2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Improving Cache Management Policies Using Dynamic Reuse Distances

Nam Duong[†], Dali Zhao[†], Taesu Kim[†], Rosario Cammarota[†],
Mateo Valero[§] and Alexander V. Veidenbaum[†]
[†]*University of California, Irvine* [§]*Universitat Politecnica de Catalunya, Barcelona Supercomputing Center*
{nlduong, daliz, tkim15, rcammaro, alexv}@ics.uci.edu, mateo.valero@bsc.es

Abstract

Cache management policies such as replacement, bypass, or shared cache partitioning have been relying on data reuse behavior to predict the future. This paper proposes a new way to use dynamic reuse distances to further improve such policies. A new replacement policy is proposed which prevents replacing a cache line until a certain number of accesses to its cache set, called a Protecting Distance (PD). The policy protects a cache line long enough for it to be reused, but not beyond that to avoid cache pollution. This can be combined with a bypass mechanism that also relies on dynamic reuse analysis to bypass lines with less expected reuse. A miss fetch is bypassed if there are no unprotected lines. A hit rate model based on dynamic reuse history is proposed and the PD that maximizes the hit rate is dynamically computed. The PD is recomputed periodically to track a program's memory access behavior and phases.

Next, a new multi-core cache partitioning policy is proposed using the concept of protection. It manages lifetimes of lines from different cores (threads) in such a way that the overall hit rate is maximized. The average per-thread lifetime is reduced by decreasing the thread's PD.

The single-core PD-based replacement policy with bypass achieves an average speedup of 4.2% over the DIP policy, while the average speedups over DIP are 1.5% for dynamic RRIP (DRRIP) and 1.6% for sampling dead-block prediction (SDP). The 16-core PD-based partitioning policy improves the average weighted IPC by 5.2%, throughput by 6.4% and fairness by 9.9% over thread-aware DRRIP (TA-DRRIP). The required hardware is evaluated and the overhead is shown to be manageable.

1 INTRODUCTION

Reduction in cache miss rates continues to be an important issue in processor design, especially at the last level cache (LLC). Cache management policies such as replacement, bypass or shared cache partitioning, have been relying – directly or indirectly – on the data reuse behavior to improve cache performance. The future reuse is predicted based on past behavior and thus may not necessarily be accurate. In addition, such policies do not define an accurate cache performance model based on the reuse information and thus cannot achieve their full performance potential. This paper proposes a way to address these issues for the above-mentioned cache management policies and focuses on the LLC.

Let us start with the replacement policy. Many such policies have been proposed for the LLC [1, 6, 14, 18, 19, 20, 24, 31] aiming to improve over the LRU replacement policy, the most widely used replacement policy, which has been shown [33, 29, 24] to have anomalous behavior for applications whose working set is larger than the LLC. The newer policies are often adaptive and use heuristics based on predicted future memory reference behavior for cache line replacement. A widely used estimate of the future behavior is the observed reuse distance.

Many ways of reuse distance prediction and its use by replacement heuristics have been proposed. For instance, EELRU [29] accurately measures the reuse (stack) distance but uses a probabilistic model to predict which cache line should be evicted on a cache miss. RRIP [14] approximates the reuse distances as near, long or distant future and tries not to replace lines which are predicted to be reused sooner. IGDR [31] builds an accurate reuse distance distribution but replacement is based on a "weight" function, with a line of smallest weight replaced. However, the weight function does not necessarily reflect future behavior of a line. The counter-based algorithm [19] uses a counter matrix to predict when lines are reused. The approach in [17] predicts reuse distances using program counters and evicts lines with the longest remaining distance (more in Sec. 7).

A more precise knowledge of reuse would allow a replacement policy (1) not to evict lines too early, before their next reuse point, and at the same time (2) not to keep the lines in the cache for too long to avoid cache pollution. A better "balance" between (1) and (2) can improve the policy performance. Sec. 2.1 presents a case study of this phenomenon, and the replacement policy proposed in this paper indeed achieves such balance.

The *reuse distance* (RD) used in this paper is defined as the number of accesses to a cache set between two accesses to the same cache line (Sec. 7 compares this definition to others). A *reuse distance distribution* (RDD) is a distribution of RDs observed in a program at a given time. It is a unique signature of a program or a phase for a given cache configuration. Fig. 1 shows the RDDs of several SPEC CPU2006 benchmarks (see Sec. 5 for our measurement methodology). As RDs can be very large, this paper limits the *maximum measured distance* d_{max} to 256. The fraction of RDs below the d_{max} is shown as a bar on the right of each figure.

Using the RDD to direct prediction, lines can be kept only until a desired level of reuse is achieved *and* cache pollution is minimized. For example, in Fig. 1, in 436.cactusADM enough lines are reused at or below the RD of 64, increasing the RD beyond 64 leads to cache pollution. The problem, however, is how to find an RD balancing reuse vs. pollution and how to use it to manage replacement. This paper proposes a way to solve this problem.

Similar to prior work [19, 31], this paper measures the RDs in execution and builds the RDD dynamically. The RDD is used to

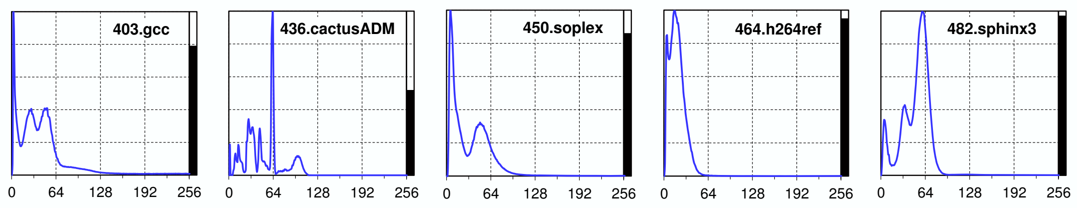


Figure 1. Distribution of reuse distances for selected benchmarks.

compute an RD that maximizes the cache hit rate, i.e., the RD that balances the opportunity for reuse with timely eviction. This RD is called a *Protecting Distance* (PD) and our replacement algorithm "protects" a line for this many accesses to its set before it can be evicted. This policy is thus called the *Protecting Distance based Policy* (PDP).

It can be said that the LRU policy protects a line for W unique accesses before eviction, where W is the cache associativity. However, W may not be the RD that maximizes the hit rate. PDP achieves the same effect for protecting distances PD $\leq W$, i.e. in LRU-friendly benchmarks. But the PDP also works for PDs $> W$ using the same algorithm. Thus PDP does not need to switch between LRU and another algorithm as has been done in prior work [29, 24, 14, 17]. Note also that EELRU [29] can be said to "protect" the late eviction point by evicting any "older" lines first, if present (see Sec. 7).

The RDD can also be used to manage a cache bypass policy in non-inclusive or exclusive caches. Typically, the bypass is used to place a line in an upper level cache and avoid duplicated use of storage in the lower level cache. However, it may be better to store a line with a lot of reuses in the lower level. A bypass policy that makes such a choice intelligently can, in fact, have a better performance than a strictly inclusive cache. The PDP can be combined with the bypass policy to enhance the replacement policy by protecting cache lines from early eviction. The non-inclusive and exclusive caches are already used in commercial products, such as the AMD processors [8], and several policies to support bypass have been proposed [6, 13, 18, 19, 5]. The work in [10] proposed a policy to achieve performance similar to bypass in an inclusive cache, and the algorithms in [6] are developed specifically for exclusive caches.

Finally, this paper describes a PD-based shared cache partitioning policy that is based on the measured RDDs of different threads and a selection of PDs for the threads such that the shared cache hit rate is maximized.

The paper discusses the hardware support required to measure the reuse distance, compute the RDD dynamically, determine the hit rate and find the PD. The hardware was synthesized to evaluate the overheads and is shown to be feasible.

In summary, the main contributions of this paper are:

1. The concept of protecting distance PD and a policy for protecting cache lines, which is adaptive to changes in program phases and memory access behavior.
2. A hit rate model for a single-core processor as a function of reuse distances, which is shown to accurately predict the cache behavior of an application.
3. A method and a hardware design to compute the PD that maximizes the hit rate.
4. A bypass policy that guarantees a desired level of reuse in the LLC and is integrated with the replacement policy.
5. A multi-core shared LLC hit rate model that is a function of PDs of individual threads.
6. A shared cache partitioning policy based on reuse distances, which computes a set of PDs, one per thread, to maximizes the overall hit rate.

The proposed policies were evaluated and compared to a number of prior policies. For single-core replacement PDP was compared to DIP, DRRIP, EELRU and sampling dead-block prediction (SDP). Shared cache partitioning for multi-core processors was compared to TA-DRRIP, UCP and PIPP.

The rest of the paper is organized as follows. The PD concept is defined in Sec. 2 together with a hit rate model to quantitatively relate the PD and the RDD. The hardware design is in Sec. 3. The hit rate model for multi-core cache partitioning is defined in Sec. 4. The experimental methodology is in Sec. 5. The experimental results are in Sec. 6. Prior work is discussed in Sec. 7. Appendix A reports results for SPEC CPU2006 benchmarks.

