Dynamic Access Distance Driven Cache Replacement

MIN FENG, CHEN TIAN, CHANGHUI LIN, and RAJIV GUPTA, University of California, Riverside

In this paper, we propose a new cache replacement policy that makes the replacement decision based on the reuse information of the cache lines and the requested data. We present the architectural support and evaluate the performance of our approach using SPEC benchmarks. We also develop two reuse information predictors: a profile-based static predictor and a runtime predictor. The applicability of each predictor is discussed in this paper. We further extend our reuse information predictors so that the cache can *adaptively* choose between the reuse information based replacement policy and an approximation of LRU policy. According to the experimental results, our adaptive reuse information based replacement policy performs either better than or close to the LRU policy. Our experiments show that L2 cache misses are reduced by 12.32% and 19.95% using the profiling-based static and runtime adaptive predictors respectively.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*cache memories*; B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids—*simulation*

General Terms: Design, Performance

Additional Key Words and Phrases: Cache Replacement Policy, L2 Cache, Value Prediction

ACM Reference Format:
Feng, M., Tian, C., Lin, C., and Gupta, R. 2011. Dynamic Access Distance Driven Cache Replacement ACM Trans. Architec. Code Optim. 9, 4, Article 39 (March 2010), 31 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Because of the huge speed gap between processor and memory, the cache performance is a key factor which affects the execution speed of applications. The LRU replacement policy and its variants (e.g., the pseudo-LRU policy) are currently the industry standard for cache replacement policy and have been widely used for many decades. The LRU policy was designed for workloads with high temporal locality. For the workloads that have a cyclic memory reference pattern and a working set larger than the cache size, the LRU policy gives poor performance. Therefore many techniques have been proposed to improve the performance of the LRU policy on the low-locality workloads [Qureshi et al. 2007; Etsion and Feitelson 2007; Wang et al. 2002; Wong and Baer 2000]. However, the performance of most techniques relies heavily on the data access patterns of the specific workloads. Thus, many of them do not adapt to different applications or even different phases of the same application. This paper aims to design a cache replacement policy that performs well for wide range of workloads.

In order to work well for different kinds of workloads, a cache replacement policy must be able to make the replacement decisions based on the data access patterns of those workloads. Since the reuse information reveals the data access patterns of the

Author's address: M. Feng, C. Tian, C. Lin and R. Gupta, Computer Science and Engineering Department, University of California, Riverside.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2010 ACM 1544-3566/2010/03-ART39 $10.00
DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

workloads, it is a powerful basis for guiding cache replacement. In this paper, we propose a reuse information based cache replacement policy that keeps the higher-locality data in the cache and bypasses the cache when the lower-locality data is referenced. The new cache replacement policy makes the replacement decision based on the reuse information of the cache lines and the requested data. To realize this policy, the cache is required to maintain the reuse information of each cache line and the reuse information of each data access needs to be predicted by the processor. We present the architectural modifications required to support our cache replacement policy and evaluate its performance.

Our cache replacement policy requires the reuse information of each data access. In this paper, we develop two reuse information predictors: a *static predictor* that predicts the reuse information based on the profiling runs and then passes the information on to the processor by encoding it in form of cache hints associated with memory instructions; and a *dynamic predictor* that makes the prediction for a memory instruction based on its reuse history. Each of the predictors has its own advantages. The static predictor requires minimal hardware support. Thus it has low hardware design cost and saves power at runtime. It is suitable for low-end computer systems, such as embedded systems. On the other hand, the dynamic predictor requires extra hardware logic and storage but does not require any compiler support. Therefore, commercial computer systems (e.g., personal computers) can benefit from the dynamic predictor without worrying about backward compatibility.

In this paper, we also present a method for reuse information prediction that allows our replacement policy to approximate the LRU replacement policy. Our experiments show that the difference between the LRU policy and our approximation of the LRU policy is insignificant. We then extend our static and dynamic predictors to enable them to adaptively choose between our access distance driven cache replacement policy and our approximate LRU policy. When the prediction accuracy for a program is low, the approximation of LRU can be used to avoid the cache performance degradation due to low prediction accuracy. This extension to the predictors does not require any extra hardware overhead. Our experiments show that L2 cache misses are reduced by 12.32% and 19.95% using profiling-based static and the runtime *adaptive* predictors respectively.

Specifically, we make the following contributions in this paper. First, we propose a reuse information based cache replacement policy and its architectural support. Second, we develop two reuse information predictors: a profiling-based static predictor and a runtime predictor. Finally, we extend our reuse information predictors so that the cache can adaptively choose between the reuse information based replacement policy and an approximation of LRU policy.

The remainder of the paper is organized as follows. In section 2 we first present the design of our cache replacement policy and then propose two predictors that provide the reuse information for the cache at runtime. In section 3, we evaluate the performance of our reuse information based cache. Section 4 discusses the related work and section 5 concludes this paper.

2. ACCESS DISTANCE BASED CACHES

Intuitively, a good cache replacement scheme needs to keep the data with high temporal locality in the cache while evicting the data that will not be accessed in the near future [Belady 1966; Mattson et al. 1970; Gu et al. 2008; Wang et al. 2002]. Reuse information is often used for analyzing the data access patterns of programs. Reuse information reveals the temporal locality of the data used by the programs. Therefore, it can be used to guide cache replacement for better cache performance.

In this paper, we define the *forward access distance* of a memory access as the number of memory reference between the current and the next reference to the same data block. Note that this is different from the definition of forward reuse distance that only counts distinct data blocks. Our *forward access distance* counts multiple accesses to the same data element. For example, in the access sequence "ld a, ld b, ld c, ld b, ld a" (where a, b, and c are mapped into the same cache set), the *forward access distance* for the first instruction is 3 (instead of 2 when only distinct elements are counted). We do not use the original reuse distance definition since counting distinct data blocks requires significant hardware overhead for recording data access history.

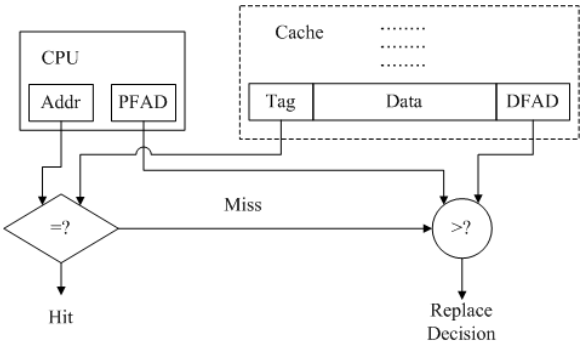

Fig. 1. Dynamic access distance based cache.

Fig. 1 shows the overview of our access distance based cache. In our scheme, when a memory instruction is issued, the processor also predicts the expected forward access distance for the data access. In addition, the expected forward access distance of each cache line is also maintained at runtime. If the requested data causes a cache miss, the replacement decision is based on the forward access distances of the data accesses and the related cache lines such that the temporal locality of the data in the cache is maximized. For brevity, we denote the Dynamic Forward Access Distance of a *cache line* as DFAD and the Predicted Forward Access Distance of a *data access* as PFAD.

In the above scheme, there are two main design issues. First, the cache replacement policy must be devised. Second, mechanisms must be designed for the prediction of the forward access distances. In the following sections, we present our solutions to these two problems in detail.

2.1. Cache Replacement Policy

2.1.1. Direct Mapped Cache. The objective of our cache replacement policy is that when a cache miss occurs, we keep the data block with higher temporal locality in the cache. If the requested data block is reused earlier than the data stored in the cache, we replace the block in the cache with the requested data block; otherwise, the requested data access bypasses the cache and the old data block is retained in the cache.

PFAD. A memory access instruction in a program may have various forward access distances along different execution paths. These forward access distances can be used to characterize the data access pattern of the memory instruction. Short forward access distances imply that the data block requested by the memory instruction exhibits good temporal locality. In our replacement policy, when a memory access instruction is executed, the CPU also generates its Predicted Forward Access Distance (PFAD) for making the replacement decision. In this section, we will assume that the processor

somehow knows the ideal PFAD of the executed memory instruction. However, in the next section we will present our techniques for PFAD generation.

DFAD. Similar to the forward access distance of the data block accessed by a memory access, the dynamic forward access distance (DFAD) of a cache line in the cache is defined as in how many memory accesses the cache line is going to be accessed again. A cache line's DFAD indicates its next usage time. Our replacement policy makes the replacement decision based on the PFAD of the executed memory instruction and the DFADs of the cache lines in the cache. For ease of presentation, we first describe our replacement policy for direct mapped caches and later show how it is extended for set associative caches.

Data replacement. When a memory access instruction is executed, the processor first looks for the requested data in the cache. In the direct mapped cache structure, the requested address is uniquely mapped to a cache line. If the cache line does not contain the requested data, a cache miss occurs and thus the data needs to be accessed from the main memory. If the PFAD of the executed memory instruction is smaller than the DFAD of the corresponding cache line, it means that the requested data will be accessed first after this access. Therefore, we replace the cache line with the requested data. On the other hand, if the DFAD of the cache line is smaller that the PFAD of the data access, it means that the data in the cache line will be accessed first after this access. In this case the requested data is sent directly to the processor thus bypassing the cache.

Please note that the DFAD of the relevant cache line may be zero when a cache miss takes place. In this special case, the cache line was predicted to be used by this data access but is actually not accessed, i.e. the prediction was wrong. Since we do not know when the cache line will be accessed again and it is impossible to keep it in the cache forever, we replace the cache line with the requested data block simply according to the LRU policy.

DFAD maintenance. The DFAD of each cache line in the cache needs to be maintained at runtime. Whenever a cache line is hit or replaced, we set its DFAD value to the PFAD value of the accessing instruction. Moreover, after every data bypass, the relevant cache line's DFAD must be decremented since its next usage is one step closer. Fig. 2 summarizes our cache replacement scheme. The DFAD of each cache line is implemented using a saturating up-down counter.

```
CacheAccess(instr, line)
Input:
   The accessing instruction instr,
   and the accessed cache line line.
Begin
   if (cache miss)
      if (DFAD(line) = 0 or DFAD(line) ≥ PFAD(instr))
         replace line with the requested data;
         DFAD(line) ← PFAD(instr);
      else
         bypass the data access;
         DFAD(line) ← DFAD(line) - 1;
      endif
   else // cache hit
      DFAD(line) ← PFAD(instr);
   endif
End
```

Fig. 2. **Cache access procedure.**

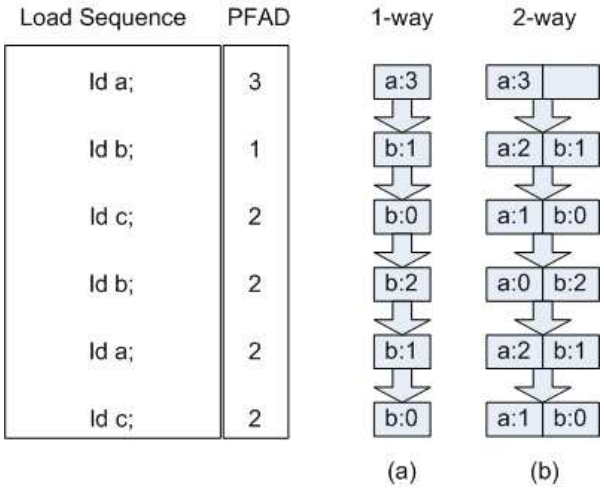

Fig. 3. Examples of our cache replacement policy in 1-way and 2-way cache. $x : n$ in each block indicates the DFAD of x is n.

Fig. 3(a) shows an example of our replacement policy in a direct mapped cache. Each block shows the cache state at the end of the data access. After the first data access, a is brought into the cache and its DFAD is updated to be 3 at the end of the data access. At the second data access, since "ld b" has smaller PFAD, a is replaced by b. Later on, b is kept in the cache all the time because its next reuse is always earlier than those of other data accesses. In this example, our cache replacement policy incurs one fewer miss than the traditional LRU cache replacement policy.

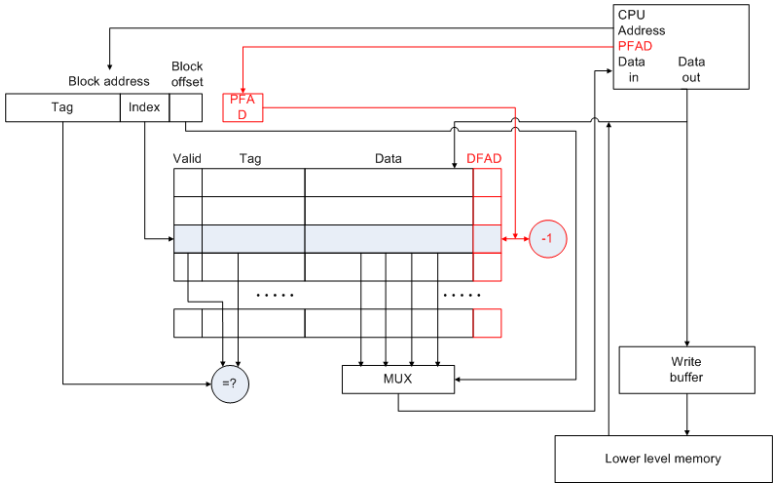

Fig. 4. Cache design.

