section.

Because it is unlikely that reuse distances are predictable at a very fine resolution (i.e., it is unlikely to encounter the exact same reuse distance over and over again) we consider predictions as *correct* when the observed and predicted reuse distances are of the *"same magnitude."* There are several ways to implement such a scheme and we have chosen to use a power-of-two magnitude comparison with good results. Thus, for the update of the predictor, we consider as *equivalent* the reuse distances whose log2 values are the same.

According to our study a predictor size of 256 entries is more than enough to capture all the significant PC's in a typical program phase. Additional entries can hold information for rarely executed load/stores that may not be easy to predict and do not significantly improve the effectiveness of our prediction.

Furthermore, aliasing effects in the predictor that can potential taint history information can be removed by making the structure associative (lowering the power consumed by the predictor as well). We examined several associativities for the predictor structure and we concluded that an 8-way associative structure is almost as good a fully associative organization in terms of performance. Finally, our predictor design includes confidence counters for its predictions. A confidence counter per entry determines whether it is *"safe"* to attempt a prediction ("safe threshold" —see Section V). The confidence counters are incremented with each correct prediction and decremented with a wrong prediction. The predictor entry is allowed to change its reuse distance prediction (prediction replacement) only if its confidence counter is 0.

IV. Practical Reuse-Distance Collection at Run-Time

The predictor structure discussed in the previous section can deliver predictions based on its stored history information about reuse distances. An additional mechanism is needed, however, to collect this history and update the predictor. Verifying the predictions increases the confidence of the prediction entries. In contrast, refuting the predictions decreases confidence and (depending on the setup of the confidence mechanism) allows the update of the predictor entries with the new observed reuse distances.

Naively, a mechanism to achieve this would be to track each prediction until it is verified or refuted. We call such a mechanism a *"full sampler"* (FS). This means that for *every* access for which we have a prediction, the actual reuse distance of the accessed block must be determined. The actual reuse distance is then compared to the prediction and the predictor entry for the corresponding PC is updated.

Obviously, the state of such a tracking mechanism is enormous. The problem is that very large (actual) reuse distances —of the order of a million accesses— require that the corresponding prediction must be remembered for a very long time. Note that, by nature, predictions for reuse distances are verified out-of-order. That is, a prediction for a long reuse distance is verified far into the future, even if subsequent predictions for the same PC (but for smaller reuse distances) can be verified sooner. This is inherent in all our tracking schemes.

To determine the actual reuse distance for a prediction we must remember when it was made (wall-time measured always in L2 accesses). A structure holding all the outstanding predictions (yet to be verified) needs to be searched fast and often: with every cache access we must check whether a previous prediction exists for the cache block address. One could consider a very large CAM for this job, but that would be impractical due to area, power, and speed concerns. An alternative would be to implement a large hash table, but that would exacerbate the storage problem. Clearly, something much more practical is needed for the run-time prediction of reuse distances.

Our solution is to piggyback the tracking mechanism on to the predictor entries. Each predictor entry, corresponding to a single PC, has the ability to store one outstanding (yet unverified) prediction. Until this prediction is verified (or refuted) the predictor entry can deliver new predictions (for the subsequent invocations of the same PC), but the predictor entry cannot be updated. We call this *"Single-Prediction Sampler"* (SPS). The structure of the SPS in relation to the predictor is shown in Figure 5. Each entry in the SPS is linked to a single predictor entry. Every accessed address is checked against the addresses in the sampler.[1] A match determines the reuse distance for the corresponding predictor entry.[2] Although this sampler can skip many updates, its performance holds up very well against the full sampler (see Figure 7 for a comparison). But, there are some cases that need special attention.

[1] This can be organized as a CAM or a set-associative structure. The latter case introduces a few storage conflicts but for associativity of 8 or more the effects are insignificant.