2 SINGLE-CORE PROTECTING DISTANCE

This section defines the protecting distance PD for a single-core processor. A replacement policy using the PD and a combined replacement and bypass policy are presented. A hit rate model for the policy is also described.

2.1 Protection – A Case Study

Let us start with a case study using RRIP policy [14], which was recently proposed to improve the performance of applications with a working set which is larger than the LLC. RRIP is able to preserve a portion of the working set in the cache. It tackles the thrashing and scanning issues by predicting that most of the inserted lines are re-referenced (reused) in a *distant* future, the rest inserted lines in a *long* future, and the reused lines in a *near* future.

Now consider RRIP from the perspective of *protection*. The re-reference future of lines is predicted using a parameter, ϵ. This is the probability that inserted lines are predicted to be reused in a long future. In RRIP, ϵ is experimentally set to 1/32. This means that an inserted line which is classified as being reused in the long future can be protected for 32 misses. In RRIP with 2-bit storage per line, on average a reused line can be protected for as long as $32 \times (2^2 - 1) = 96$ misses (and an even larger number of accesses)[1].

Fig. 2 shows the behavior of dynamic RRIP (DRRIP) as a function of ϵ for four benchmarks (483.xalancbmk.3 is a certain window of execution, see Sec. 5 for details). ϵ is varied from 1/128 to 1/4 and two trends can be observed. For 436.cactusADM and 483.xalancbmk.3 decreasing ϵ increases the number of misses. However, for 403.gcc and 464.h264ref a higher ϵ can reduce misses. This means that, in the latter two benchmarks, lines should not be

[1]These numbers vary during runtime depending on access behavior.

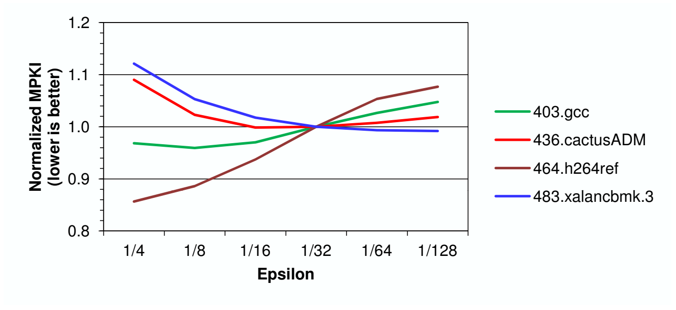


Figure 2. DRRIP misses as a function of Epsilon ϵ.

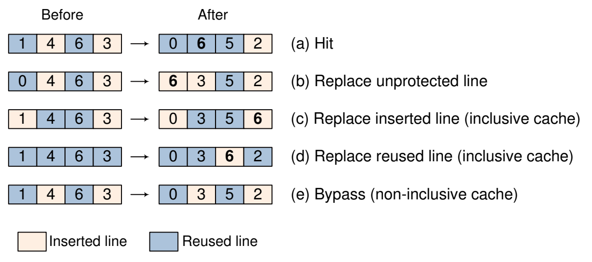

Before
After
1 4 6 3 → 0 6 5 2 (a) Hit
0 4 6 3 → 6 3 5 2 (b) Replace unprotected line
1 4 6 3 → 0 3 5 6 (c) Replace inserted line (inclusive cache)
1 4 6 3 → 0 3 6 2 (d) Replace reused line (inclusive cache)
1 4 6 3 → 0 3 5 2 (e) Bypass (non-inclusive cache)
Inserted line
Reused line

Figure 3. Example of the PDP cache operation.

protected too long if they are not reused. Yielding cache space to new lines improves performance. Sec. 2.3 will analyze these benchmarks in more detail.

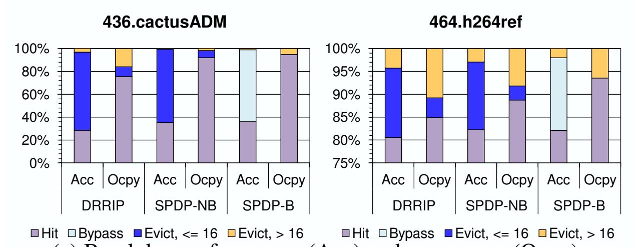
2.2 The Protecting Distance

The *protecting distance* PD is a reuse distance that “covers” a majority of lines in the cache, that is they are reused at this or smaller distance. It is a single value used for all lines inserted in the cache. The PD is the number of accesses to a cache set that a line is protected for, i.e. the line cannot be evicted until after PD accesses. The PD is used as follows. Each cache line is assigned a value to represent its *remaining PD* (RPD), which is the number of accesses it remains protected for. This distance is set to the PD when a line is inserted or promoted. After each access to a set, the RPD of each line in the set is decremented (saturating at 0). A line is protected only if its RPD is larger than 0. An unprotected line is chosen as the victim. Victim selection when there are no unprotected lines is different for inclusive and non-inclusive caches.

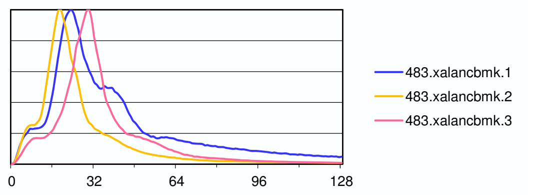
In an inclusive cache a line with the highest RPD, i.e. the youngest inserted or promoted line, is chosen in order to protect older lines. Similar to DIP and RRIP, the fact that inserted lines are less likely to be hit than reused lines is utilized. An inserted line with the highest RPD is replaced or, if there are no such lines, a reused line with the highest RPD is chosen. This requires a “reuse bit” to distinguish between inserted and reused lines.

For a non-inclusive cache, the cache is bypassed if no unprotected lines are found. This is based on the prediction that the protected lines are more likely to be reused in a the nearer future than the missed lines. Note that the bypassed lines are inserted in a higher-level cache. Here the bypass policy and the replacement policy work together to further protect lines as opposed to only the replacement policy in the inclusive cache. The reuse bit is not required here.

The example in Fig. 3 illustrates the PDP policy in a cache set with four lines. Let us assume that the current predicted PD is 7. In the figure, the number inside each box is the RPD of the corresponding cache line. There are 5 possible scenarios in this example.



(a) Breakdown of accesses (Acc) and occupancy (Ocpy).



(b) RDDs in 3 windows of 483.xalancbmk.

Figure 5. Case studies.

For each scenario, RPDs before and after the access are shown. In Fig. 3a, the access results in a hit to the second line. In the other 4 cases, the access results in a miss leading to victim selection. In Fig. 3b, the victim is the unprotected line. Fig. 3c and 3d are for a cache without bypass. In Fig. 3c, the inserted line with the highest RPD is evicted, while in Fig. 3d there are no inserted lines and the reused line with highest RPD is evicted. Fig. 3e is for the cache with bypass, where the cache is bypassed as there are no unprotected lines. In all cases, the RPD of the promoted or inserted line is set to 7, and the RPDs of all lines, including the new line, are decremented.

2.3 Evaluation of the Static PDP

The static PDP (SPDP) uses a constant PD throughout program execution. The SPEC CPU2006 benchmarks were simulated with static PDs between 16 (the associativity) and $d_{max} = 256$. The PD which minimizes misses per 1K instructions (MPKI) varies from benchmark to benchmark. Even in the three simulation windows of 483.xalancbmk, the best PDs are different (see Appendix A for details). Fig. 4 compares the static PDP with the best PD for non-bypass (SPDP-NB) and bypass (SPDP-B) policies to DRRIP with the best ϵ (as described in Sec. 2.1).

First, better DRRIP performance can be achieved with a “dynamic” ϵ. It is significant for 403.gcc, 450.soplex and 464.h264ref. Second, both SPDP-NB and SPDP-B can further reduce misses over DRRIP, by as much as 20% for SPDP-NB and by 30% in SPDP-B in 464.h264ref. Third, SPDP-NB and SPDP-B have different behaviors in different benchmarks. For example, in 436.cactusADM the miss reduction is similar, whereas for 483.xalancbmk.3 SPDP-B has a significantly higher reduction than SPDP-NB. In general, SPDP-B achieves a higher miss reduction than SPDP-NB and both perform better than DRRIP.

The PDP aims to protect cache lines long enough to be reused, but not for too long to avoid pollution. Let us define the *occupancy* of a line as the number of accesses to its cache set between an insertion or a promotion and the eviction or the next promotion and analyze accesses and occupancy for two benchmarks, 436.cactusADM and 464.h264ref. The accesses and occupancy are shown in Fig. 5a, each broken into promoted lines and evicted lines. The latter is further divided into lines which are evicted before 16 accesses and all the rest The fraction of bypassing for SPDP-B is also shown. The

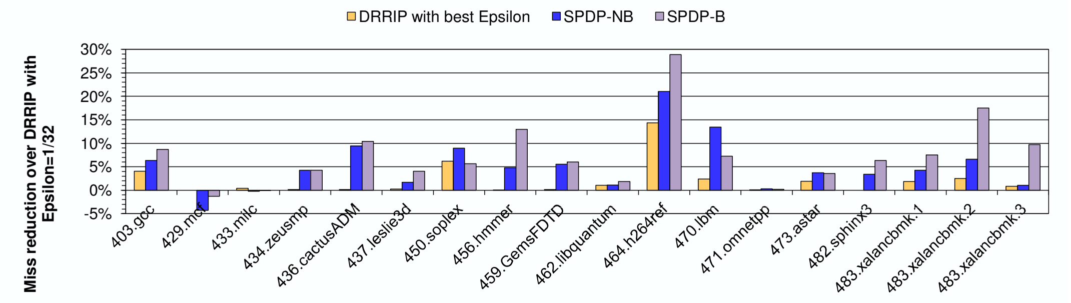


Figure 4. Comparing RRIP and static PDP.