Fig. 4 presents the design of hardware support for our cache replacement policy. For each cache line, we add several bits to hold the DFAD value. To transfer the PFAD from the memory access instruction to the corresponding cache line, we need extra bus lines connecting the processor and the cache. A simple subtractor is attached to the cache in order to update the DFAD after each data access. The DFAD maintenance

can be done at any time during or after the memory access is executed. Therefore we can easily hide its overhead by conducting it in parallel with the data access.

2.1.2. Extension for Set Associative Caches. For set associative caches, our cache replacement policy must decide not only when to bypass the cache but also which set element to evict when replacement is required. We extend our cache replacement for set associative cache as follows.

Data placement. When a cache miss occurs in a set associative cache, we first check the corresponding cache set to see if there is any cache line having 0 as DFAD. If so, we replace that cache line with the requested data. Otherwise, we compare the PFAD of the memory access instruction with the largest DFAD in the cache set. If the PFAD of the memory access instruction is larger, we directly send the data to the processor thus bypassing the cache. Otherwise, we select the cache line that has the largest DFAD in the set for replacement.

Finding the largest DFAD in a cache set takes time. Therefore to make this simple, we continuously track the largest DFAD and maintain a pointer that always points to the cache line with the largest DFAD in a cache set. Since the pointer must always point to the cache line with the largest DFAD, whenever a cache line's DFAD is updated, the new value is compared with the DFAD of the cache line pointed by the pointer. If the new value is larger, the pointer is updated to point to the updated cache line. In this approach, only a single comparison is required during each DFAD update to maintain the pointer to the largest DFAD cache line in the set.

DFAD maintenance. Like the DFAD maintenance for direct mapped cache, when a bypassing decision is made, all the cache lines in the related cache set need to decrement their DFADs. Besides, the DFAD of a cache line must be decremented when another cache line in the same cache set is hit or replaced since the access distance is defined with respect to the entire cache set.

Fig. 3(b) shows an example of our replacement policy in a 2-way cache. Each block shows the cache state at the end of the data access. After the first two data accesses, a and b are brought into the cache. At the third data access, "ld c" is bypassed since its PFAD is larger than a and b. The load sequence in this example visits three data elements repeatedly while the size of the cache set is only 2. Our cache replacement policy keeps a and b from the beginning to the end which prevents thrashing among a, b, and c.

2.1.3. Improvement with an Ideal Predictor. In this section, we show the potential of our access distance based cache replacement policy and discuss how many bits are needed to represent the DFAD for each cache line. Before presenting the data, we first briefly describe our experimental methodology.

Configuration. We conduct our experiments using Flexus [Hardavellas et al. 2004], which is a cycle-accurate full-system simulator built on Virtutech Simics [Magnusson et al. 2002]. Flexus models the SPARC ISA and allows commercial applications and operating systems to be executed without any modification. The configuration used in our experiments is summarized in Table I. We run the Solaris 10 operating system on the simulated processor. We use the LRU policy as the baseline when we show performance results.

Benchmarks. The SPEC CPU2006 benchmarks are used in our experiments. We also use a SPEC CPU2000 benchmark – `art`, which is a memory-intensive application. All benchmarks are executed using the reference inputs. For every benchmark, we use the SMARTS sampling approach [Wunderlich et al. 2003] to measure the cache misses and IPC for the complete execution. The sample size for each benchmark is 10000, the sample unit size is 1000 instructions, and the detailed warmup for each sample unit is

Table I. System parameters.

Processor	single-core, SPARC v9 ISA, 8-stage pipeline, out-of-order execution, 256-entry ROB, 8-wide dispatch, 32-entry store buffer
L1 Caches	Split ID, 64KB 2-way, 2-cycle latency, 2 ports, 32 MSHRs, 16-entry victim cache
L2 Shared Cache	1MB 16-way, 64B lines, 24-cycle latency, 1 port, 32 MSHRs, 16-entry victim cache
Main Memory	4GB total memory, 200-cycle access time

2000 instructions. According to [Wunderlich et al. 2003], this setting gives $99.7\% \pm 0.3\%$ confidence. Table II shows the characteristics of each benchmark under the LRU policy.

Table II. Benchmark summary. From left to right: accesses per 1000 instructions, misses per 1000 instructions, and percentage of compulsory misses out of total misses.

Name	APKI	MPKI	Compulsory Misses
hmmer	4.69	2.44	0.01%
sphinx3	18.44	14.19	0.01%
lbm	41.30	26.96	0.39%
art	29.27	22.24	0.53%
libquantum	14.78	14.78	0.72%
dealII	4.44	0.76	1.06%
leslie3d	19.23	6.39	1.34%
bzip2	22.44	8.67	1.57%
mcf	49.56	39.12	1.86%
xalancbmk	13.10	10.24	2.51%
milc	17.62	11.62	3.52%
gcc	6.59	1.25	4.00%
bwaves	21.29	5.04	5.16%
perlbench	12.60	0.75	8.43%
h264ref	2.35	0.72	9.47%
omnetpp	39.50	31.08	10.03%
GemsFDTD	42.95	9.03	25.57%
gobmk	4.46	0.36	25.92%
zeusmp	14.59	2.12	26.98%
soplex	5.04	0.64	30.77%
gromacs	8.54	0.22	33.85%
cactusADM	8.37	4.53	48.07%
namd	10.44	0.05	56.59%
gamess	3.27	0.17	61.01%
wrf	7.60	0.02	70.56%
calculix	6.29	1.19	75.88%
tonto	7.94	0.13	76.12%
sjeng	1.31	0.33	78.35%
astar	8.98	0.79	89.92%
povray	10.72	0.05	98.28%

Potential for Improvement. To measure the potential of our replacement policy, we assume that we have an ideal access distance predictor that can precisely predict the forward access distance of each memory instruction instance. To achieve this, we execute each benchmark twice. We first scan the trace of each benchmark to collect the actual forward access distances and then use them to simulate our access distance based cache. With the ideal access distance predictor, our access distance-based cache policy equals the optimal cache replacement policy described in [Mattson et al. 1970].

Fig. 5 and Fig. 6 show the potential of our replacement policy in L2 MPKI compared to the LRU replacement policy. Given an ideal predictor, our replacement policy on average reduces the L2 cache misses by 34% across these benchmarks. Among the

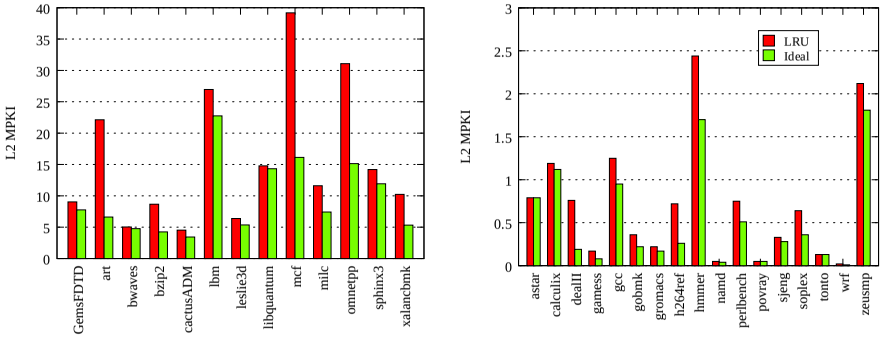

Fig. 5. L2 MPKI achieved by an ideal predictor.

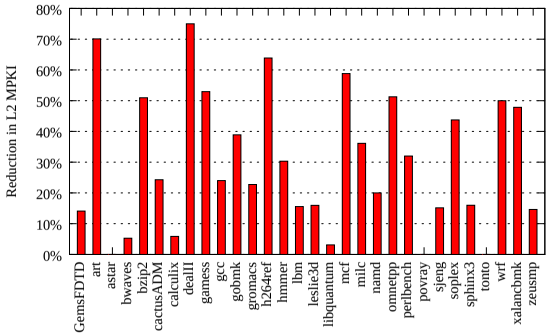

Fig. 6. Reduction in L2 MPKI achieved by an ideal predictor.

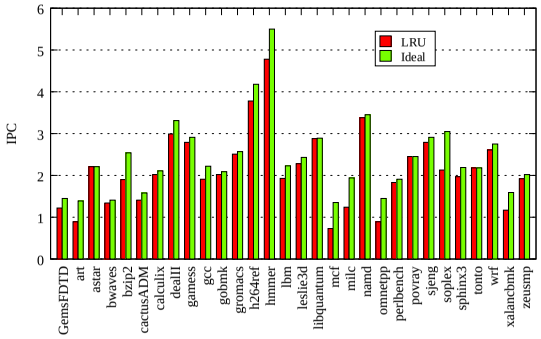

Fig. 7. IPC achieved by an ideal predictor.

25 benchmarks, the cache performance of `dealII` is improved the most, where nearly 75% of cache misses are eliminated. In the figure, three of the benchmarks do not have large potential ($< 1\%$ reduction) for improvement. This is because they have very high percentage of compulsory misses ($> 75\%$ compulsory misses). Overall, the huge reduction in cache misses shows our replacement policy has substantial potential for improving the L2 cache performance. We also measured the potential of our replacement policy for reducing L1 cache misses, which appears to be insignificant. This is because the LRU replacement policy works very well for L1 caches due to the data locality in the benchmarks. However, L2 caches only get the data requests that are

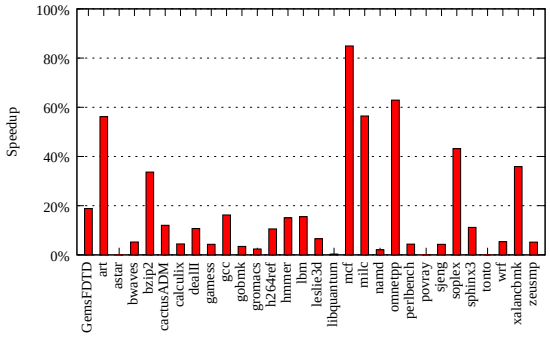

Fig. 8. Speedup achieved by an ideal predictor.

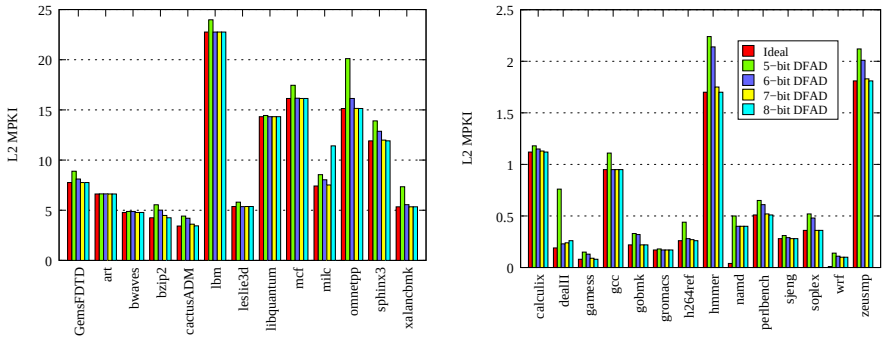

Fig. 9. L2 MPKI by an ideal predictor with various bits of DFAD.

missed in L1 caches. Therefore, the workloads for L2 caches have relatively low locality and the LRU policy in L2 caches has relatively poor performance. Fig. 7 and Fig. 8 show the potential of our replacement policy in IPC improvement in comparison to the LRU replacement policy. Given an ideal predictor, our replacement policy on average increases IPC by 16.0% across these benchmarks. Among these benchmarks, there are three benchmarks that have very little IPC improvement potential ($< 1\%$). This is because they have very high percentage of compulsory misses ($> 75\%$ compulsory misses) under the LRU policy. We remove these benchmarks from subsequent experiments and focus our attention on memory-intensive benchmarks.

Limiting Range of DFAD. In a realistic design of our access distance based cache we have to limit the range of DFAD values so that they can be stored in a few bits. Fig. 9 shows the L2 MPKI achieved by the ideal predictor when a limited number of bits are used to store DFAD for each cache line. When the PFAD given by the ideal predictor exceeds the range of the DFAD counter, the maximum possible value is set in the DFAD counter. We can see that 7 bits are enough to hold the DFAD for each cache line since the performance under 7-bit DFAD is nearly as good as the ideal performance. The reason is that the access distances in the benchmarks are rarely over 127. Moreover, almost every data access that has a PFAD of greater than 127 gets bypassed according to our replacement policy. Therefore, when a PFAD is greater than 127, its value rarely influences the replacement decision. For the benchmark `dealII`, the L2 MPKI increases with the number of bits in the DFAD counter. This is because some data blocks with long access distances (more than 127) are brought into the cache during execution of `dealII`. Due to the limit on DFAD length, the DFADs of these data

blocks can be set to at most 127. When their DFADs decrement to zero, these data blocks are found not to have been reused. Therefore, the cache resources taken by these data blocks are wasted. With more bits of DFAD counter, more cache resources are wasted since these data blocks spend more time in the cache.