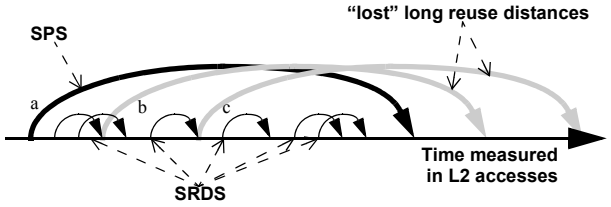

Figure 6. Handling of reuse distances by the SRDS and the SPS

The problem with SPS is a scenario where a prediction for a long reuse distance is followed by a string of short reuse distances. Assuming that the long reuse distance prediction is correct, it will be verified when we see again the same cache block far into the future. Since the predictor entry does not change until then, subsequent predictions will also give long reuse distances, *even though the behavior of the subsequent accesses changes to short reuse distances*. All such predictions would be incorrect, without hope of adjusting the prediction. A full sampler, on the other hand, would have no problem correcting this situation. In the full sampler, even if there is an outstanding long-reuse-distance prediction, short reuse distances would be verified a lot sooner —*"under"* the outstanding prediction— and would update the predictor entry to deliver correct (short-reuse-distance) predictions.

While, for its simplicity, the SPS gives good results, we need to safeguard against such situations. To this end, we add a very small sampler intended to capture short reuse distances that we would otherwise ignore, waiting for a long reuse distance to be verified. We call this the "*Short-Reuse-Distance Sampler*" (*SRDS*) and it works only with conjunction with an SPS.

SRDS is simply a small FIFO sampling the L2 access stream with a sampling rate of 1/*N*. It randomly picks one out of *N* (on average) accesses and stores the prediction made on this access (a link to the predictor entry). Because it is a FIFO its size and sampling rate determine the maximum reuse distance it can capture: *max reuse distance = size × sampling period.* For example an 8-entry FIFO sampling every 256 accesses can "see" the reuse distances of up to 8×256=2048.

SRDS works in parallel to the SPS and can update the predictor entry independently when it detects small reuse distances. However, it introduces a bias towards small reuse distances. This is because SRDS can see many short reuse distances both because they are quick to detect *and* because SRDS can overlap their detection. In contrast, SPS may take a long time to verify a *single* long reuse distance and in the mean time no other long reuse distance can be detected (cannot be overlapped). This situation is shown in Figure 6. SRDS is able to capture all the short reuse distances (even if they overlap). SPS can deal with a single long reuse distance at a time. The result is that predictions with the long reuse distances labeled b and c in the figure go undetected (SRDS cannot see them, SPS cannot overlap them).

[2] Reuse distances in all our schemes are represented up to a maximum of 1M accesses (20 bits). If we have not seen a match within this reuse distance we eagerly update the predictor entry (with the maximum reuse distance) allowing the verification of a new prediction. The mechanism for this is similar to *decay* [15].

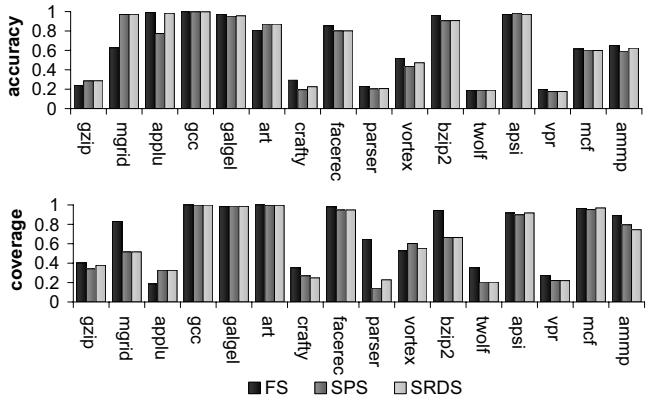

Figure 7. Comparison of FS, SPS, and SPS+SRDS (tagged as SRDS) in terms of accuracy and coverage

There is, however, a way to account for such "lost" long reuse distances: if an entry falls out of the SRDS FIFO without ever seeing a reuse distance then we assume that it corresponds to a *long reuse distance*. SRDS balances its updates using an accounting scheme that tallies the long and the short reuse distances it sees. For each long reuse an update with a short reuse distance is inhibited. The tallies are kept in the corresponding predictor entry.