RDDs of three execution windows for 483.xalancbmk are shown in (Fig. 5b).

436.cactusADM. 28% of the accesses in DRRIP are hits and the rest are misses, leading to replacement. Most of the missed lines are evicted early, before 16 accesses, with only 3% evicted after that. However, the total occupancy of the latter lines is as high as 16%, whereas for early evicted lines, it is only 8%. In fact, our results show that a number of lines occupy the cache for more than 256 accesses without reuse.

The occupancy of evicted lines is smaller for both PDP policies, 8% for SPDP-NB and 5% for SPDP-B, and much smaller for long eviction distance lines. Neither has lines with occupancy above 90 accesses. Also, the small difference between the occupancy of evicted lines in SPDP-NB and SPDP-B leads to the small difference between their performance – SPDP-B reduces 1% more misses than SPDP-NB compared to DRRIP (Fig. 4). The values of PD are 76 (SPDP-NB) and 72 (SPDP-B), respectively, which cover the highest peak of the RDD.

464.h264ref. Both static PDP policies result in a smaller occupancy fraction for lines evicted after more than 16 accesses. SPDP-B is better than SPDP-NB due to a high bypass rate (89% of the misses), hence protecting more lines and leading to higher miss reduction. This shows that the bypass plays an important role in this and several other benchmarks.

483.xalancbmk. Fig. 4 shows that the three windows of execution within the same benchmark have different performance for both static PDP policies. The best PD for each window are also different in SPDP-B – 100, 88 and 124, respectively (see Appendix A). This difference is due to different RDDs of these windows, as seen in Fig. 5b. The peaks are at different reuse distances and even the shapes of RDDs are not quite the same. This implies that a periodic dynamic computation of the PD would perform even better.

In summary, replacement based on protecting distance reduces cache pollution. A combination of replacement and bypass further improves reuse and reduces pollution. The rest of the paper thus targets non-inclusive caches with bypass.

2.4 The Hit Rate Model for a Non-Inclusive Cache

The definition of the protecting distance PD is not very helpful in finding the optimal PD, static or dynamic. This section develops a quantitative model relating the RDD, the PD and the cache hit rate for a non-inclusive cache. The RDD or its approximation can be built dynamically and the model then used to find the PD maximizing the hit rate. The model takes into account both replacement and bypass policies.

The following notation is used in this section: d_p is a value of the PD, N_i is the hit count for reuse distance i, and N_t is the total number of cache accesses. W is the set associativity. A line with an RD larger than d_p is called a *long line* and N_L is the number of *long lines*.

A function $E(d_p)$ approximating the hit rate is derived as following. Given a set of counters $\{N_i\}$ representing the RDD and N_t, $N_L = N_t - \sum_{i=1}^{d_p} N_i$.

Let us use the concept of occupancy defined in Sec. 2.3. A line with an RD of i, $i \le d_p$, has an occupancy of i and the line is hit after i accesses. The total occupancy L_i of all lines with RD of i in the cache is $L_i = N_i * i$.

The total occupancy of the long lines is $L_L = N_L * (d_p + d_e)$. The additional term d_e accounts for the fact that a line may not be immediately evicted after becoming unprotected. This may happen due to the presence of an invalid or another unprotected line during victim selection or when an access to a set is a hit on another line. The total occupancy of all lines is therefore $L_T = \sum_{i=1}^{d_p} L_i + L_L$.

The number of hits contributed by all lines is $Hits(d_p) = \sum_{i=1}^{d_p} N_i + H_L$, where H_L is the number of hits contributed by long lines. For $d_p \ge W$ the number of hits from long lines is quite small compared to that from protected lines. And the computed PD is greater than W (see below). Therefore, the following approximation is used $Hits(d_p) \approx \sum_{i=1}^{d_p} N_i$.

One access to a set with W lines increases the occupancy of each line in the set by 1 unit, thus W units for the whole set. The total number of accesses, therefore, is $Accesses(d_p) = L_T/W$ and the hit rate is $HR(d_p) = Hits(d_p)/Accesses(d_p)$.

To eliminate the dependence on cache organization (W) let us define $E(d_p) = HR(d_p)/W$. Here $E(d_p)$ is proportional to the hit rate and will be used to compute the protecting distance PD that maximizes the hit rate.

Substituting the expressions for L_i and L_L in the equation for $E(d_p)$ results in

$$E(d_p) \approx \frac{\sum_{i=1}^{d_p} N_i}{\sum_{i=1}^{d_p} (N_i * i) + \left(N_t - \sum_{i=1}^{d_p} N_i \right) * (d_p + d_e)} \quad (1)$$

Finally, the d_e is a function of cache associativity and program behavior. It has an impact on $E(d_p)$ only when the d_p is small. It

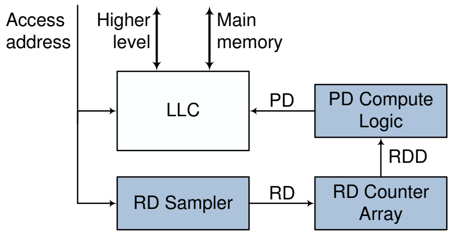


Figure 7. The PDP cache organization.

has been experimentally determined that it can be set to a constant equal to W. This is another reason why $E(d_p)$ is an approximation of the hit rate.

Fig. 6 shows the $E(d_p)$ and the actual hit rate as a function of static $d_p < 256$ for four SPEC CPU2006 benchmarks (Appendix A shows the complete suite). The RDD is also shown. The model can be seen to approximate the actual hit rate well, especially around the PD which maximizes hit rate.

Eq. 1 (with $d_e = W$) is used to find the optimal PD, i.e. the PD achieving the highest hit rate. The search algorithm computes $E(d_p)$ for all $d_p < 256$ and finds the maximum $E()$. Not that this search is only performed periodically and that $E(d_p + 1)$ can be computed from $E(d_p)$ to significantly reduce the search time.

3 PDP CACHE ORGANIZATION

This section describes the cache operation and the additional hardware required for dynamic computation of the optimal PD. Fig. 7 shows a block diagram of the PDP cache. The additional blocks include an RD sampler to measure the reuse distances (RDs), an array of RD counters which track the number of hits at each reuse distance (i.e. the RDD), and logic to find the PD using Eq. 1.

The RD sampler. The RD sampler monitors access to a small number of cache sets to collect the reuse distances observed. It uses a per-set FIFO of addresses which accessed the set. A new access to a sampled set has its address compared the set's FIFO entries. The FIFO position of the most recent match, if any, is the reuse distance RD. Its corresponding reuse counter in the counter array is incremented. An RD as large as d_{max} needs to be detected.

It has been shown in [24, 14] that sampling just 1/64 of the total sets is sufficient to capture the behavior of an application. A reduced FIFO insertion rate may be used to allow smaller FIFO size (note that cache tag check is still performed for each access). In this case a new entry is only inserted on every M^{th} access to the set, with a *sampling counter* counting up to M. The reuse distance RD is now computed as $RD = n \times M + t$, where n is the FIFO position of the hit, t is the value of of the sampling counter on the sampler hit. An entry is marked invalid on such a hit to reduce error in RD measurement. Similar to the work in [18], a FIFO entry uses 16 bits to store a tag. The total number of bits per sampled set is $\frac{d_{max}}{M} * 16 + \log_2 M$.

The array of RD counters. The RD counter array stores the RDD $\{N_i\}$. The i^{th} counter is the number of accesses with the RD of i. An RD arrives from the RD sampler and the corresponding counter is incremented. An additional counter is used for the total number of accesses N_t. These are saturating counters and, when a counter saturates, all other counters are frozen to maintain the RDD shape.

A space optimization for the array is to store hit counts for a consecutive range of RDs in one counter. The range extent is called a *step counter*, (S_c). For instance, if $S_c = 4$ then the first counter is incremented for RDs from 1 to 4, the next one for the RDs from 5 to 8, and so on. The number of counters can thus be reduced to $\frac{d_{max}}{S_c}$. This also reduces the computation time to select the protecting distance PD. However, this also means that the PD belongs to a range rather than being a single value. The proposed PDP implementation uses 16-bit N_i counters and a 32-bit N_t counter. Total number of bits required for the counter array is $\frac{d_{max}}{S_c} \times 16 + 32$.