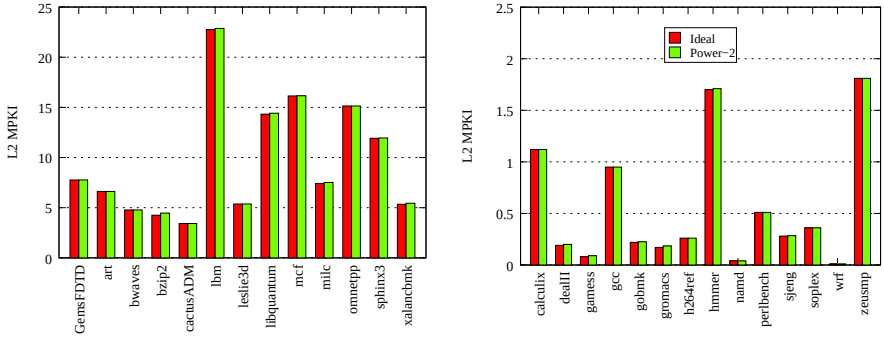

Fig. 10. Comparison of the an ideal predictor and a power-2 predictor.

To make our replacement policy work, we need to obtain the PFAD for each data access. However, it is very difficult to predict the exact PFAD for each data access since it is impossible to exactly know when a data block will be accessed again as it depends upon future execution path taken by the program. Therefore, we must use *approximate PFADs* instead of the exact PFADs in the replacement policy. An *approximate PFAD* is defined as the nearest larger power of 2 of the actual forward access distance. For very short access distance, the approximate PFAD is close to the exact PFAD. But the difference between the approximate and exact PFADs becomes larger with the increase of access distance. Fig. 10 compares the cache performance of an ideal predictor and a power-2 predictor. A power-2 predictor provides the nearest larger power of 2 of the actual forward access distance as the PFAD for each data access. From the Fig. 10, the performance of the power-2 predictor is nearly the same as the ideal performance, which means that the scaling of PFAD to the nearest larger power of 2 does not cause significant loss in the accuracy of the access distance information. Therefore, the *approximate PFAD* is a good replacement for the exact PFAD. Furthermore, the *approximate PFAD* can be represented by its logarithm $\lceil log_2(PFAD + 1) \rceil$, which takes much less storage space compared to the exact PFAD.

2.2. Access Distance Estimation

Our cache replacement policy needs the access distance information to help make the replacement decision. In this section, we present two access distance prediction techniques: a profile-based static technique and a runtime technique.

2.2.1. Static Access Distance Prediction. Static access distance prediction technique calculates the PFAD for each memory access instruction through profiling and then, at compile time, the PFAD is stored in the cache hint segment of each memory instruction. When a memory access instruction is executed, the processor retrieves the value from its cache hint segment and sends it to the cache to assist our cache replacement policy. The cache hint extension for memory access instructions was proposed in EPIC (explicitly parallel instruction computing) [Schlansker and Rau 2000] architectures in order to reduce the number of data cache misses at runtime. Embedding the cache hint extension in other ISAs may introduce some additional overhead. Each cache hint is a few bits attached to a memory access instruction. It is originally designed to specify

whether the instruction has temporal locality at a given cache level. Our technique uses the cache hint segment to hold the PFAD computed via profiling for each memory access instruction.

Our static prediction technique specifies the PFAD for each memory instruction based upon its observed forward access distances in the profiling runs. By profiling, we can easily collect all the forward access distances of a memory instruction during a program execution. However, it is difficult to precisely determine the PFAD for an instruction. Since the cache hint is specified in the instruction, each instruction can only have one cache hint. But at runtime, a single memory instruction can generate multiple data accesses with different forward access distances. Therefore, we set the PFAD of an instruction to be the *median* of its forward access distances.

Suppose at runtime, a memory load instruction ld generates a cache miss and the requested data is mapped into the cache line cs. We have the set of forward access distances of ld through profiling. Let us denote the median of ld's forward access distances as $FAD_m(ld)$ and the forward access distance of the data stored in cs as $FAD(cs)$. Consider the cache replacement in two cases. (Case 1) If $FAD_m(ld)$ is smaller than $FAD(cs)$, at least half of the forward access distances of ld are smaller than $FAD(cs)$. Therefore, it is more probable ($\geq 50\%$) that the forward access distance of the current data access generated by ld will be smaller than $FAD(cs)$. In other words, the requested data will probably be reused before the cache data. As a consequence, the requested data should be brought into the cache. (Case 2) If $FAD_m(ld)$ is larger than $FAD(cs)$, then at least half of the forward access distance of ld are larger than $FAD(cs)$. The cache data will probably be reused before the requested data. Therefore, the current data access generated by ld should bypass the cache. The above analysis shows that the median of an instruction's forward access distances is a good cache hint for cache replacement optimization.

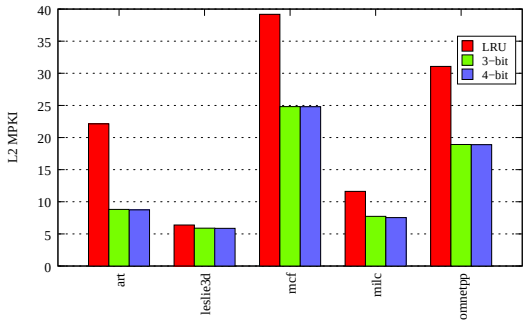

Fig. 11. 3-bit vs. 4-bit cache hint.

As discussed in Section 2.1.3, we store $\lceil log_2(PFAD + 1) \rceil$ in the cache hint for each memory instruction. Thus a 3-bit cache hint can hold the PFAD of at most 127 and a 4-bit cache hint for at most 32767. Fig. 11 shows the impact of using 3-bit and 4-bit cache hint. We did not examine the performance of 2-bit cache hint since it can only support PFAD of up to 7 if the PFAD is stored as $\lceil log_2(PFAD + 1) \rceil$. We used the five benchmarks that gave the most improvement in this tuning experiment for the static predictor – results of all benchmarks are presented later. From the figure, we can see that the performance of 3-bit cache hint is very close to that of 4-bit cache hint. However, using 3-bit cache hint can save 25% space compared to 4-bit cache hint. For the remainder of the paper, we use 3-bit cache hint and 7-bit DFAD counter for our

static predictor. The total space overhead taken by the DFAD counter is around 14KB for a 1MB cache with 64B cache lines.

2.2.2. Dynamic Access Distance Prediction. The static prediction consists of two phases: statically calculating the PFADs of the memory access instructions by profiling and setting the DFAD bits in the cache at runtime. The dynamic prediction follows the same two phases, but now the two phases are both conducted at runtime. In our replacement mechanism, when a processor executes a memory access instruction, it simultaneously sends an access distance request to a dynamic predictor. The predictor then sends the predicted access distance to the cache to assist the replacement procedure. Our prediction technique is an extension of the conventional last value prediction technique [Lipasti et al. 1996]. It does not need any software or compiler support and is thus fully transparent to the programmers.

The access distance of a memory access instruction usually does not change much at runtime. Given a memory access instruction, later we will show that it is safe to assume that its next access distance is often equal to its previous access distance. Therefore, our prediction technique uses the most recent access distance of a memory access instruction as its predicted assess distance. To realize our predictor, we need a history table to store the last access distances of the memory access instructions. Instead of storing the access distance information for every memory access instruction, the table only needs to store the frequently used access distances. To avoid outliers, the table is only updated after a new access distance appears twice in a row.

The history table is indexed by the instruction address. Each entry of the table contains the instruction address tag, the PFAD, and the usage counter of a memory access instruction. The PFAD is stored in the form of $\lceil log_2(PFAD + 1) \rceil$. The access distance is updated using two-delta policy [Eickemeyer and Vassiliadis 1993]. In this policy, the access distance gets updated only if a value different from the correct access distance appears twice in a row. To implement this policy, two access distances are stored in each entry. The first access distance stores the value used for prediction and the second access distance stores the value calculated most recently. The first access distance is only updated when the newly calculated access distance is equal to the second access distance but different from the first access distance. The usage counter of each entry is a saturating counter that is incremented after every correct prediction and decremented after every misprediction. When a new entry needs to be created, the one with the lowest usage counter is replaced.

To maintain the aforementioned table, our predictor needs to calculate the access distance for each executed memory instruction. The access distance of a memory access instruction is unknown at the time when it is executed. We can only obtain the access distance when the same cache line is accessed again. Therefore, we record the address of the instruction that previously accessed the line in the cache with a tag in that line. The tag will be used to calculate the PFAD and update the access distance table.

When a processor executes a memory access instruction, it sends its instruction address to the dynamic predictor to request its access distance. Upon receiving the request, the dynamic predictor performs two actions: predicting the access distance and updating the history table. These two actions can be done in parallel as described below.

Access distance prediction. Upon receiving the instruction address from the processor, the predictor first computes the index in the history table by hashing the instruction address. If the corresponding set of entries contain the information of the memory instruction, the PFAD stored in the first access distance column is returned as the predicted access distance. If the history does not have the information of the memory, the

value (cache associativity - 1) is returned as default access distance – as we show later in Section 2.2.3, this essentially approximates the LRU policy.

Access distance update. Upon receiving the requested data address, the predictor calculates the access distance of the instruction that previously accesses the requested data block in the following ways.

— If the requested data block is hit in the cache, the predictor can get the address tag of the previous accessing instruction from the cache. Its new PFAD is set to the difference between the PFAD stored in the history table and the remaining DFAD in the cache line.
— If the requested data block is missing in the cache, the predictor can get the address tag of the instruction that previously accesses the evicted cache line. If the DFAD value of the evicted cache line is equal to 0, we know the cache line has not been accessed with its PFAD. We then double its PFAD in the history table.

After we get the new PFAD, we update the record by using two-delta policy. To implement the two-delta policy, we have two access distance records for each entry in the history table, as shown in Fig. 12. Access distance #1 is used for prediction and access distance #2 stores the most recent access distance. We only update the access distance #1 when the same access distance occurs twice in a row. For example, when a cache line L is hit, we can get the remaining DFAD from the cache line. Based on the address tag in the cache line, we can also get the access distance record of the previous accessing instruction from the history table. The forward access distance of the previous accessing instruction is then calculated by subtracting the remaining DFAD from the access distance #1 stored in the history table. If the new forward access distance is equal to the access distance #2, we update the access distance #1 to the new forward access distance. If they are different, we update the access distance #2 to the new forward access distance and keep the access distance #1 unchanged. If the history table does not contain a record for the instruction, a new record is created to take the place of the one with the lowest usage counter.

Like the conventional value prediction techniques, the proposed access distance prediction scheme can be implemented entirely in hardware. The predicted access distances are used for making the bypassing/replacement decision. In a real machine, the requested data can be directly sent to the processor before the bypassing/replacement decision is made. Therefore, the prediction does not have to be performed before the memory access is executed. Since the prediction actions are not on the critical path of a memory access, the memory access latency under the new scheme will be the same.

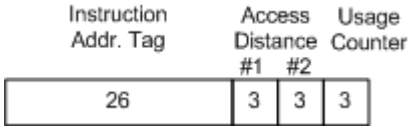

Fig. 12. Table entry line for the history table.

Fig. 13 shows the impact of the history table size on six benchmarks that have the highest L2 MPKI under the LRU policy – results of all benchmarks are presented later. We use a 4-way history table in this tuning experiment. We can see that 256-entry history table can work nearly as well as an infinitely large history table. Fig. 12 shows the size of an entry line of the history table. Each entry line of the history table takes 35 bits and therefore the total space overhead for a 256-entry history table is (35*256/8)=1120B.

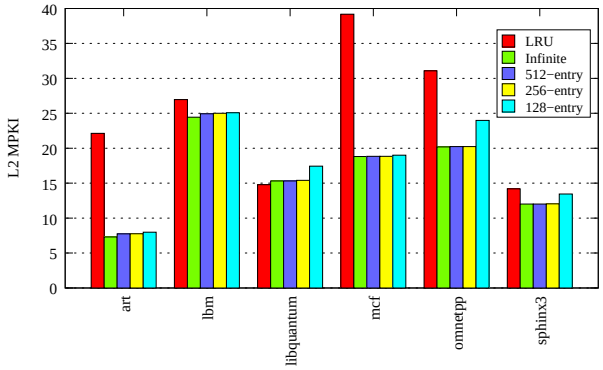

Fig. 13. Comparison of different history table sizes.