Given this analysis Figure 7 compares the accuracy and coverage of the prediction for FS, SPS and SPS with SRDS. It is evident that SPS alone performs well compared to FS (or even better, e.g., in *gzip*, *mgrid* and *apsi* —because it discards noise) and in the cases SPS lags behind FS, SRDS can make up for the lost ground (e.g., in *applu*, *crafty* and *ammp*).

Finally, the memory overhead introduced by our predictor and sampler is the following: 256 × 71 bits per entry[1] or 2.3Kbytes, which is a reasonable overhead compared to a 512Kbyte or 1Mbyte L2 cache. Also, since the table is very small, it is not latency sensitive and it can be located on-chip. Obviously, the memory requirements of the 8-entry SRDS are negligible.

V. Reuse-Distance Prediction Applications

Ideally, what we would like to do in a replacement algorithm for the L2 is to employ Belady's optimal replacement in which we replace the cache block that is going to be accessed farthest in the future. Reuse distance prediction allows us to approximate this because it gives us a way to estimate the next access for a cache block *relative to the anticipated accesses for other cache blocks in a set*. Note that this is the only mechanism that allows such quantitative comparisons [22,32,16].

Assuming that we have a prediction of the reuse distance for every cache block in a set, it is straightforward to implement comparisons between the anticipated future accesses. Reuse distance prediction can be stored with each cache block, along with an indication of the time the cache block was last accessed. At any moment, the estimated arrival of the next access to a cache block is computed as the difference between the predicted reuse distance and the time (measured in L2

[1] 26b recorded address +32b PC +5b predicted bucket +5b time stamp in power-of-two granularity + 2b confidence counter + valid bit = 71b