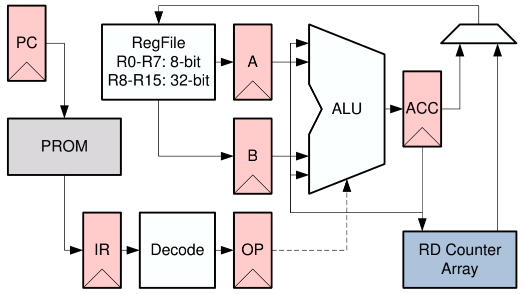


Figure 8. A "PD compute logic" special-purpose processor.

Logic to compute the PD. The PD is recomputed infrequently and thus its computation is not time critical. The logic to find the optimal PD can thus be implement it as a special-purpose "processor". A possible 4-stage pipelined architecture is shown in Fig. 8. It uses a 32-bit ALU, eight 32-bit registers and eight 8-bit wide. The processor uses the RD counter array as input and outputs the optimal PD. It executes one algorithm using sixteen integer instructions: add/sub, logical, move, branch, mult8 and div32. The mult8 multiplies a 32-bit register by an 8-bit register and is performed using shift-add. The div32 is a division of two 32-bit numbers and is performed as shift subtract/add non-restoring division (33 cycles). The processor takes 64 cycles to compute $E(d_p)$ for one d_p in Eq. 1. Thus the total time to compute the optimal PD, 64×256 cycles, is negligible compared to the number of cycles between two PD computations (512K accesses to the LLC in this paper). The processor can be put in low-power sleep mode when not in use. The processor was synthesized using a 65nm technology library and required 10K NAND gates operating at 500MHz clock frequency.

Cache tag overhead. The computed PD is used by the cache to set the initial value of remaining PD, RPD (see Sec. 2.2 for the definition), for inserted or promoted lines. Storing the RPD that can range up to $d_{max} = 256$ requires $n_c = 8$ bits per line. This overhead can be reduced by using an increment, the *Distance Step* S_d. An update of RPDs is done once every S_d accesses to the set using a per-set counter (and counting bypasses). An access causing the counter to reach $S_d - 1$ triggers the decrement of all RPDs in the set. This does not have to be done in parallel but can be done sequentially to reduce complexity. The maximum value of S_d is $\frac{d_{max}}{2^{n_c}}$.

The PDP parameters. A number of hardware parameters are used in the PDP: (1) the RD sampler size, (2) the S_c counter used in the counter array, (3) the number of bits to store the PD n_c, and (4) the maximum distance d_{max} (Sec. 1). The choice of these parameter values impacts both the hardware overhead and the cache performance. Sec. 6.1 presents a design space exploration of these parameters.

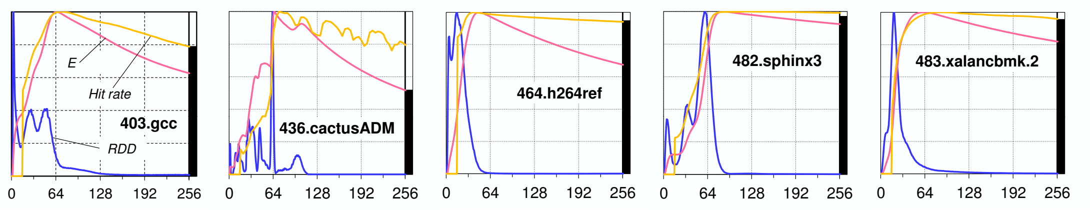


Figure 6. $E(d_p)$ **vs the actual hit rate.**

4 A PD-BASED CACHE PARTITIONING POLICY

All high-performance multi-core processors today use a shared LLC. The shared LLC can benefit significantly from (cooperative) cache partitioning. This section defines a new multi-core partitioning policy based on the concept of protecting distance. The key insight is that decreasing a thread's PD leads to a faster replacement of its lines and thus shrinks its partition. Increasing the PD has the opposite effect. The proposed policy defines a per-thread protecting distance and computes a set of PDs that partitions the cache to maximize hit rate. The computation is based on a new shared LLC hit rate model that is a function of a set of PDs.

The numerator and denominator in Eq. (1) are $H(d_p)$, the number of hits, and $A(d_p)/W$, the number of accesses. They need to be computed per thread when multiple threads access the shared cache. A thread t has $H_t(d_p^{(t)})$ hits and $A_t(d_p^{(t)})/W$ accesses for a protecting distance of $d_p^{(t)}$. The total number of hits and accesses for T threads are $Hits(d_p^{(t)}) = \sum_{t=0}^{T-1} H_t\left(d_p^{(t)}\right)$ and $Accesses(d_p^{(t)}) = \sum_{t=0}^{T-1} A_t\left(d_p^{(t)}\right)/W$, respectively. The multi-core function E_m is $E_m = Hits/(Accesses * W)$. Using a vector $\overrightarrow{d_p} = \left[d_p^{(0)}, ..., d_p^{(T-1)}\right]$ to denote an ordered set of $d_p^{(t)}$, the multi-core hit rate approximation as a function of $\overrightarrow{d_p}$ is:

$$E_m\left(\overrightarrow{d_p}\right) = \frac{\sum_{t=0}^{T-1} H_t\left(d_p^{(t)}\right)}{\sum_{t=0}^{T-1} A_t\left(d_p^{(t)}\right)} \qquad (2)$$

A vector $\overrightarrow{PD} = \left[PD^{(0)}, ..., PD^{(T-1)}\right]$ that maximizes $E_m(\overrightarrow{PD})$ defines the protecting distances $PD^{(t)}$ of each thread.

A heuristic is used to find the vector $\overrightarrow{PD}$ instead of an an exhaustive search. It is based on three observations. First, a thread with a high single-core E will also make a high contribution to the multi-core E_m. Second, a computed multi-core PD for a given thread is likely to be near one of the "peaks" in its single-core E. And third, the number of "important" peaks in an application is quite small. The heuristic thus builds $\overrightarrow{PD}$ by adding a thread at a time and searches for the added thread's PD near one of its peaks only.

The multi-core partitioning policy requires a counter array per thread but still uses the single-core PD computing logic to generate the "high peaks" for each thread . The heuristic first sorts the threads by their E's, then adds the thread with the "highest E" to the vector. The next highest thread is processed next, each of its peaks (computed using the single-core E) is considered in combination with peaks of the thread already in the vector. A search algorithm is used to find the peak of the combination that maximizes E_m. The process is repeated for each remaining thread.

It has been experimentally determined that considering just three peaks per thread is sufficient. For each thread combination, the search algorithm has the complexity of $O\left(T \times S\right)$, where S is the number of single-thread E re-computations. Given that the number of combinations is a linear function of T, the complexity of E_m is $O\left(T^2 \times S\right)$. The complexity is similar to that of UCP [25]. The processor described in the previous section can be easily extended to execute this heuristic. The latency of the PD vector search is still negligible compared to the PD recomputation interval.

Pipeline Depth	8
Processor Width	4
Instruction Window	128
DCache	32KB, 8-way, 64B, 2 cycles
ICache	32KB, 4-way, 64B, 2 cycles
L2Cache	256KB, 8-way, 64B, 10 cycles
L3Cache (LLC)	2MB, 16-way, 64B, 30 cycles
Memory latency	200 cycles

Table 1. The single-core processor.

5 EXPERIMENTAL METHODOLOGY

The proposed single- and multi-core management policies at the LLC were evaluated. A memory hierarchy with three levels of cache is modeled which is similar to the Intel Nehalem processor. An out-of-order core is modeled with the CMP$im [11] simulator. The core and the memory hierarchy parameters are described in Table 1. The L1 and L2 caches use the LRU policy. A multi-core processor with a shared LLC is also modeled with CMP$im. The shared LLC size is the single-core LLC size times p, where p is the number of cores.

The following SPEC CPU2006 benchmarks were used for the single-core evaluation: 403.gcc, 429.mcf, 433.milc, 434.zeusmp, 436.cactusADM, 437.leslie3d, 450.soplex, 456.hmmer, 459.GemsFDTD, 462.libquantum, 464.h264ref, 470.lbm, 471.omnetpp, 473.astar, 482.sphinx3 and 483.xalancbmk. Other benchmarks in the suite were not considered because they do not stress the LLC, e.g. their MPKI is less than 1 for the baseline DIP policy. A window of 1 billion consecutive instructions per individual benchmark was simulated. To see how each policy reacts to phase change within an application, three different 1B instruction windows were studied for 483.xalancbmk to observe the phase changes, with results denoted as 483.xalancbmk.X, where X is the window number. A subset of benchmarks which demonstrate significant phase changes were simulated with a window of 10 billion instructions.

Multi-core, shared LLC simulations used workloads generated by combining the individual benchmarks described above. Benchmark duplication was allowed in a workload. 80 workloads were generated using random selection for the 4-core and 16-core configurations. A workload completes execution when each of its threads completes the execution of its 1B instructions. A thread completing before other threads "rewinds" and continues its execution from the first instruction of its window. Per-thread statistics are collected when a thread finishes its first one billion instructions.