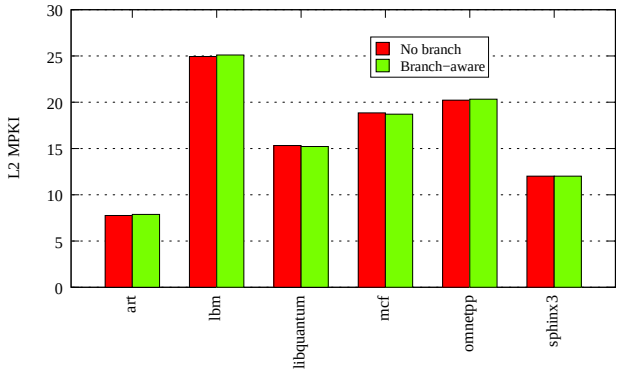

Fig. 14. Impact of branch-aware dynamic predictor.

We also compare the dynamic predictor with a branch-aware predictor. Instead of predicting the access distance for a memory access instruction regardless of the execution path, a branch-aware predictor estimates the access distances according to the branch history. Since the access distances of a memory access instruction may be different along different execution paths, the branch-aware predictor may provide more accurate prediction. To realize the branch-aware predictor, the branch history is maintained in a register. When locating an access distance record in the history table, the predictor hashes both the instruction address and the branch history into the index.

Fig. 14 compares the branch-aware dynamic predictor with the normal dynamic predictor. For `libquantum` and `mcf`, the branch-aware predictor reduces MPKI slightly more than the normal predictor. This happens because the forward access distances of the memory instructions in these three benchmarks are different along different paths and thus the branch-aware predictor can provide more precise estimation. On the other hand, the normal predictor can store more memory instructions' PFAD information given the same history table size. Therefore, it outperforms the branch-aware predictor on other benchmarks.

2.2.3. Default Estimation. To fully take advantage of our cache replacement policy, the processor has to know the access distance information for the given program. However, sometimes it may not be possible for the processor to generate accurate information. In this section, we introduce default access distance strategy whose behavior is very sim-

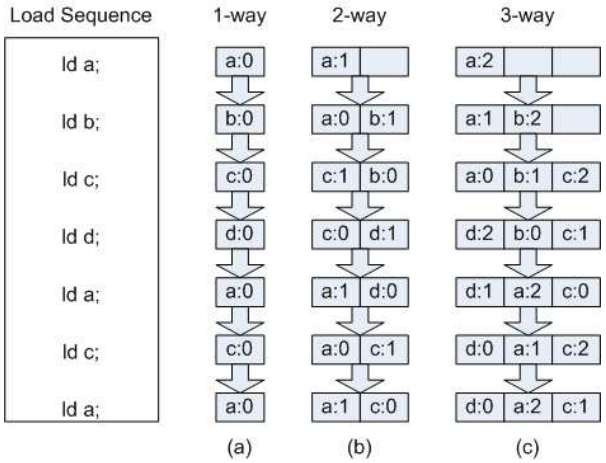

Fig. 15. Examples of using our cache replacement mechanism using default estimation. $x : n$ in each block indicates the DFAD of x is n.

ilar to the well-known LRU policy. This has an additional advantage that selectively incorporating the default estimations into the access distance driven cache replacement policy can make the cache performance better.

When the access distance information is unavailable, the default PFAD of each data access is set to $n-1$ for a n-way cache. By the default estimation, our cache replacement policy behaves exactly the same as the LRU policy in a 1-way or 2-way cache and approximates the LRU policy in a n-way cache where $n \geq 3$.

1-way cache. Fig. 15(a) shows an example for our cache replacement policy in a 1-way cache by setting the PFAD of each memory access instruction to be 1. Every time a cache line is accessed, its DFAD is set to be 1. Then at the end of this cache access, the DFAD is decreased to 0. Therefore, no bypassing will be allowed since the DFADs on all cache lines are always 0.

2-way cache. Fig. 15(b) shows an example for our cache replacement policy in a 2-way cache using default estimates. Whenever a cache line is accessed, the DFAD on the other cache line in the same cache set is at most 1 since DFADs keep decreasing if not being reset. Thus at the end of this data access, the DFAD on the other cache line will be decreased to 0. Next time a data element needs to be brought in, the other cache line will be replaced. This cache behavior is exactly the same as the LRU policy.

3-way or more cache. Default estimation can also be used in a n-way cache when access distance information is unavailable. When a cache miss happens, our cache replacement policy places the new data into a cache line which is not accessed in the last $n-1$ data accesses. To explain this, we assume a cache miss happens at the i^{th} data access. During the previous $n-1$ data accesses, at least one cache line is not accessed since there are n cache lines in a set. The DFAD on that cache line must be 0 because it keeps decreasing since the $(i-n)^{th}$ data access. Therefore, our cache replacement policy will place the new data onto that cache line which is not accessed in the last $n-1$ data accesses. Fig. 15(c) shows an example. When multiple cache lines are not accessed in the last $n-1$ data accesses, our cache replacement policy will randomly choose one to be replaced. The cache behavior in this case is similar to the LRU policy since our cache replacement policy keeps the most recently used data element in the cache and evicts a relatively less recently used data element from the cache.

The above analysis shows that by setting access distances using our default estimation strategy, we can closely approximate LRU policy. In other words, the classical

LRU policy can be approximately seen as a particular case of our cache replacement policy by assuming every memory access instruction has the same access distance.

Table III. Comparison of the LRU policy and its approximation with default estimation.

Name	LRU (MPKI)	Approximation (MPKI)
hmmer	2.44	2.46
sphinx3	14.19	14.16
lbm	26.96	26.96
art	22.24	22.25
libquantum	14.78	14.78
dealII	0.76	0.78
leslie3d	6.39	6.35
bzip2	8.67	8.72
mcf	39.12	39.12
xalancbmk	10.24	10.24
milc	11.62	11.63
gcc	1.25	1.26
bwaves	5.04	5.04
perlbench	0.75	0.75
h264ref	0.72	0.71
omnetpp	31.08	31.06
GemsFDTD	9.03	9.03
gobmk	0.36	0.39
zeusmp	2.12	2.12
soplex	0.64	0.61
gromacs	0.22	0.23
cactusADM	4.53	4.54
namd	0.05	0.05
gamess	0.17	0.17
wrf	0.02	0.02
calculix	1.19	1.18
tonto	0.13	0.13
sjeng	0.33	0.34
astar	0.79	0.78
povray	0.05	0.05

Table III shows the comparison of the LRU policy and our cache replacement policy with default estimation. Since the L2 cache is 16-way, we use 15 as the PFAD for each memory access instructions. We observe that the performance of our method is almost the same as that of the LRU policy. The performance difference is within $\pm 2\%$.

The default estimation can be integrated into our access distance prediction techniques to further enhance cache performance. When the prediction accuracy for a program is low, the default estimation can be used to approximate the LRU policy to avoid the cache performance degradation due to low prediction accuracy. In particular, we extend our prediction techniques to enable them to adaptively choose between our access distance driven cache replacement policy and the approximate LRU policy. This extension to the predictors does not introduce any extra hardware complexity.

Adaptive static prediction. In our static prediction technique, each memory instruction can only be tagged with one PFAD. Therefore, for those instructions that have multiple forward access distances at runtime, our static prediction technique cannot always produce accurate prediction. The inaccurate prediction may cause a wrong replacement decision. Based on the profiling results, it is easy to know whether a replacement decision is correct or not by comparing it with the decision generated by the ideal predictor. We can score an access distance prediction using the percentage of correct replacement decisions caused by this prediction. For a memory instruction, we choose between the

static prediction and the default estimation based on their scores. If the static prediction has higher score, we choose it for this memory instruction since using static prediction for it can cause more correct replacement decision. If the default estimation has higher score, we use the default estimation to approximate the LRU policy to avoid the performance degradation due to the low prediction accuracy.

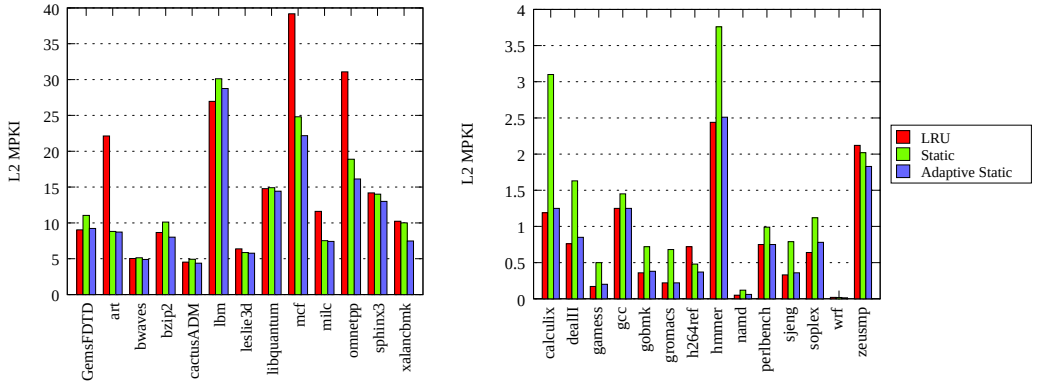

Fig. 16. Comparison of the original static predictor and the adaptive static predictor in L2 MPKI.

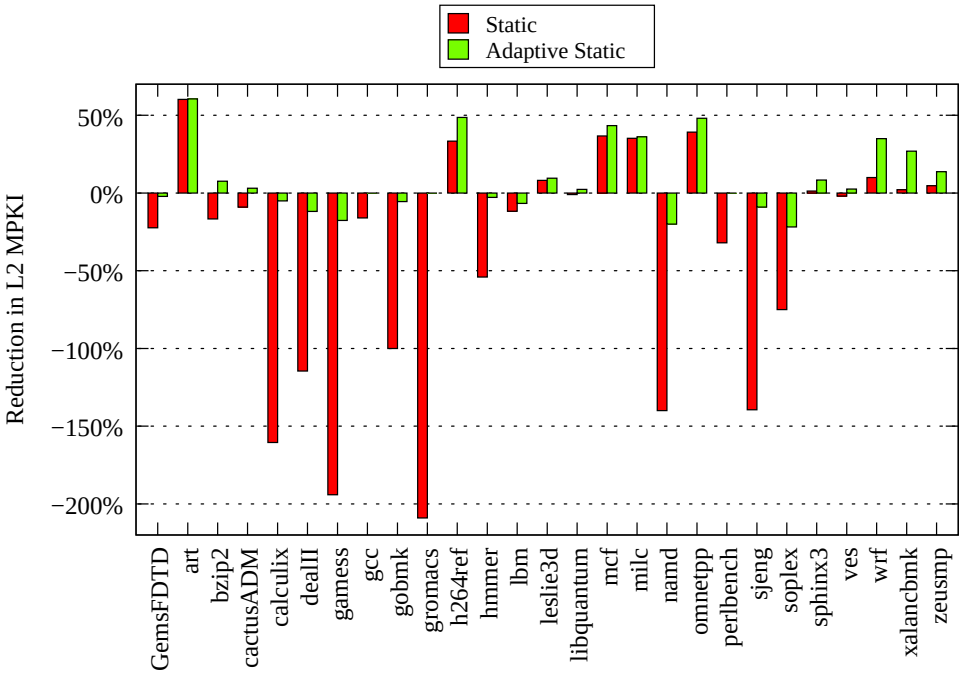

Fig. 17. Comparison of the original static predictor and the adaptive static predictor in L2 MPKI reduction.

Fig. 16 shows the L2 MPKI of both the original static predictor and the adaptive static predictor in comparison to the LRU policy. The left side includes the memory-intensive benchmarks with MPKI larger than 4 and the right side shows the rest. Fig.

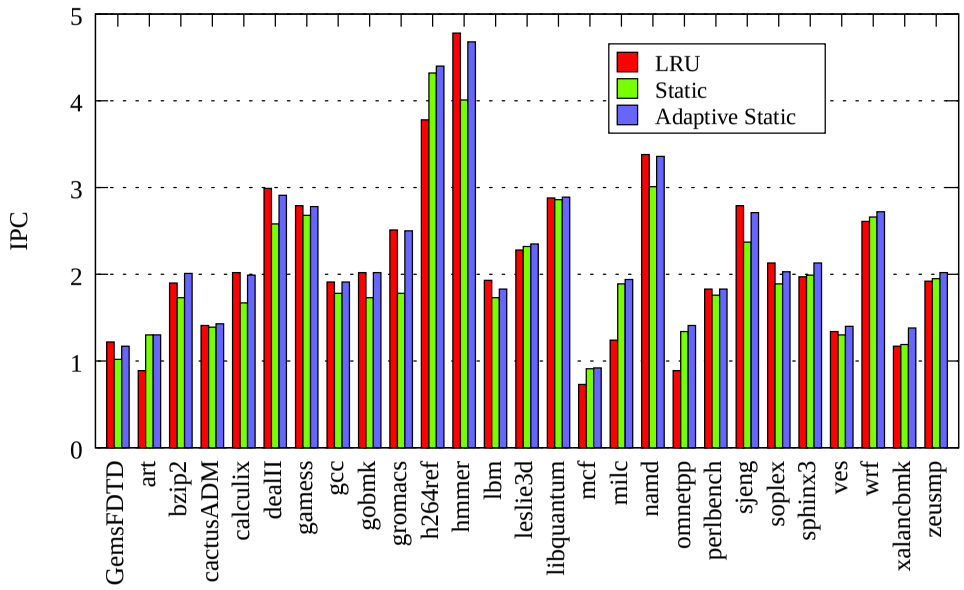

Fig. 18. Comparison of the original static predictor and the adaptive static predictor in IPC.