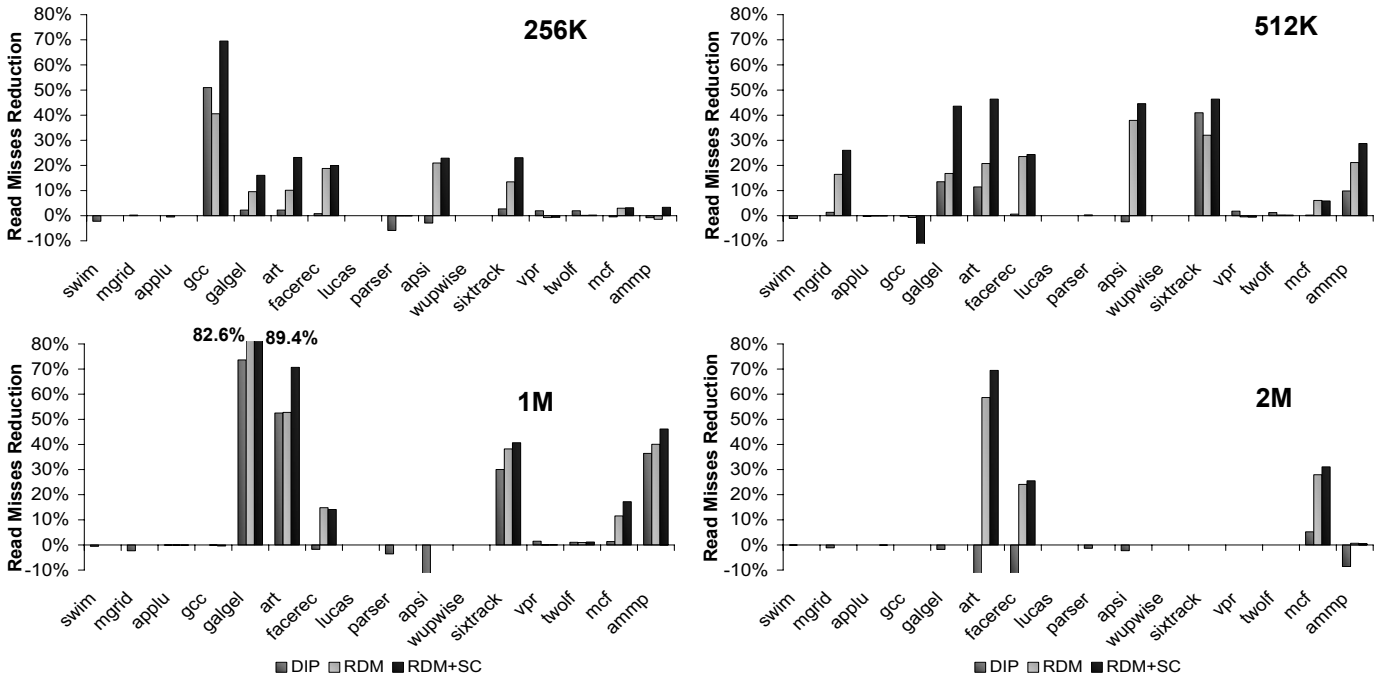

Figure 8. Read misses reduction results for 16 SPEC2K benchmarks with more than 1 miss per 1K-instructions.

accesses) the cache block has been resident in the set. The cache block with the farthest estimated next access is the victim for a replacement.

While the above description requires arithmetic calculations and comparisons, the same effect can be achieved by storing the predicted reuse distance with each cache block (when it is last accessed) and then decrement it with each intervening access as time passes. It is evident that at each point in time the remaining reuse distance corresponds to the time left for the predicted next access. An important realization here is that storing and manipulating the reuse distances at a reduced resolution (for example discarding 4 to 8 of their low-order bits) has little effect on the quality of management we can achieve. In such a case we decrement the reuse distances every *2^4 to 2^8 intervening L2 accesses, which corresponds, on average, to thousands or tens of thousands of cycles*. This corresponds to similar power overhead as with cache decay counters which was shown to be negligible [15].

The above scenario works well only if we have a prediction for all the cache blocks in a set. Since reuse distance prediction does not provide complete coverage (i.e., a prediction for each access), in many cases we may have cache blocks without predictions. To handle these situations, we compare the estimated time for the next access for the blocks for which we do have a prediction to the time blocks have remained *unaccessed* in the cache for those blocks for which we do not have a prediction. In other words, we either look into the future for the next access or try to estimate whether a block has *decayed* [16] by staying unaccessed for a long time. An additional decay counter that is incremented with every access is required. The decay counter operates similarly to the reuse distance prediction counter and at the same resolution. Among all blocks we select for replacement the one with the largest next-access time (which is the candidate for the optimal replacement) or the largest decay time (which is the LRU block).

An enhancement of the above algorithm is to take into account the currently fetched cache block for victim selection. If the currently fetched block has a prediction for a reuse distance that exceeds the time-to-next-access or the decay time of all the blocks in the set, then it can be chosen not be inserted into the cache instead of selecting one of the residing blocks for eviction. We call this feature *Selective Caching,* or *SC*.

A. Simulation Setup

Our experiments were performed using a detailed cycle accurate simulator that supports a dynamic superscalar processor model. Our baseline processor is a 4-way superscalar processor with an 80-entry reorder buffer. 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. The instruction-based predictor utilizes 2-bit confidence counters and a prediction is attempted —considered *safe*— if the value stored in the confidence counters is greater or equal to two.

In order to show the effectiveness of our approach we compare it to the recently proposed replacement algorithm called Dynamic Insertion Policy or DIP [23]. DIP is a very successful scheme requiring negligible hardware cost (less than 2 bytes[1]). The underlying idea of DIP is to prevent the

[1] Recall that our approach requires 2.3Kbytes (less than 1% of the total cache size), but we do not consider it as an important overhead in the high performance computing area.