Single-core performance metrics are misses per thousand instructions (MPKI) and the IPC. Multi-core performance metrics are the *weighted IPC* ($W = \sum \frac{IPC_i}{IPC_{Singlei}}$), the *throughput* ($T = \sum IPC_i$), and the *harmonic mean of normalized IPC* ($H = N/\sum \frac{IPC_{Singlei}}{IPC_i}$). IPC_i above is the IPC of a thread i in multi-core, multi-programmed execution for a given replacement policy. $IPC_{Singlei}$ is the IPC of the thread i executed stand-alone on the multi-core and using the LRU policy. LRU is used as the baseline here for consistentency with prior work [25]. The results for each metric for a given replacement policy are are shown normalized to the shared cache DIP.

The single-core PDP is compared with DIP [24], DRRIP [14], a variant of EELRU [29] and the sampling dead block predictor (SDP) [18]. DIP and DRRIP used the dynamic policy. The $\epsilon = 1/32$ was used for BIP and BRRIP except when evaluating the impact of ϵ. An SDM with 32 sets and a 10-bit PSEL counter was used for DIP and DRRIP. Writebacks were excluded in updating PSEL counters in these policies. The hardware overhead for SDP was made 3 times as large as that reported in the original work in order to maximize the performance.

Each cache set for EELRU evaluation was augmented with a recency queue to measure the number of hits at a stack position. Two global counter arrays were used to count the number early hits and total hits for each pair of early eviction point *e* and late eviction point *l* over all sets. The parameters *e* and *l* were chosen aggressively to make sure that EELRU achieves its best possible performance. The maximum value of *l* is set to $d_{max} = 256$ to be compatible with PDP.

Three thread-aware shared cache partitioning policies were compared with PDP: the UCP [25], the PIPP [36] and the TA-DRRIP [14]. The lookahead algorithm was used to compute the ways for UCP and PIPP. Thirty two sampling sets were used for UCP and PIPP. $p_{prom} = \frac{3}{4}$, $p_{stream} = \frac{1}{128}$, $\theta_m \geq 4095$, and $\theta_{mr} = \frac{1}{8}$ were used for PIPP, per original work. The implementations of DIP, RRIP and TA-DRRIP were verified against the source code from the author's website.

6 EVALUATION AND ANALYSIS

The performance of PDP depends on a number of parameters. Thus we start with a parameter space design exploration for the single-core PDP. The parameters can be chosen to balance overhead and performance. Once most of the parameters are selected, this section presents and compares results for the single-core replacement and bypass policies, as well as the PDP shared cache partitioning policy.

6.1 A PDP Parameter Space Exploration

The performance and hardware overhead of the PDP are a function of the maximum PD allowed, sampler and FIFO sizes, and the number of bits to store the remaining PD per line. Fig. 9 shows the effect of two PDP parameters: the RD sampler size and the counter step S_c. The *Full* configuration uses a FIFO per LLC line. The *Real* configuration uses a 32-entry RD sampler (32 FIFOs, each with 32 entries). The impact of the counter step S_c is shown for the *Real* configuration. The specific PDs of each configuration can be found in Appendix A.

Range of PD	16-32	33-64	65-128	129-256
# of benchmarks	4	5	4	3

Table 2. The PD distribution of SPEC CPU2006 benchmarks.

The results show that the RDDs obtained using the 32-entry RD sampler are basically identical to those obtained without sampling. An even smaller sampler size can be used, but it will take longer to warm up and to build a good-quality RDD. Therefore, the rest of the paper uses the *Real* configuration of the RD sampler.

Varying the counter step S_c from 1 to 8, the $S_c = 2$ has mostly no difference with the $S_c = 1$. Two benchmarks, 456.hmmer and 470.lbm, show a noticeable change for higher values of S_c. This is due to the rounding errors in the PD computation. The $S_c = 4$ is selected for the rest of the paper in a trade-off between performance and hardware overhead,

A third parameter, the maximum protecting distance d_{max}, is chosen based on results in Table 2. The table shows the distribution of optimal PD for the sixteen SPEC CPU2006 benchmarks used and the *Full* sampler configuration. None of the benchmarks has a PD larger than 256, even if d_{max} is set to be larger. Therefore the $d_{max} = 256$ is used in the rest of the paper. The table also shows that a smaller d_{max} can also be used, but with some impact on performance. For example, the $d_{max} = 128$ results in lower performance for three benchmarks.

A fourth parameter, the number of bits per cache tag, is evaluated in the next section.

6.2 The Single-core PDP

Fig. 10 shows the performance of PDP and three prior policies: DRRIP [14], EELRU and SDP [18]. The static SPDP-B is also shown. All the results are normalized to the DIP [24] pollcy. The evaluated metrics are the miss reduction, IPC improvement, and the fraction of accesses bypassed. Three different PDP configurations are shown varying the number of extra bits per cache tag n_c (e.g, PDP-3 has $n_c = 3$).

First, let us compare prior replacement policies. DRRIP has several benchmarks which improve over DIP, significantly in the 450.soplex, 456.hmmer, and 483.xalancbmk.3. The benchmark 464.h264ref shows degradation over DIP. This benchmark was analyzed in Sec. 2.1. In fact, DRRIP achieves similar performance to DIP with $\epsilon = 1/4$ for 464.h264ref. On average, DRRIP reduces the misses by 1.8%, leading to a 1.5% improvement in IPC over DIP. This is consistent with results for the SPEC CPU2006 suite reported in the DRRIP work [14]. The EELRU is shown to have a significant degradation compared to DIP in several benchmarks. This is due to the fact that a majority of cache lines are evicted before reaching their reuse point, hence they pollute the cache without contributing to hits. In fact, it was previously reported that DIP outperforms EELRU [24].

Second, let us compare the miss reduction for the static PDP (SPDP-B) and the dynamic PDP (PDP-8). Recall that the dynamic PDP uses the hit rate model to find the PD which maximizes the

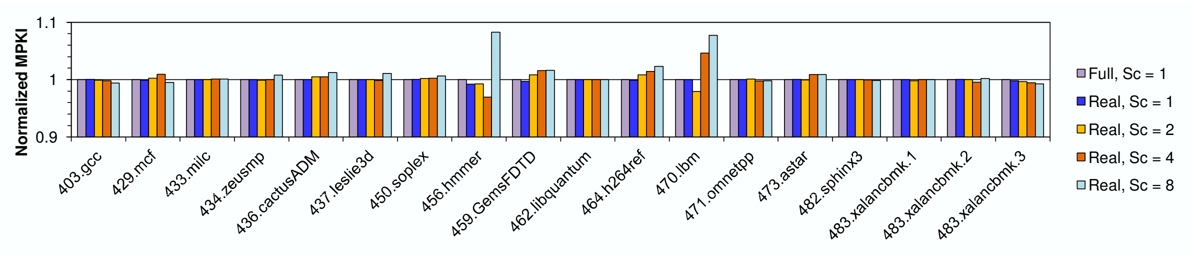


Figure 9. Comparing PDP parameters.

hit rate. The upper bound for short traces (1B instructions) is the static PDP and the results show that in most cases the dynamic PDP is close to this bound. However, there is a significant difference in the case of 429.mcf and 456.hmmer. Appendix A also shows that these two benchmarks have the best static PD which is quite different from the computed PD. On average, SPDP-B eliminates 1.4% more misses than PDP-8.

Third, consider the impact of n_c. PDP-8 has a 1.5% and 0.5% higher miss reduction, respectively, compared to PDP-2 and PDP-3. This results in a 2.0% and 0.5% higher improvement in IPC, respectively. But PDP-2 and PDP-3 have significantly less overhead. Thus the n_c parameter can be used to reduce the overhead with a small performance impact. But for 436.cactusADM PDP-2 leads to a significant performance reduction due to the approximation errors eviction of many lines before reuse. Sometimes though, benchmarks get a improvement with smaller n_c, such as 429.mcf and 473.astar, although the impact is less significant for the latter. 462.libquantum has a big performance loss for $n_c < 8$, because its PD is 256, equal to the chosen d_{max}, and all lines are evicted before this PD. A larger d_{max} will be able to avoid this loss.

Note that the three different execution windows of 483.xalancbmk have different performance. The second window sees a miss reduction of up to 21%, with a 21% improvement in IPC (over DIP) which is higher than in the other two windows. The results from only one window, with medium improvement, are used in computing all the averages, for a fair comparison among policies.

Overall, PDP significantly improves performance over DIP and DRRIP in several benchmarks. 436.cactusADM, 450.soplex, 482.sphinx3 and 483.xalancbmk have more than a 10% improvement over DIP. Other benchmarks do not have a significant improvement. In some the LRU replacement works fine (LRU-friendly), such as 473.astar. Others are streaming benchmarks with very large data sets, such as 433.milc, 459.GemsFDTD, 470.lbm. The average IPC improvement over DIP for PDP-2 and PDP-3 is 2.9% and 4.2%, respectively, while the DRRIP improvement over DIP is 1.5%.