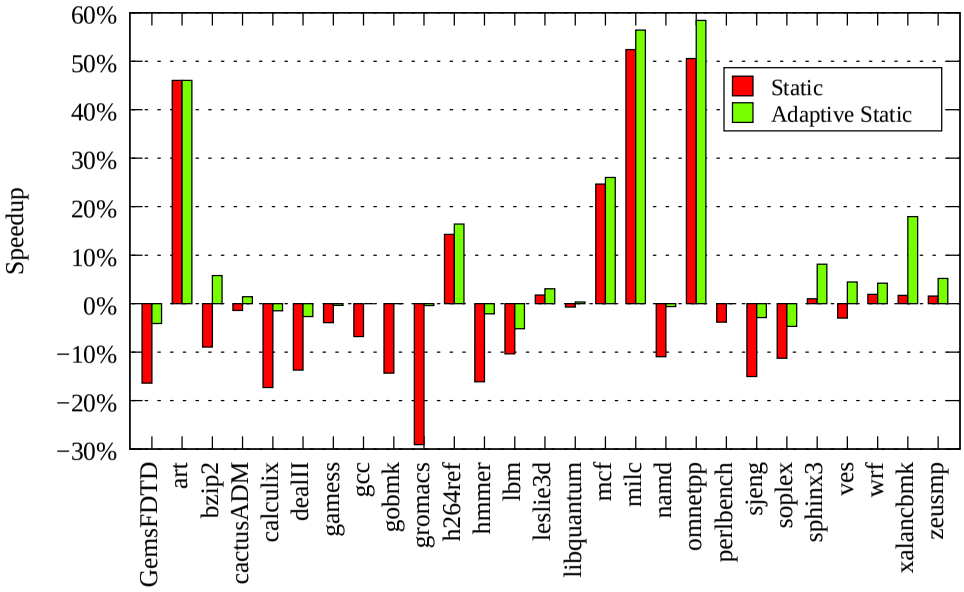

Fig. 19. Comparison of the original static predictor and the adaptive static predictor in speedup.

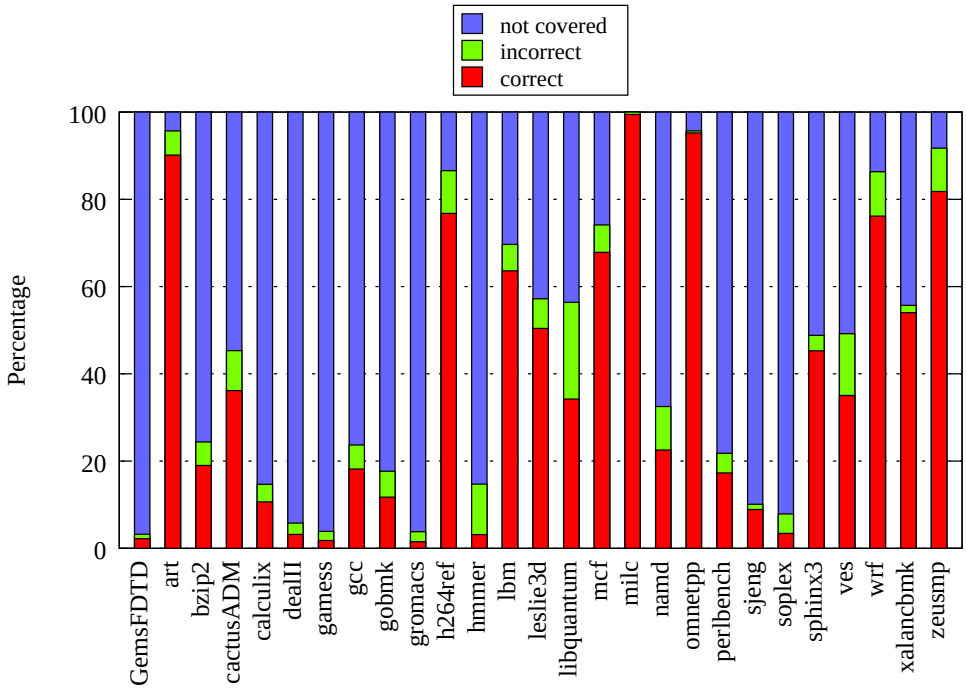

Fig. 20. Accuracy and coverage of the adaptive static predictor.

17 shows the reduction in L2 MPKI achieved by both predictors. In the experiment, we used the training inputs for profiling and the reference inputs for measuring the performance. Our original static predictor significantly reduces the L2 MPKI for five benchmarks, where the MPKI for `art` is reduced by around 60%. Our original static predictor significantly increased the MPKI of fifteen benchmarks. This is because their data access patterns vary at runtime and therefore it is not possible for the static predictor to set the cache hint accurately. The adaptive technique makes the static predictor always work better than or close to the LRU policy. For the benchmarks that are slowed down by the original static predictor, the adaptive static predictor chooses default estimation since the static prediction makes more wrong replacement decision. On average, the L2 MPKI across these benchmarks is reduced by around 12.32% with the adaptive static predictor.

Fig. 18 shows the system performance of both the original static predictor and the adaptive static predictor. Fig. 19 shows the IPC improvement over the LRU policy of both predictors. On average, the IPC across these benchmarks is increased by 7.32% with the adaptive static predictor. The IPC of six benchmarks `art`, `mcf`, `h264ref`, `milc`, `omnetpp`, and `xalancbmk` is increased by more than 10%.

Fig. 20 shows the accuracy and coverage of the adaptive static predictor. For ten benchmarks, the adaptive static predictor gives correct access distance estimations for more than 50% instruction instances. On average, 40.6% instruction instances are given correct estimations. The adaptive static predictor applies default estimation to 53.8% instruction instances. Only 5.6% instruction instances are given wrong estimations.

Adaptive dynamic prediction. Some memory instructions' forward access distances keep changing at runtime. Therefore it is not possible for our dynamic predictor to es-

timate their forward access distances accurately. As mentioned in Section 2.2.2, each access distance record in the history table has a usage counter, which indicates the confidence of the record. Similar to the adaptive static prediction, our adaptive dynamic predictor returns the default estimation when receiving a request related to a record with low confidence.

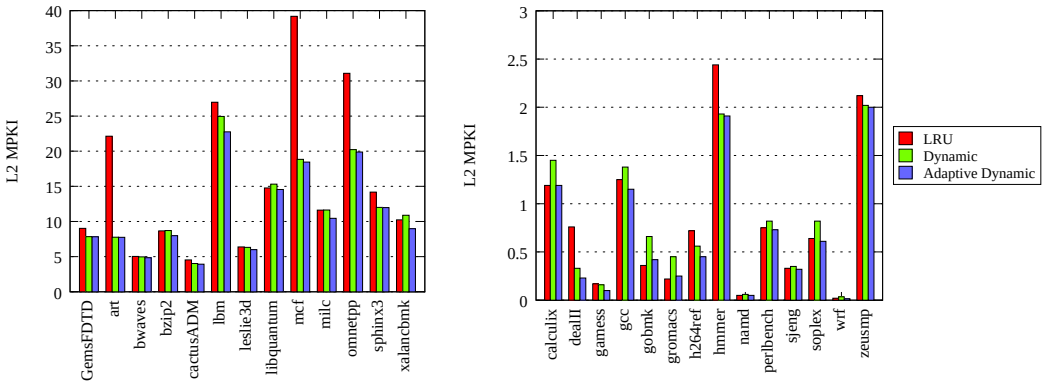

Fig. 21. Comparison of the original dynamic predictor and the adaptive dynamic predictor in L2 MPKI.

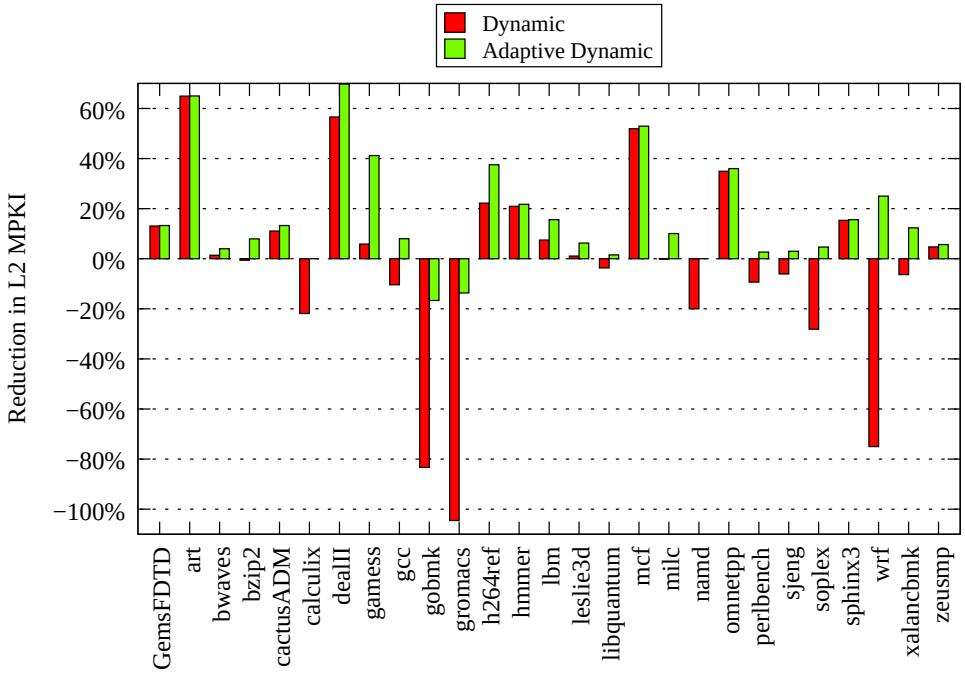

Fig. 22. Comparison of the original dynamic predictor and the adaptive dynamic predictor in L2 MPKI reduction.

Fig. 21 shows the L2 MPKI of both the original dynamic predictor and the adaptive dynamic predictor Fig. 22 shows the reduction in L2 MPKI achieved by both predictors.

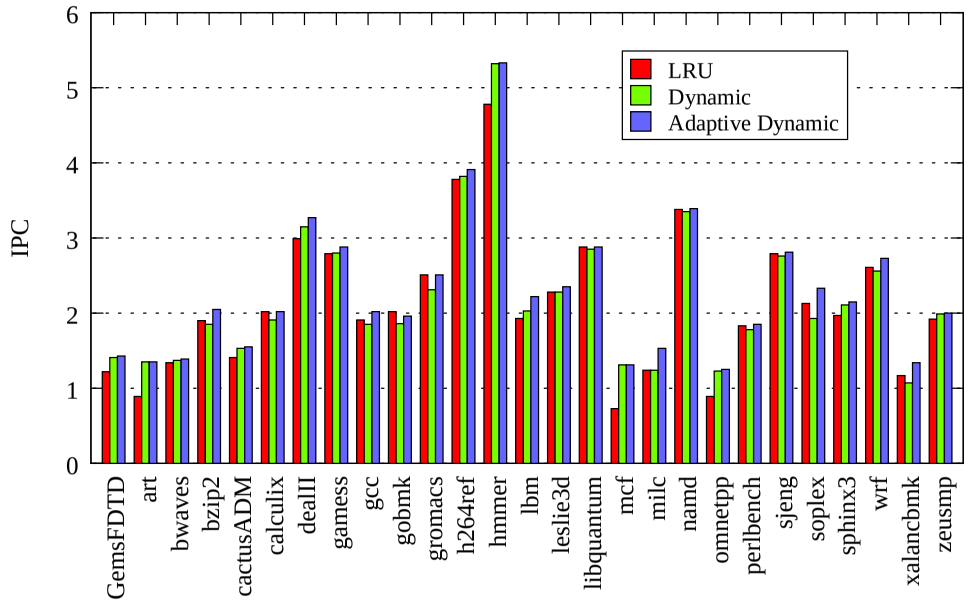

Fig. 23. Comparison of the original dynamic predictor and the adaptive dynamic predictor in IPC.