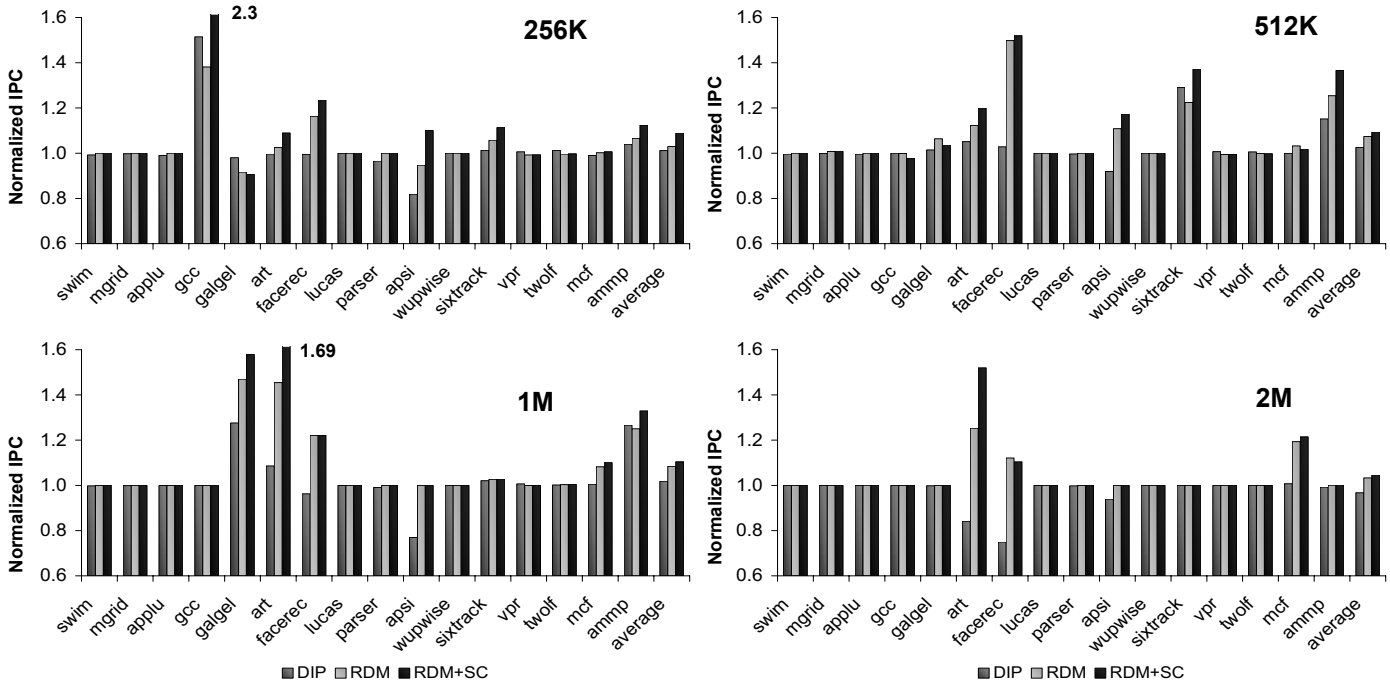

Figure 9. Normalized IPC results for 16 SPEC2K benchmarks with more than 1 miss per 1K-instructions.