Next, let us compare PDP with SDP, a bypass policy using the program counter based dead block prediction. SDP is able to identify and bypass many “dead-on-arrival” lines, but it does not always have a good prediction. This happens for 464.h264ref and 483.xalancbmk, where SDP loses significantly to DIP. Note that for 483.xalancbmk SDP still improves over LRU, the baseline policy it builds upon. A look at the RDDs shows that these benchmarks do not have many lines whose RDs are large, the target of the SDP predictor. This explains the difference in performance for SDP and PDP. Benchmarks where SDP is significantly better than all other policies are 437.leslie3d and 459.GemsFDTD where the use of the PC-based prediction is shown to be advantageous. On average, SDP improves the IPC of 1.6% over DIP.

Bypassing a cache reduces its active power dissipation, an important issue for large LLCs, by not writing the data into the LLC. Fig. 10c shows the fraction of bypassed lines normalized to the number of LLC accesses. Benchmarks which have high miss rates also have high bypass rates. For nearly 40% of accesses the LLC does not have to be written.

The results in this section show that PDP-2 or PDP-3 proved a good balance of overhead and performance. The hardware overhead of the RD sampler, the array of RD counters, the processor to compute the PD and per-line bits was estimated for a 2MB LLC using the PDP-2 and PDP-3 policies. Expressed in terms of SRAM bits, the overhead is 0.6% for PDP-2 and 0.8% for PDP-3 of the total LLC size. The overheads for DRRIP and DIP are 0.4% and 0.8%, respectively.

6.3 A Direction to Improve PDP

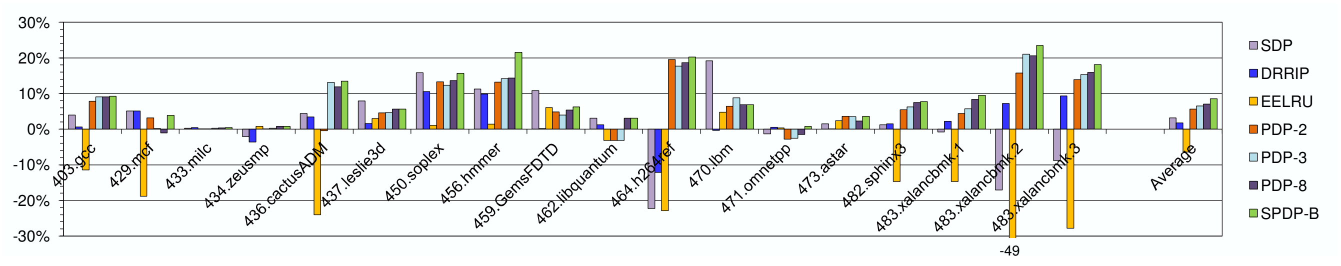
The analysis above gives some hints to further ways to improve PDP. For example, 429.mcf has a higher performance with smaller n_c while SDP is better than other policies in 437.leslie3d and 459.GemsFDTD. The common cause in all cases is the inability to determine how long a line should be protected. For 437.leslie3d and 459.GemsFDTD this can be solved by using a PC-based predictor. A variant of PDP was used to better understand the case of 429.mcf. The variant inserts missed lines with the PD = 1 (mostly unprotected) instead of the computed PD and this result in an 8% miss reduction over DIP. The reduction for SPDP-B is 3.9% and 5.1% for DRRIP. This means that when the inserted lines are removed faster a higher miss reduction is achieved for this benchmark.

The above suggests that the PDP can be improved by grouping lines into different classes, each with its own PD, and where most of the lines are reused. The lines in a class are protected until its PD only, thus they are not overprotected if they are not reused. In fact, prior approaches have classified and treated cache lines differently. A popular way is using the program counters [18, 22, 9]. Another way is to use a counter matrix to group lines [19]. SHiP [34] proposes to improve RRIP using different ways to group cache lines. However, the hardware overhead needs to be carefully considered.

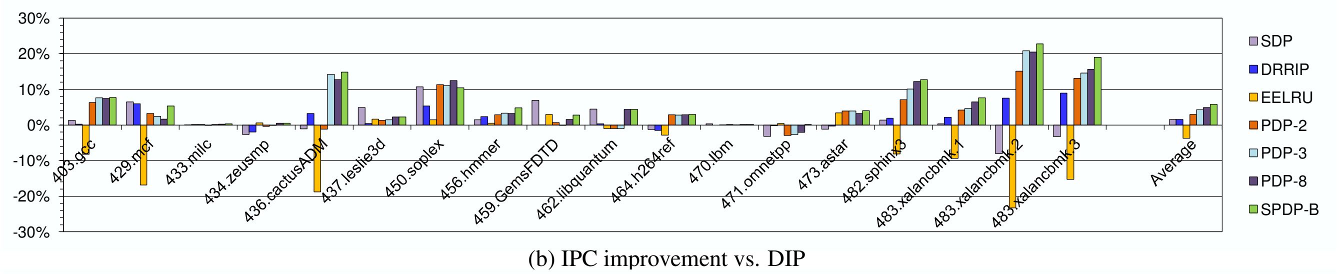
6.4 Adaptation to Program Phases

An application phase change may require the PD to be recomputed. For instance, the three execution windows of 483.xalancbmk have different PDs and different behaviors. to detect phase change The PD needs to be recomputed and the RD counter array reset frequently enough to detect the change. The computed PD is used until the next recompilation while the counters collect the new RDD.

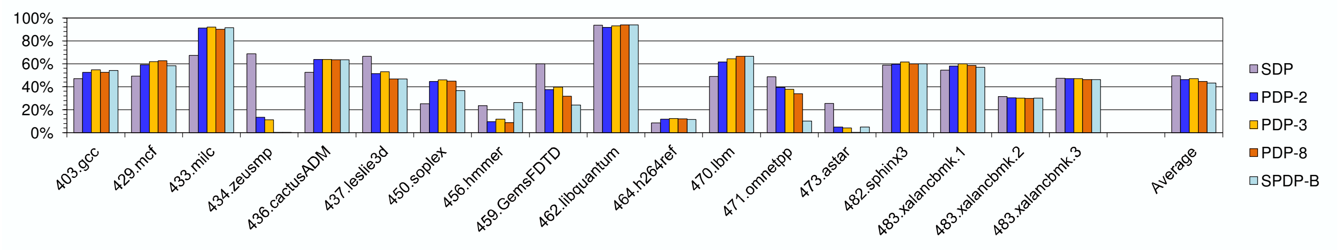
Five SPEC CPU2006 benchmarks have phase changes in an ex-



(a) Miss reduction vs. DIP



(b) IPC improvement vs. DIP



(c) Cache bypass as a fraction of accesses

Figure 10. Performance of replacement and bypass policies (Results for 483.xalancbmk.1 and 483.xalancbmk.2 are excluded from averages)**.**

ecution window of 10B instructions. Fig. 11c shows the PD change over time. Fig. 11a shows the effect of the PD recomputing interval between 1M and 8M accesses, which can be significant. Fig. 11b compares different replacement policies for these benchmarks. PDP is able to adapt to phase changes for these benchmarks.

6.5 The Case for Prefetch-Aware PDP

The PDP policy can be applied in the presence of prefetching. The RDDs in this case are dependent of a specific prefetching mechanism. Note that prefetching techniques often target very long distance access streams, i.e. lines which have very large RDs. Two prefetch-aware variants of the PDP were investigated: (1) prefetched lines are inserted with the PD of 1, and (2) prefetched lines bypass the LLC.

The initial evaluation using a simple stream prefetcher and the modified PDP-8 showed the following. First, the prefetch-unaware PDP had a 3.1% improvement over the prefetch-unaware DRRIP, which is similar to the results without prefetching. The two PDP variants further improve the IPC to 4.1% and 5.6% over prefetch-unaware PDP. Many benchmarks show an improvement of over 20%, including 403.gcc, 450.soplex, 482.sphinx3 and 483.xalancbmk. The improvement is due to the fact that PDP removes prefetched lines fast and even bypasses them, hence they do not pollute the cache. This shows that the PDP can be easily modified to work with prefetching.

6.6 The Cache Partitioning Policy

Fig. 12 shows the performance of the multi-core PD-based partitioning together with other thread-aware policies for 4- and 16-core workloads. Three metrics are reported: the weighted IPC (W), the throughput (T), and the harmonic mean of normalized IPC (H). The PD-based partitioning parameters are the same as for the single-core PDP, except that $S_c = 16$.

The average W, T, and H on four cores are slightly higher for both PDP-2 and PDP-3 compared to the TA-DRRIP and are higher than the UCP and PIPP. The PD-based partitioning is significantly better than other policies for approximately 30 individual workloads, for another 20 it has a negligible improvement, and is not as good as the TA-DRRIP for the rest.

The PD-based partitioning is shown to be more scalable with a larger improvement on the 16-core workloads. Its performance is better than that of all other partitioning policies for more than 70 of the 80 workloads. UCP and PIPP do not scale as well compared to the other policies. On average, PDP-3 has a 5.2%, 6.4% and 9.9% improvement in W, T, H, respectively, over the TA-DRRIP.