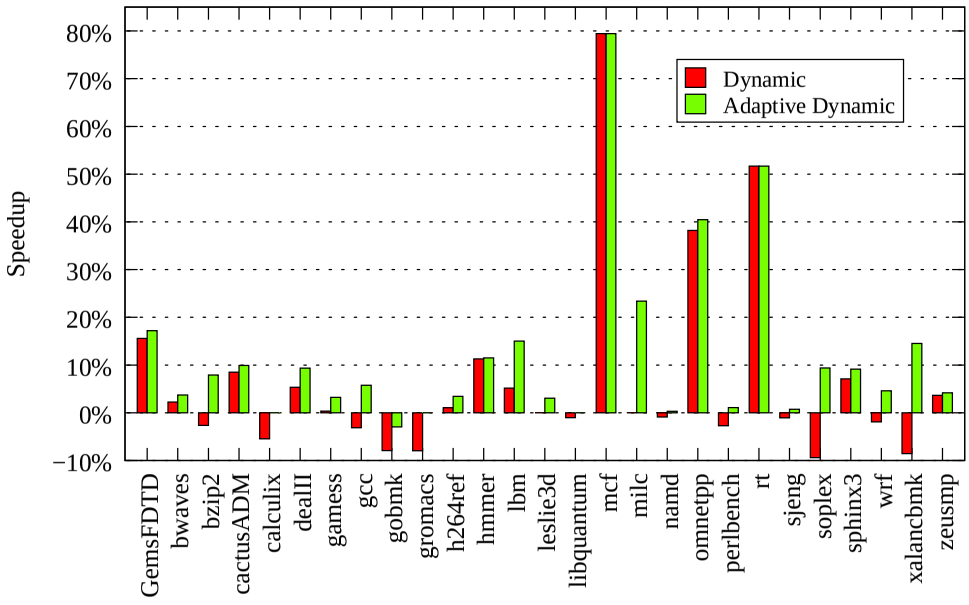

Fig. 24. Comparison of the original dynamic predictor and the adaptive dynamic predictor in speedup.

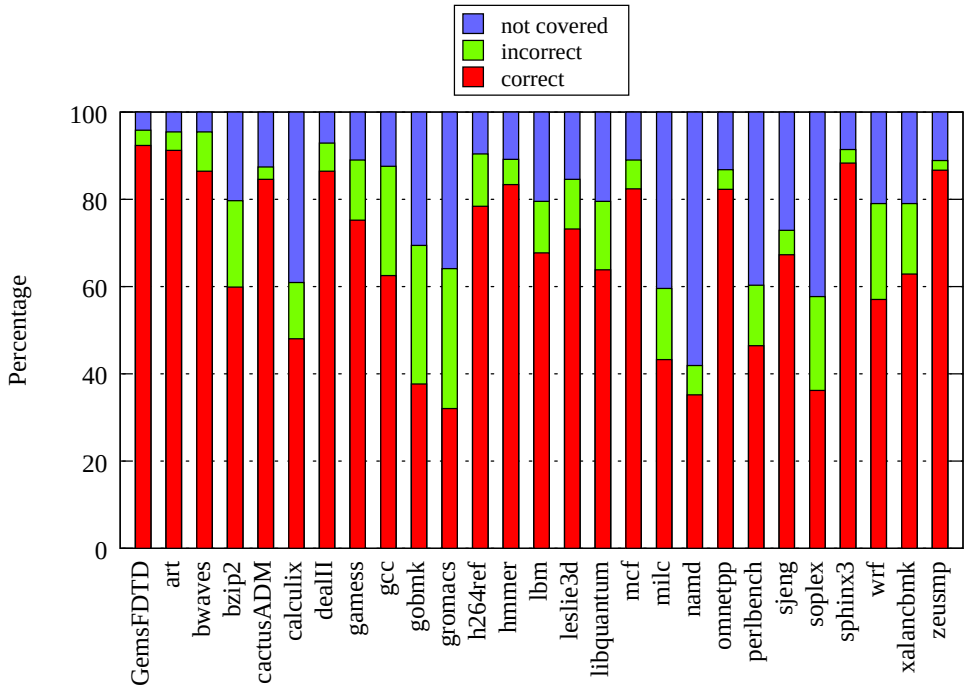

Fig. 25. Accuracy and coverage of the adaptive dynamic predictor.

The original predictor reduces the MPKI by more than 10% for thirteen benchmarks. For six benchmarks `art`, `mcf`, `h264ref`, `hmmer`, `omnetpp`, and `dealII`, the original dynamic predictor outperforms the LRU policy more than 20%. The average reduction in L2 MPKI achieved by the original dynamic predictor is 14.16%. The original dynamic predictor increases the L2 misses slightly for five benchmarks. As shown in the figure, the adaptive technique significantly reduces the MPKIs of the benchmarks that incur an MPKI increase with the original dynamic predictor. The adaptive dynamic predictor does not increase the MPKI of any benchmark for more than 1% except for `gobmk` and `gromacs`. On average, the L2 MPKI across the benchmarks is reduced by around 19.95% with the adaptive dynamic predictor.

Fig. 23 shows the system performance of both the original dynamic predictor and the adaptive dynamic predictor. Fig. 24 shows the IPC improvement over the LRU policy achieved by both predictors. On average, the IPC across these benchmarks is increased by 10.91% with the adaptive dynamic predictor. The IPC of eight benchmarks is increased by more than 10%.

Fig. 25 shows the accuracy and coverage of the adaptive dynamic predictor. For ten benchmarks, the adaptive static predictor gives correct access distance estimations for more than 80% instruction instances. On average, almost 70% of instruction instances are given correct estimations. The adaptive static predictor applies default estimation to 16.5% instruction instances. Only 14.4% instruction instances are given wrong estimations.

Our adaptive dynamic predictor can adapt to different phases of the same application. Fig. 26 shows the L2 miss reductions obtained by the adaptive dynamic predictor in different phases for 4 SPEC CPU2000 benchmarks. We ran each benchmark for 60B instructions and sampled a segment of 250M instructions from every 2B instructions

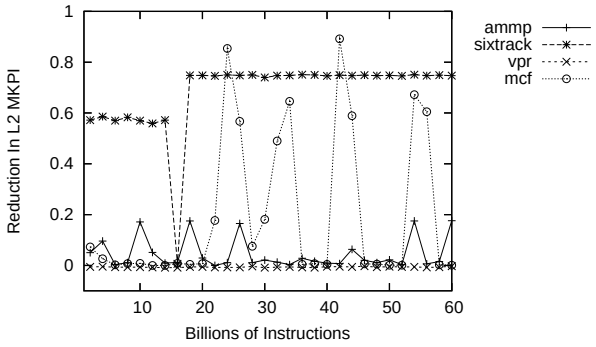

Fig. 26. Reduction in L2 MPKI during benchmark execution.

for measurement. For `sixtrack`, our technique outperforms the LRU policy in all except for the 14B-16B instruction segment. For `ammp` and `mcf`, their data access patterns change during the execution. Our technique works better than the LRU policy on part of the segments and approximates the LRU policy on the other segments. For `vpr`, the forward access distances of most memory instructions keep changing at runtime. Therefore, our dynamic predictor cannot estimate `vpr`'s access distance information well. Therefore, due to the low prediction rate, our adaptive dynamic predictor always uses the default estimation to approximate the LRU policy.

2.3. Hardware Cost

Table IV. Hardware cost of our access distance-based cache for 1MB cache size.

	Regular Cache	w/ Sta. Est.	w/ Dyn. Est.
Space Cost (bytes)	1,048,576	+14,336	+31,840
Area (mm^2)	7.3435	+0.1333	+0.3109
Dynamic energy (nJ)	1.2987	+0.0262	+0.0611
Dynamic power at max freq (W)	3.1725	+0.0307	+0.0980
Standby leakage power per bank (W)	0.7618	+0.0094	+0.0203
Access Time (ns)	2.4416	+0.0179	+0.0404

We study the hardware cost of our access distance-based cache. We use CACTI [Thoziyoor et al. 2008] to model access time, area, leakage, and dynamic power of our cache. We model a 1MB 16-way cache with 64B linesize and 1 read/write port. All numbers are calculated by using 45nm technology size.

Table IV compares the space/area cost, dynamic/leakage power, and access time of our access distance-based cache with a regular cache, which only contains two SRAM components – a data array and a tag array. Energy and power are shown in per read port. Compared to a regular cache, our access distance-based cache needs an additional 7-bit DFAD counter for each cache line. The total storage overhead of the counters is 14KB for a 1MB cache, around 1.3% of the total cache size. This increases the hardware area of the cache by 1.82%. The dynamic energy consumption per read port is 2.01% more than that of a regular cache. The access time of our access distance-based cache is slightly longer than that of a regular cache. Our access distance-based cache can work with a dynamic access distance predictor. To enable dynamic prediction, an 8-bit address tag of the previously accessing instruction is stored in each cache line. The total storage cost of the address tags is 16KB. The history table takes 1120B storage cost for a 1MB cache. It costs 0.15mm^2 area, around 2% of the area of a regular cache.

A cache with a dynamic access distance predictor costs 4.7% more energy per read port. The access time increases by 1.8% due to the additional tag on each cache line.

3. ADDITIONAL EVALUATION

In this section, we compare our techniques with other cache replacement policies. We also evaluate our techniques with different L2 cache sizes. All the experiments are conducted by using the adaptive predictors.

3.1. Comparison with Other Replacement Policies

In this section, we compare our techniques with other cache replacement policies. Dynamic Insertion Policy (DIP) [Qureshi et al. 2007] is a scheme for resolving cache thrashing. It dynamically selects between the MRU insertion policy and the Bimodal Insertion Policy (BIP). BIP randomly selects LRU or MRU position for inserting new data. Access Interval Predictor (AIP) [Kharbutli and Solihin 2005] is another counter-based cache replacement policy. We compare our techniques with these two policies.

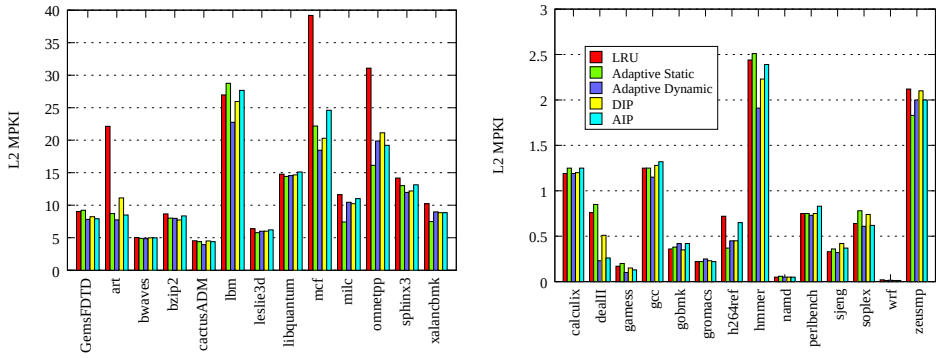

Fig. 27. MPKI comparison of cache replacement policies.

Fig. 27 and Fig. 28 compare the L2 MPKI of our techniques, DIP, and AIP. AIP outperforms both LRU and DIP for most benchmarks. On average, DIP reduces the L2 MPKI by 12.41% for these benchmarks compared to LRU and AIP reduced L2 MPKI by 13.38%. The overall performance of our policy with the adaptive dynamic predictor is best among all five policies. It outperforms DIP for 19 out of 27 benchmarks and AIP for 21 out of 27 benchmarks. On average, our access distance-based cache with the adaptive dynamic predictor reduces the MPKI by 19.95% over LRU.

Fig. 29 and Fig. 30 compare the system performance of our techniques, DIP, and AIP. Compared to LRU, DIP improves the system performance by 7.1% while AIP improves it by 7.2%. Our technique with the adaptive dynamic predictor outperforms the other replacement policies for most benchmarks shown in the figure. Overall, it improves the system performance by 10.91% for these benchmarks.