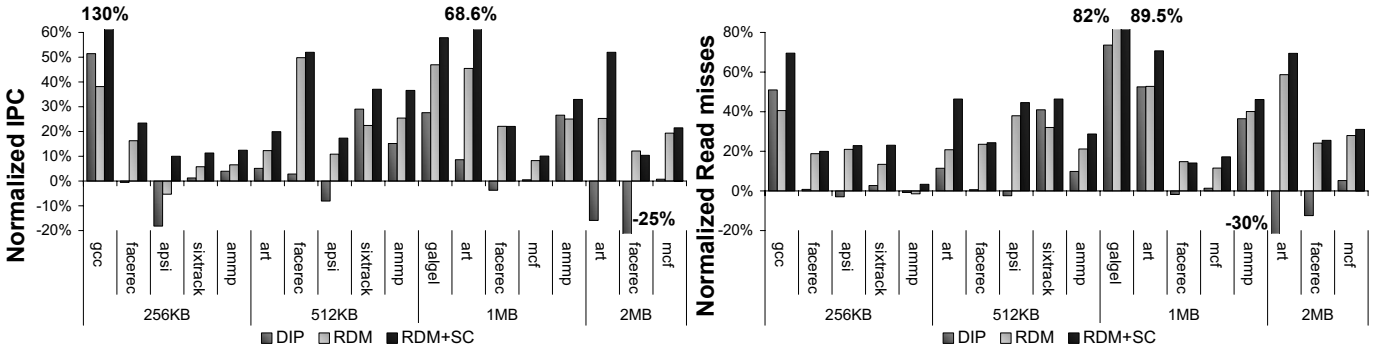

Figure 10. IPC improvement (left) and Read misses reduction (right) results for DIP, RDM, RDM+SC

cache lines that are not expected to exhibit temporal locality from not staying in the cache for long. This is done by dynamically adapting the position of the LRU stack at which the cache blocks are inserted. We have implemented the DIP approach in our simulation framework by carefully porting the code distributed by the authors [8].

To directly compare with prior work, we run *all* the applications of the SPEC2000 benchmarks (both data and computational intensive programs). But in the interest of clarity we do not show results for benchmarks which have less than 1 miss per thousand instructions. Such benchmarks have low cache requirements and do not benefit from advanced replacement algorithms. We note here that *our* proposal does not negatively affect their performance. This leaves 16 benchmarks (shown all in Figure 8 and Figure 9) that can be affected by replacement algorithms. Of those 16 benchmarks only eight actually show significant performance improvement (or degradation in several cases for DIP): *gcc, galgel, facerec, apsi, sixtract, ammp, art,* and *mcf* and we concentrate on these. The simulations were performed after the necessary skipping instructions for every benchmark. We simulate 200M instructions after skipping 3 billion instructions for *ammp,* 2B for *mcf*, 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 and after that period we start to collect statistics.

B. Performance Evaluation

We performed the simulations for several L2 cache sizes (256K to 2M) to assess the effects of cache management for each program in different conditions. Our results show that for the cache sizes where LRU performs adequately compared to an optimal replacement, further management does not contribute much. Similar conclusions are reached in several other papers [16,21,22,23,32]. However, in the cases where LRU does not fare well, then the opportunities for management are significant. Figure 8 and Figure 9 show the complete results in terms of IPC reduction and read misses reduction for

the 16 memory-intensive benchmarks and for all cache configurations.

For clarity Figure 10 concentrates on the cases where management works best (256K L2 for *gcc*, *facerec*, *apsi, sixtrack,* and *ammp*, 512K for *art, facerec, apsi, sixtrack*, and *ammp*, 1M for *galgel*, *art*, *facerec, mcf,* and *ammp* and 2M for *art, facerec,* and *mcf*). Incidentally, these are the cache sizes where the working sets of these programs show an inflection point, and begin to "fit" in the cache of the corresponding size [2]. Management in either smaller or larger caches makes less difference, since the working sets either do not fit at all or fit in their entirety.

Figure 10 depicts the performance improvements —in terms of IPC— (left-side graph) and the reduction in the performance-critical read misses (right-side graph) over the LRU for the 8 SPEC benchmarks. The first bar in each set of bars represents the IPC improvements (or read-miss reduction) when the DIP replacement algorithm is employed (the best performing replacement policy previously published), while the second and the third bar illustrate the results for the Reuse Distance Management (RDM) and RDM enhanced with Selective Caching (SC).

As we can see from Figure 10, our results indicate that in the presented cases the benefits are substantial for both the RDM and the RDM+SC case compared either to the LRU or even the DIP approach. In the RDM case (without selective caching), *facerec* (in 512K cache) shows the greatest performance gains over the LRU (48.9%), while significant speedups are reported in the other benchmarks as well. A 38.1% speedup is achieved in *gcc* (256K cache), 25.4% in *ammp*, 22.4% in *sixtrack* (512K cache), 47% in *galgel*, and 45.6% in *art* (1M cache). The best improvement for *mcf* is achieved in the 2M cache (up to 19.3% increase in IPC). As we have already mentioned, the improvement in each benchmark is related to the cache size. *mcf* is a cache greedy application and is the most memory intensive program of the SPEC2000 suite and according to [2], a 2M cache is still not able to accommodate its working set.