7 RELATED WORK

Cache line protection. Let us discuss prior policies from the perspective of line protection, even if they do not use this concept explicitly. LRU can only guarantee that a line is protected for the number of unique accesses equal or smaller than associativity. A line

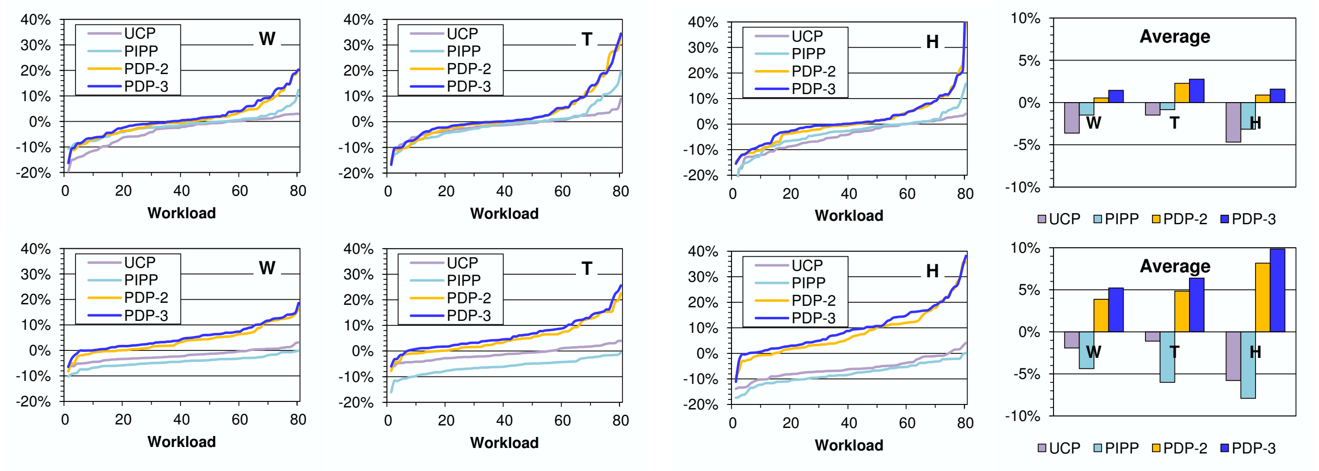
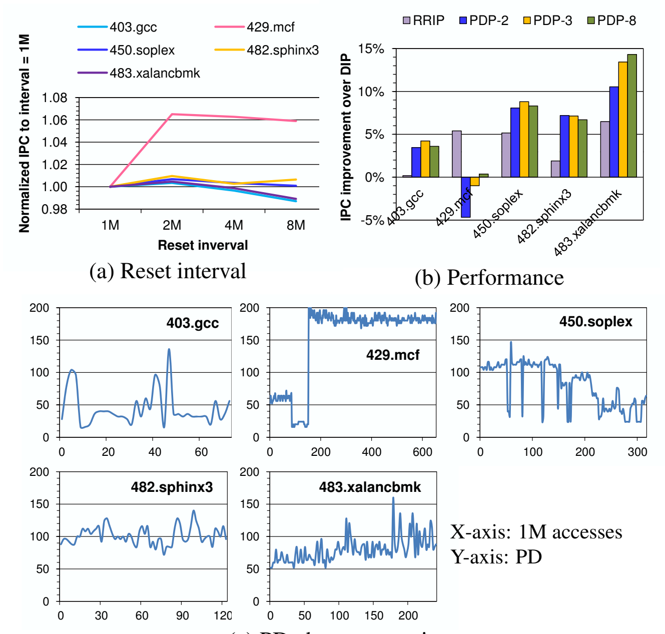


Figure 12. Cache partitioning policies for 4-core (top) and 16-core (bottom) workloads (normalized to TA-DRRIP).



(c) PD change over time

Figure 11. Adaptation to program phases.

with a reuse distance RD larger than associativity can be protected either by replacing other lines or by explicitly protecting the line. DIP [24] and RRIP [14] predict the re-reference future of lines and protect lines which are predicted to be reused sooner. The counter-based replacement policy [19], using a matrix of counters, protects lines by not evicting them until they expire. This approach does not use an explicit protecting distance, but rather predicts how long a line should be protected by using the past behavior of lines in the same class. The work in [15] measures Inter Reference Recency (IRR) which is similar to reuse distance. Based on this IRR information, cache lines are classified into 2 categories: high IRR and low IRR, and low IRR lines are protected over high IRR lines. The work in [21] uses PC information to identify which lines to protect and the cache is also divided into two regions: MainWays and DeliWays. The use of DeliWays is to protect cache lines longer. Dead block prediction [18] uses program counters (PC) to predict which lines have a very long RD and to bypass them, thus protecting existing lines in the cache.

The approach in [17] used a PC-based reuse distance predictor and monitored the distance remaining till reuse. An LRU line or a line with the largest "distance till reuse" is evicted, whichever is larger. This can be viewed as "protecting" lines with lower distance till reuse. The two issues with this policy are: 1) the PC-based reuse distance prediction is not very accurate and 2) the decision to switch eviction to an LRU line is an approximation without a model to support it.

Shepherd cache [26] emulates the Belady [2] policy to estimate RD and evict lines with a long RD and protect lines with a short RD. The insertion algorithms in [6], which target an exclusive LLC, insert a line with a certain age to determine how long the line should be protected. Similarly, SHiP [34] determines the insertion age of a line by using information about either the memory region or program counters or instruction sequence.

The EELRU [29], or early eviction LRU, was defined for page replacement using the model in [33] and accurately measured the reuse distance. It used LRU for part of the set (physical memory) and switched to early eviction when LRU failed. It can be said to divide the pages into protected and unprotected regions and to compute the region boundaries dynamically. The computation used hit counters and the search process for the best boundary is similar to the search for the best PD. However, unlike the PDP, the EELRU "protection" is managed via eviction only (by evicting all older lines first), i.e. all missed blocks are inserted into the set.

The PDP is different from prior work in that it explicitly estimates how long a line should be protected using the concept of protecting distance. After this explicit protection, if it is not reused, the line becomes unprotected and can be replaced.

Distance measurements. Three ways to measure the reuse distance were defined in prior work. The first one is the stack distance, used by LRU and its variants, with the time unit defined as the number of unique accesses [24, 29, 28]. Stack distance can be used to tell which lines are older, but not its real age. The other two are non-stack: one uses a cache miss as the time unit [21], and the other one uses an access as a unit [22, 17, 4]. The former is dependent

Benchmark	SPDP		Full	Real, $S_c =$			
	NB	B		1	2	4	8
403.gcc	68	72	63	64	64	64	64
429.mcf	136	152	182	182	180	184	176
433.milc	248	248	216	216	216	216	208
434.zeusmp	16	16	16	16	16	16	16
436.cactusADM	76	72	65	66	66	66	64
437.leslie3d	36	40	40	40	40	40	48
450.soplex	60	52	73	74	72	72	80
456.hmmer	44	176	22	22	24	24	32
459.GemsFDTD	24	24	26	26	28	32	32
462.libquantum	256	256	256	256	256	256	256
464.h264ref	40	40	48	48	48	48	48
470.lbm	52	52	50	50	52	56	64
471.omnetpp	20	20	33	32	32	32	32
473.astar	24	20	16	16	16	16	16
482.sphinx3	84	120	84	84	84	88	80
483.xalancbmk.1	88	100	113	114	116	112	112
483.xalancbmk.2	72	88	56	56	56	56	64
483.xalancbmk.3	96	124	74	74	72	72	80

Table 3. The PDs of SPEC CPU2006 benchmarks.

on the replacement policy. Computing stack distance is more complex. Thus we used access-based distance measurement in this paper. Note that the approach in [17] uses global RDs, while PDP uses set-based RDs, which significantly reduces overhead.

Cache bypass. There has been a number of studies using bypass [6, 18, 20, 16, 32, 19, 5]. The dead block prediction work [18] bypasses blocks predicted dead. The work in [32] explores bypass by characterizing data cache behavior for individual load instructions. The approach in [20] uses a hash table to identify never-accessed lines. The approach in [16] uses memory addresses to identify which lines to bypass. Recently, the work in [6] proposed bypass algorithms that use sampling sets to identify dead and live blocks. Our approach bypasses a block when existing blocks are still protected, but without explicitly identifying which lines are likely to be dead. The bypass mechanism in [19] also bypasses lines if there are no expired lines in the set.

Cache sampling. Prior work used sampling on a subset of cache sets [24, 14, 18] and on a fraction of the accesses [18, 22]. Other approaches use feedback to adjust the cache replacement [3] or partitioning [30]. PDP's only contribution here is a low hardware overhead.

RDDs and hit rate model. The IGDR policy [31] used the Inter-Reference Gap Distribution which is similar to the RDD. It used the concept of Inter-Reference Gap from [23]. IGDR used the distribution to compute a "weight" of cache lines and evict the line with a smallest weight. The work in [19] used the distribution implicitly. The approach in [22] used PCs to compute the RD of a line. PDP uses the RDD to explicitly compute a global protecting distance. The PD computation searches for a maximum of E(PD, RDD) and uses the PD achieving the maximum. EELRU [29] used a similar search approach to choose the best set {eviction point, probability}.