Fig. 31 and Fig. 32 show the accuracy and coverage of different cache replacement policies. We define *correct evictions* as ones that agree with the optimal policy, *wrong evictions* as ones that disagree with the optimal policy, and *unpredicted evictions* as the optimal evictions that are not predicted. Accuracy is equal to *correct evictions* divided by the sum of *correct evictions* and *wrong evictions*. Coverage is equal to *correct evictions* divided by the sum of *correct evictions* and *unpredicted evictions*. The figures show that on average, our adaptive static and adaptive dynamic predictors correctly predict more evictions than DIP and AIP (71% for the adaptive static predictor, 74%

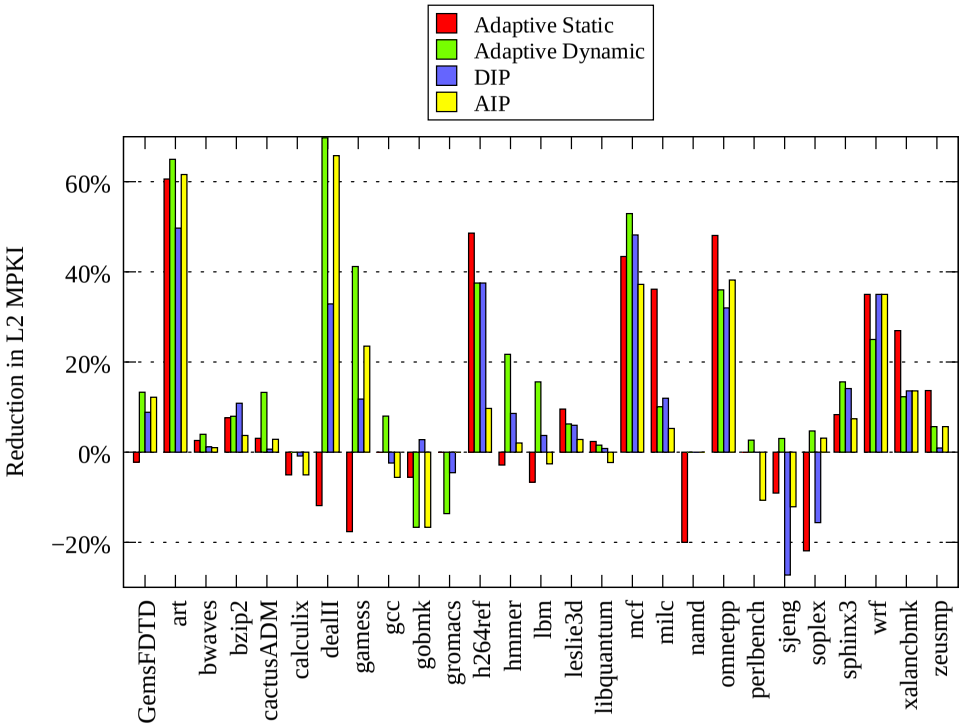

Fig. 28. Comparison of cache replacement policies in L2 MPKI reduction.

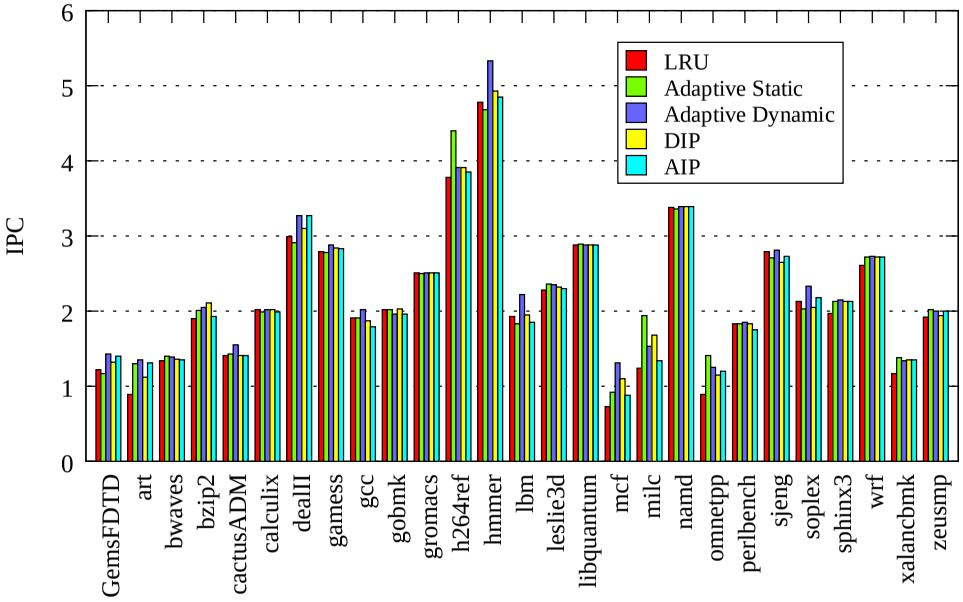

Fig. 29. IPC comparison of cache replacement policies.

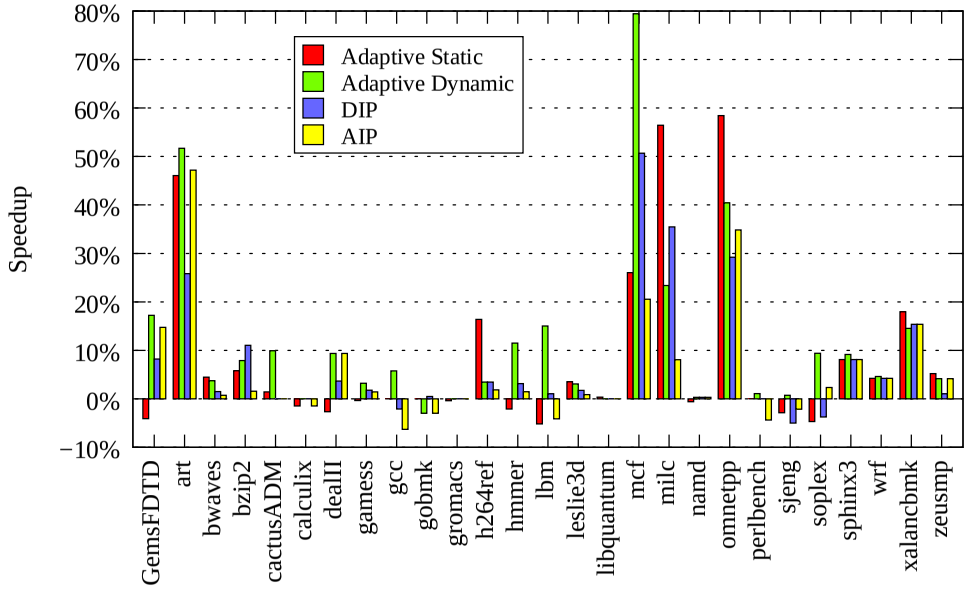

Fig. 30. Speedup comparison of cache replacement policies.

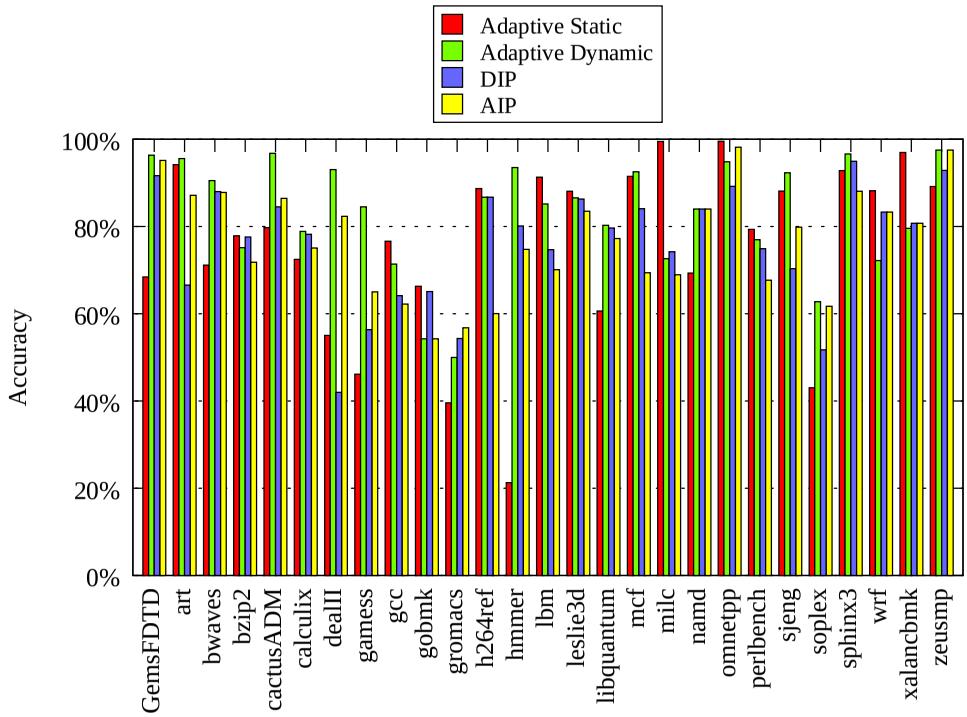

Fig. 31. Accuracy of cache replacement policies.

for the adaptive dynamic predictor, and 64% for both DIP and AIP). The adaptive dy-

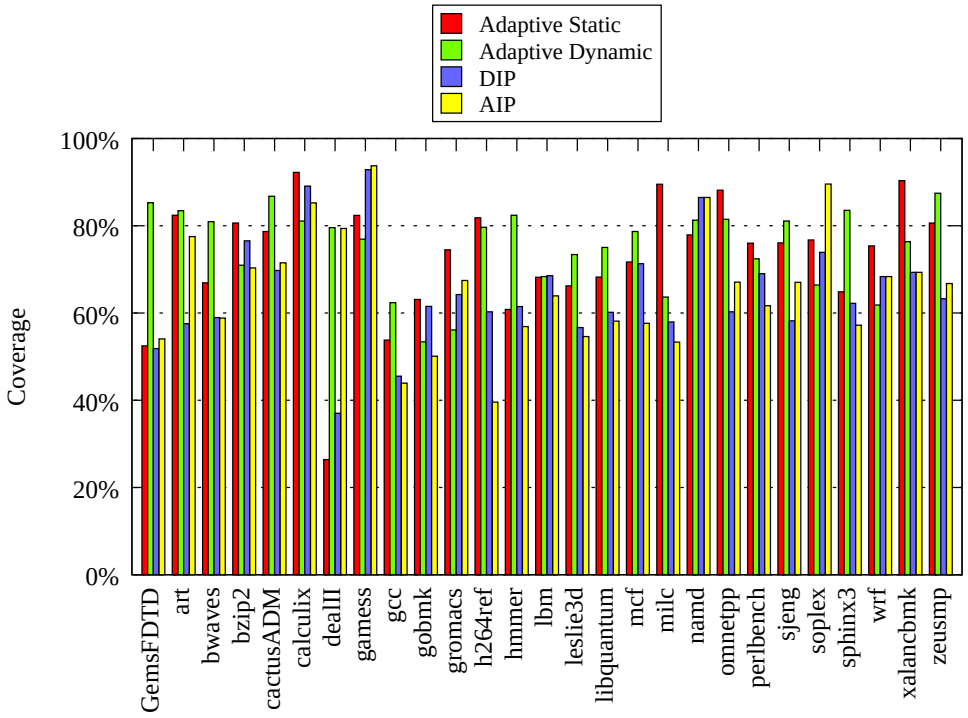

Fig. 32. Coverage of cache replacement policies.

namic predictor also gives the least wrong predictions (19% for the adaptive dynamic predictor, 29% for the adaptive static predictor, 25% for DIP, and 24% for AIP).

3.2. Varying the Cache Size

We study the performance of our access distance based cache for different cache sizes. Fig. 33 shows the impact for different cache sizes. For most benchmarks, our adaptive predictor achieves more MPKI reduction on smaller cache. This occurs because the improvement potential for larger caches is smaller since they can store more data blocks. For `bzip2`, our adaptive dynamic predictor can achieve more MPKI reduction for a larger cache. The reason is that its working set is very large. Even with 4MB cache, its cache performance still has potential for improvement. On average, our method reduces the cache misses by 4.9% and 3.3% with the adaptive static predictor for 2MB and 4MB caches respectively. With the adaptive dynamic predictor, the average reduction is 5.8% and 2.3%. Although the caches will be larger in the future, the working sets of the programs will also get larger. Thus, our method can still benefit performance in future designs.

4. RELATED WORK

Cache Replacement Based on Reuse Information. MIN [Belady 1966] and OPT [Mattson et al. 1970] are two optimal cache management strategies based on forward reuse distance. Neither of them can be implemented in hardware. MIN is very costly to implement because it requires forward scanning to find a data element with furthest reuse when a cache miss happens. The OPT algorithm is proposed based on MIN which has slightly lower overhead than MIN. The OPT algorithm is a two pass algorithm.