Compared to the DIP replacement algorithm, our approach is not always superior. RDM still outperforms DIP by 10% on average and in no case degrades performance as much as DIP (up to 25%), but there are cases (*gcc*-256K, *sixtrack*-512K, and *ammp*-1M) where DIP performs better than RDM. Still, when the selective cache technique is employed (RDM+SC), our approach clearly outperforms DIP: RDM+SC increases IPC relative to DIP by up to 80.7% and 15.8% on average.

C. Power and Area Issues

Finally, we evaluate our prediction structures (SPS+SRDS) in terms of power (energy and EDP) and area overhead – common metrics in the low power embedded processor world. We used a modified version of the Cacti 5.3 tool [30] to estimate the power and area characteristics of the proposed predictors. Our Cacti estimates showed that the power consumption of all the structures that we introduced is approximately 10mW (peak power) which translates into less than 0.024% increase of the processor's energy consumption for all our experiments (0.0032% on average). Consider the area, SPS+SRDS area requirements correspond to less than 1% of the area occupied by a 256KB L2 cache (even less for larger caches) which is negligible as well. As a result, we believe that both the energy and the area overhead of the SPS and SRDS predictors is minimal and affordable for every low power processor which is equipped with an L2 cache, rendering our proposal a viable solution in the embedded system world.

VI. Related Work

In this work, we argue that it is possible to directly predict the temporal characteristics (reuse distances) of the memory references via run-time instruction-based prediction and exploit this information for cache replacement. We will briefly overview related work in each of these directions.

Predictability of Reuse Distances. Reuse distance analysis was mainly explored with great success at compile time [33,4,6,5] or during a profiling step [2,3,19] in order to understand and improve the temporal locality of the programs. The reuse distance of a memory reference, unlike stack distance, can easily be captured using functionality supported in today's hardware and operating systems [21].

Previous approaches show that both whole-program [33,4] and instruction-based [6] reuse distances can be predicted accurately across program inputs using a few profiling runs. Recently, Fang et al. [5] associate the prediction of the reuse distances to the program's data set. As a result, they were able to predict the reuse distances (at compile time) across various data sets and estimate the whole program miss rates of the L1 and the L2 caches. Our approach is the first that we are aware of that utilizes the instruction-based prediction at run-time and not at compile time.

Cache Management. Early work studied the limits of cache replacement algorithms using program traces. The first attempt was by Belady [1] who formulates the area by comparing random replacement algorithms, LRU and an optimal algorithm that looks into the future. Sugumar and Abraham [27] extended the Belady's algorithm in order to characterize capacity and conflict misses, while Temam [29] used the Belady's optimality results by simultaneously exploiting spatial and temporal locality. All those studies were more of a way to provide a better understanding of the cache behavior rather than to implement a real cache and related replacement algorithms.

In the area of L1 cache management, many techniques categorize, and handle accordingly, the memory references based on their temporal and spatial characteristics [20,7,13,14]. Jeong and Dubois introduce the idea of cost sensitive replacement algorithms [10]. The authors proposed different variations of the LRU by assigning finite costs between load and store misses. In [11], the costs were associated with the type (read or write) of the next access to a block. In [24], the idea of Memory Level Parallelism (MLP) aware cache replacement was introduced.

Another class of replacement techniques [16,28] 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 [9,15].

Most of the work in the context of the L2's focuses either in the identification of dead lines (e.g., last touch prediction), or in providing cache bypassing schemes [18,12,22,31,17,32]. A similar approach, called dynamic insertion policy [23], tries to prevent cache blocks that are not expected to exhibit temporal

locality from not 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 (blocks without temporal locality are directly inserted in the LRU position). We compare against these techniques and show that i) we outperform them in the vast majority of the cases and ii) they are not as stable as our proposal: while such techniques reduce the number of misses, they do not necessarily have a positive impact on performance.