Prefetch-aware caching policies. Prior work investigated replacement policy in the presence of prefetching [7, 35, 6]. PAC-Man [35] is one such approach showing that RRIP can be modified to adapt to prefetching. Our preliminary results also show opportunities to improve PDP in the presence of prefetching.

Shared-cache policies. A number of multi-core, shared cache partitioning policies [25, 36] have been proposed as well as thread aware insertion or eviction [12, 14]. Recently, Vantage mechanism [27] used an analytical model to enhance partitioning in CMP and this model can be used with existing policies. Similar to the work in [12, 14], multi-core PDP does not perform partitioning among cores explicitly, rather it estimates the needs of each thread from its PD and chooses a set of PDs, one per thread, that maximize the hit rate.

8 CONCLUSIONS

Cache management policies such as replacement, bypass, and shared cache partitioning, have been exploiting – directly or indirectly – the data reuse behavior to improve cache performance. This paper proposed the novel PDP cache management policy which explicitly protects a cache line for a predicted reuse distance, the protecting distance PD. This guarantees a hit if the reuse occurs while the line is protected. Unprotecting a line speeds up its eviction and reduces cache pollution, thus freeing the cache space for new lines. The PDP achieves a good balance between reusing lines and reducing pollution. A bypass mechanism can be used when all the lines in a set are protected. A hit rate model as a function of program behavior and PD is proposed and shown to be accurate. The model is used to periodically recompute the PDP to adapt to phase changes and memory access behavior. A shared-cache hit rate model is also developed and used by the new PD-based partitioning policy. The additional hardware is shown to have the overhead similar to existing replacement policies. The performance evaluation of PDP shows that it outperforms existing management policies for both single-core and multi-core configurations.

A RESULTS FOR SPEC CPU2006 BENCHMARKS

Table 3 shows the PDs of the benchmarks used in this paper. The first two columns are for the static PDP (see Sec. 2) with and without bypass (B and NB). The third column shows the PDs of the dynamic PDP with a full RD sampler (see Sec. 6.1). The last four columns are the PDs with a real RD sampler and different values of S_c (see Sec. 6.1). These results were collected at the end of the 1B instruction execution window.

Fig. 13 shows the RD distribution for all benchmarks and windows, the modeled hit rates (E) and the static hit rates as described in Sec. 2. Each is normalized to its highest peak.

ACKNOWLEDGMENTS

This work is supported in part by NSF grant CISE-SHF 1118047 and by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625. Nam Duong is also supported by the Dean's Fellowship, Donald Bren School of Information and Computer Sciences, UC Irvine. The authors would like to thank the anonymous reviewers for their useful feedback and Drs. G. Loh and M. Qureshi for their shepherding help.

REFERENCES

[1] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez. Scavenger: A new last level cache architecture with global block priority. In *MICRO'07*.

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

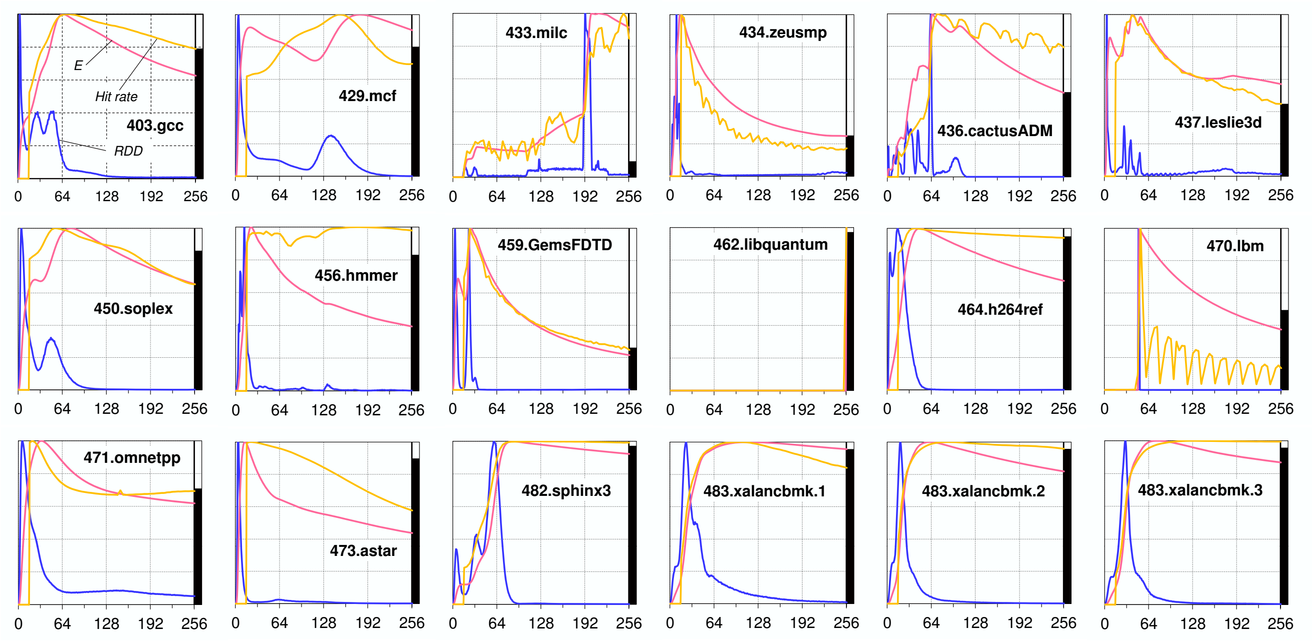


Figure 13. The RDDs, modeled and actual hit rates of SPEC CPU2006 benchmarks.

[3] N. Duong, R. Cammarota, D. Zhao, T. Kim, and A. Vendenbaum. SCORE: A score-based memory cache replacement policy. In *JWAC'10*.

[4] M. Feng, C. Tian, C. Lin, and R. Gupta. Dynamic access distance driven cache replacement. *ACM Transactions on Architecture and Code Optimization, 2011*.

[5] H. Gao and C. Wilkerson. A dueling segmented LRU replacement algorithm with adaptive bypassing. In *JWAC'10*.

[6] J. Gaur, M. Chaudhuri, and S. Subramoney. Bypass and insertion algorithms for exclusive last-level caches. In *ISCA'11*.

[7] B. S. Gill and D. S. Modha. SARC: sequential prefetching in adaptive replacement cache. In *ATEC'05*.

[8] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems. In *MICRO'09*.

[9] M. Hayenga, A. Nere, and M. Lipasti. MadCache: A PC-aware cache insertion policy. In *JWAC'10*.

[10] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr., and J. Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies. In *MICRO'10*.

[11] A. Jaleel, R. S. Cohn, C. keung Luk, and B. Jacob. CMP$im: A pin-based on-the-fly multi-core cache simulator. In *MoBS'08*.

[12] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *PACT'08*.

[13] A. Jaleel, H. H. Najaf-abadi, S. Subramaniam, S. C. Steely, and J. Emer. CRUISE: cache replacement and utility-aware scheduling. In *ASPLOS'12*.

[14] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *ISCA'10*.

[15] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *SIGMETRICS'02*.

[16] T. Johnson, D. Connors, M. Merten, and W.-M. Hwu. Run-time cache bypassing. *IEEE Transactions on Computers, 1999*.

[17] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *ICCD'07*.

[18] S. M. Khan, Y. Tian, and D. A. Jimenez. Sampling dead block prediction for last-level caches. In *MICRO'10*.

[19] M. Kharbutli and Y. Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers, 2008*.

[20] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *MICRO'08*.

[21] R. Manikantan, K. Rajan, and R. Govindarajan. NUCache: An efficient multicore cache organization based on next-use distance. In *HPCA'11*.

[22] P. Petoumenos, G. Keramidas, and S. Kaxiras. Instruction-based reuse-distance prediction for effective cache management. In *SAMOS'09*.

[23] V. Phalke and B. Gopinath. An inter-reference gap model for temporal locality in program behavior. In *SIGMETRICS'95*.

[24] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA'07*.

[25] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO'06*.

[26] K. Rajan and G. Ramaswamy. Emulating optimal replacement with a shepherd cache. In *MICRO'07*.

[27] D. Sanchez and C. Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *ISCA'11*.

[28] D. L. Schuff, M. Kulkarni, and V. S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *PACT'10*.

[29] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: simple and effective adaptive page replacement. In *SIGMETRICS'99*.

[30] S. Srikantaiah, M. Kandemir, and Q. Wang. SHARP control: controlled shared cache management in chip multiprocessors. In *MICRO'09*.

[31] M. Takagi and K. Hiraki. Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches. In *ICS'04*.

[32] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *MICRO'95*.

[33] C. Wood, E. B. Fernandez, and T. Lang. Minimization of demand paging for the LRU stack model of program behavior. *Information Processing Letters, 1983*.

[34] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. Steely, and J. Emer. SHiP: Signature-based hit predictor for high performance caching. In *MICRO'11*.

[35] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer. PACMan: prefetch-aware cache management for high performance caching. In *MICRO'11*.

[36] Y. Xie and G. H. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ISCA'09*.