Finally, a recent approach, called shepherd cache [25], attempts to emulate the Belady's optimal algorithm for a conventional cache. In this work, the large, highly associative cache is replaced by a cache with lower associativity (the main cache) and use a secondary (shepherd) cache to approximate an optimal replacement policy in the main cache. However, this approach requires very large per-cache line additional storage area (up to 46 bytes) and it is characterized by a complex replacement policy.

VII. Conclusions

Many cache management techniques have been proposed in the past that were based on rough assessments of the temporal locality of cache blocks. Those techniques either try to predict the temporal locality of the cache blocks in a binary manner ("have" or "have-not") or try to predict when a cache block will lose its temporal locality. In this work, we propose a practical way to quantitatively compare the temporal locality of individual cache blocks via their predicted reuse distance, which distinguishes our scheme as a fine grained approach, in contrast to previously proposed coarse grain approaches. We show that it is possible to predict with high accuracy the reuse distance of the cache blocks via instruction (PC) based prediction at run-time. We concentrate on practical implementations of the instruction based predictors (concerning size, associativity and embedded confidence estimation) and we study efficient techniques to collect reuse distances at run-time (reuse-distance sampling). We demonstrate our reuse distance prediction methodology by applying it to managing the replacement policy of the L2 caches and we show that a significant increase of the programs's performance can be achieved. On top and orthogonal to this, we use our instruction based reuse distance predictor to enforce a selective caching technique which further improves the resulting performance gains.

Acknowledgments

This work is supported by the EU FP6 Integrated Project - Future and Emerging Technologies, Scalable computer ARCitecture (SARC) Contract No. 27648, and the EU FP6 HiPEAC Network of Excellence IST-004408. Pavlos Petoumenos is partially supported by "K. Karatheodoris" grant awarded by the Research Committee of the University of Patras. The equipment used for this work is a donation by Intel Corporation under Intel Research Equipment Grant #15842.

We would also like to the thank Prof. Andy Pimentel, as well as the reviewers of the paper, for their helpful comments.

References

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

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

[3] C. Cascaval and D. A. Padua. Estimating cache misses and locality using stack distances. *Proc. of the International Conference on Supercomputing (ICS)*, 2003.

[4] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[5] 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 (ICS)*, 2006.

[6] C. Fang, S. Carr, S. Onder, and Z. Wang. Reuse-distance-based miss-rate prediction on a per instruction basis. *Conference System Performance*, 2004.

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

[8] http://users.ece.utexas.edu/~qk/dip/

[9] Z. Hu, S. Kaxiras, M. Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. *Proc. of the International Sympocium on Computer Architecture (ISCA)*, 2002.

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

[11] J. Jeong, P. Stenstrom, and M. Dubois. Simple, Penalty-Sensitive Replacement Policies for Caches. *Computing Frontiers*, 2006.

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

[13] 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.

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

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

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

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

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

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

[20] 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.

[21] 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.

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

[23] 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 Sympocium on Computer Architecture (ISCA)*, 2007.

[24] 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 (ISCA)*, 2006.

[25] K. Rajan and R. Govindarajan. Emulating optimal replacement with a shepherd cache. *Proc. of the Interbational Sympocium on Microarchitecture (MICRO)*, 2007.

[26] R. Subramanian, Y. Smaragdakis, G. Loh. Adaptive Caches: Effective Shaping of Cache Behavior to Workloads. *Proc. of the International Symposium on Microarchitecture (MICRO)*, 2006.

[27] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. *Proc. of the Conference on Measurement and Modeling Computer Systems*, 1993.

[28] 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 (ICS)*, 2004.

[29] O. Temam. An algorithm for optimally exploiting spatial and temporal locality in upper memory levels. *IEEE Transactions on Computers*, 1999.

[30] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. Cacti 5.1. *HP Laporatories Technical Report (HPL-2008-20)*, 2008.

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

[32] 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 (HPCA)*, 2000.

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