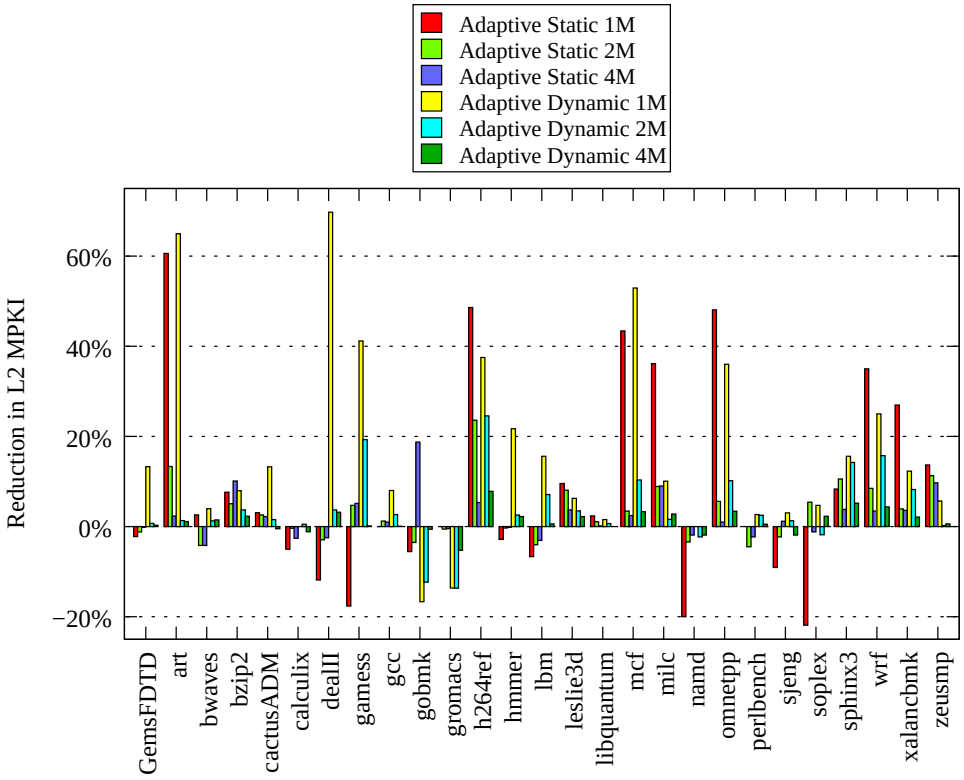

Fig. 33. MPKI reduction in L2 cache with different cache sizes.

In the first pass, it computes the forward reuse distance for each data access. Then in the second pass, it incrementally maintains a priority list based on the forward reuse distance of every cache element. Once a cache miss happens, the requested data element is inserted into the priority list based on its forward reuse distance. OPT needs to have the exact forward reuse distance for each data access. This is similar to our replacement policy with an ideal predictor. However, it is not implementable as such information is not available in practice.

Gu et al. presented a *Program-directed OPTimal cache management (P-OPT)* [Gu et al. 2008], which annotates certain accesses with bypass based on the cache access history obtained by executing the program with OPT cache replacement policy. For each eviction generated by OPT, P-OPT tags the last access of the evicted data as a bypassing access. Similar to the OPT algorithm, P-OPT needs to look up the bypass tag of each data access at runtime, which is hard to implement. Similar to P-OPT, Wang et al. [Wang et al. 2002] proposed a set of compiler algorithms to predict whether a data element will be reused. According to the reuse information, they set a one-bit tag, called evict-me, for each cache line. On a cache replacement, the hardware replaces a line where the evict-me bit is set.

Beyls and D'Hollander [Beyls and D'Hollander 2002] proposed a reuse distance-based method for statically setting cache hint in EPIC architecture [Schlansker and Rau 2000]. For each memory access instruction, their method picks a cache level where the requested data will be retained for at least 90% of the accesses generated by that instruction at runtime. Therefore, the data with smaller reuse distance will be inserted

into lower cache level while the data with larger distance will be evicted into higher cache level. Beyls and D'Hollander [Beyls and D'Hollander 2005] later developed a dynamic cache hint selection method based upon the polyhedral model. Their method calculates the reuse distance for each access at runtime and then selects the memory access instruction with the proper cache hint according to the calculated reuse distance. The dynamic method achieves lower miss rate than the static method but the overhead of calculating the forward reuse distances can be quite large.

Kharbutli and Solihin [Kharbutli and Solihin 2005] proposed two counter-based cache replacement algorithm. Access Interval Predictor (AIP) is similar to adaptive dynamic predictor. However, since AIP does not bin access distances into power-of-2 bins, a confidence counter is only increased when the same access distance is encountered twice. This makes it hard to increase confidence counters for the instructions that have approximately same access distances for different execution instances. Besides, AIP requires more bits to store access distance history than our dynamic predictor. Live-time Predictor (LvP) is another counter-based algorithm, which predicts how many times a cache line is accessed before it is evicted. It is similar to dead block prediction. Keramidas et al. [Keramidas et al. 2007] proposed a counter-based replacement policy, which makes replacement decision based on two counters. One counter (ETA) records how long the cache line is expected to be in the cache and the other counter (CD) stores how long the cache line has been in the cache. The victim is selected between the line with the largest ETA counter and the line with the largest CD counter.

Re-Reference Interval Prediction (RRIP) [Jaleel et al. 2010] is a recently-proposed cache replacement technique based on the Not Recently Used (NRU) policy [HP 2002]. Instead of predicting the reuse distance of a cache block, it predicts the reference prediction value, which is a M-bit value that indicates how soon the cache block is rereferenced. When $M = 1$, RRIP is identical to the NRU replacement policy.

Cache Bypassing. Many research works have been done in cache bypassing and early eviction. McFarling [McFarling 1992] recognized the common instruction reference patterns where storing an instruction in the cache actually harms performance. He then proposed a technique for reducing direct-mapped cache conflict misses by excluding the harmful instruction. Many works [Gonzalez et al. 1995; Johnson 1998; Tyson et al. 1995; Wang et al. 2002; Wong and Baer 2000] present techniques for bypassing or early eviction by using locality information. The underlying idea is to bypass the data accesses which have low reuse. Another series of publications focus on cache optimization by predicting the last touch of a cache line [Lai et al. 2001; Lin and Reinhardt 2002]. By knowing the last-touch references, the cache line can be turned off after the last touch to save energy [Kaxiras et al. 2001]. Lai et al. [Lai et al. 2001] proposed to use the dead cache line to store the prefetched data. Khan et al. [Khan et al. 2010] recently proposed a new dead block prediction technique that samples program counters to determine when a cache block is likely to be dead. Rajan and Govindarajan [Rajan and Ramaswamy 2007] proposed to divide the cache into two components: a Main Cache and a Shephard Cache. The Shephard Cache is used to buffer a missing line until a Main Cache replacement decision is made based on the data access order after the cache miss.

Several studies have investigated the cache bypassing policy based on sampling. Etsion and Feitelson [Etsion and Feitelson 2007] proposed a L1 cache filtering mechanism based on random sampling to identify and select the frequently used data blocks. Their method reduces the number of conflict misses by inserting only frequently used block into the main cache. Qureshi et al. [Qureshi et al. 2007] proposed a similar technique for improving L2 cache with LRU policy. Their method selects the data access to be placed in the LRU position based on random sampling, which prevents thrashing when the workload size is greater than the cache size. Their technique was later

extended for shared caches in multicore architectures [Jaleel et al. 2008]. The shortcoming of the sampling-based bypassing policy is that its performance heavily relies on the data access patterns of the programs. In order to make the sampling-based bypassing policy adapt to different applications, Qureshi et al. [Qureshi et al. 2007] proposed the Set Dueling technique that samples cache sets dedicated to different replacement policy and dynamically chooses between the LRU policy and the sampling-based bypassing policy.

Recently, Gao and Wilkerson proposed to apply the Segmented LRU (SLRU) policy as a processor cache replacement algorithm [Gao and Wilkerson 2010]. The SLRU policy divides the cache lines into two lists – the referenced list and the non-referenced list. The LRU cache line in the non-referenced list is selected first when choosing a line for replacement. They also proposed three enhancements to the SLRU policy.

Other techniques. Alameldeen and Wood [Alameldeen and Wood 2004] proposed a compression technique for L2 cache so that the cache can hold more data. They dynamically check for each cache line whether compression will eliminate a miss or incur an unnecessary decompression overhead. Based on the results, they decide whether to apply compression to the cache line.

5. CONCLUSION

In this paper, we proposed a dynamic access distance driven cache that makes the replacement decision based on the data locality. The cache consists of two parts: the access distance based replacement policy and the access distance predictors. We evaluate the proposed replacement policy and predictors in detail. According to the experimental results, our cache can work well in both high-locality and low-locality workloads and adapt to different benchmarks. Our experiments show that L2 cache misses are reduced by 12.32% and 19.95% using profiling-based static and runtime adaptive predictors respectively.

REFERENCES

ALAMELDEEN, A. R. AND WOOD, D. A. 2004. Adaptive cache compression for high-performance processors. In *ISCA*.

BELADY, L. A. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal 5,* 2, 78–101.

BEYLS, K. AND D'HOLLANDER, E. 2002. Reuse distance-based cache hint selection. In *Euro-Par*.

BEYLS, K. AND D'HOLLANDER, E. 2005. Generating cache hints for improved program efficiency. *Journal of Systems Architecture 51,* 4, 223–250.

EICKEMEYER, R. J. AND VASSILIADIS, S. 1993. A load instruction unit for pipelined processors. *IBM Journal of Research and Development*, 547–564.

ETSION, Y. AND FEITELSON, D. G. 2007. L1 cache filtering through random selection of memory references. In *PACT*.

GAO, H. AND WILKERSON, C. 2010. A dueling segmented LRU replacement algorithm with adaptive bypassing. In *JWAC*.

GONZALEZ, A., ALIAGAS, C., AND VALERO, M. 1995. A data cache with multiple caching strategies tuned to different types of locality. In *ICS*.

GU, X., BAI, T., GAO, Y., ZHANG, C., ARCHAMBAULT, R., AND DING, C. 2008. P-opt: Program-directed optimal cache management. In *LCPC*. 217–231.

HARDAVELLAS, N., SOMOGYI, S., WENISCH, T. F., WUNDERLICH, R. E., CHEN, S., KIM, J., FALSAFI, B., HOE, J. C., AND NOWATZYK, A. G. 2004. Simflex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *SIGMETRICS Perform. Eval. Rev. 31,* 4, 31–34.

HP. 2002. Inside the Intel Itanium 2 processor. *HP Technical White Paper*.

JALEEL, A., HASENPLAUGH, W., QURESHI, M. K., SEBOT, J., STELLY, S., AND EMER, J. 2008. Adaptive insertion policies for managing shared caches. In *PACT*.

JALEEL, A., THEOBALD, K. B., STEELY, JR., S. C., AND EMER, J. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *ISCA*. 60–71.

JOHNSON, T. L. 1998. Run-time adaptive cache management. *PhD thesis, University of Illinois, Urbana, IL*.

KAXIRAS, S., HU, Z., AND MARTONOSI, M. 2001. Cache decay: exploiting generational behavior to reduce cache leakage power. In *ISCA*.

KERAMIDAS, G., PETOUMENOS, P., AND KAXIRAS, S. 2007. Cache replacement based on reuse-distance prediction. In *ICCD*.

KHAN, S. M., TIAN, Y., AND JIMENEZ, D. A. 2010. Sampling dead block prediction for last-level caches. In *MICRO*. 175–186.

KHARBUTLI, M. AND SOLIHIN, Y. 2005. Counter-based cache replacement algorithms. In *ICCD*.

LAI, A., FIDE, C., AND FALSAFI, B. 2001. Dead-block prediction & dead-block correlating prefetchers. In *ISCA*.

LIN, W. AND REINHARDT, S. 2002. Predicting last-touch references under optimal replacement. *Technical Report CSE-TR-447-02, University of Michigan*.

LIPASTI, M. H., WILKERSON, C. B., AND SHEN, J. P. 1996. Value locality and load value prediction. In *ASPLOS*. 138–147.

MAGNUSSON, P. S., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HÅLLBERG, G., HÖGBERG, J., LARSSON, F., MOESTEDT, A., AND WERNER, B. 2002. Simics: A full system simulation platform. *Computer 35*, 50–58.

MATTSON, R. L., GECSEI, J., SLUTZ, D., AND TRAIGER, I. L. 1970. Evaluation techniques for storage hierarchies. *IBM System Journal 9,* 2, 78–117.

MCFARLING, S. 1992. Cache replacement with dynamic exclusion. In *ISCA*. 191–200.

QURESHI, M. K., JALEEL, A., PATT, Y. N., STEELY, S. C., AND EMER, J. 2007. Adaptive insertion policies for high performance caching. In *ISCA*.

RAJAN, K. AND RAMASWAMY, G. 2007. Emulating optimal replacement with a shepherd cache. In *MICRO*. 445–454.

SCHLANSKER, M. AND RAU, B. 2000. Epic: Explicitly parallel instruction computing. *Computer 33,* 2, 37–45.

THOZIYOOR, S., AHN, J., MONCHIERO, M., BROCKMAN, J., AND JOUPPI, N. 2008. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *ISCA*.

TYSON, G., FARRENS, M., MATTHEWS, J., AND PLESZKUN, A. R. 1995. A modified approach to data cache management. In *MICRO*.

WANG, Z., MCKINLEY, K. S., ROSENBERG, A. L., AND WEEMS, C. C. 2002. Using the compiler to improve cache replacement decisions. In *PACT*.

WONG, W. A. AND BAER, J.-L. 2000. Modified lru policies for improving second-level cache behavior. In *HPCA*.

WUNDERLICH, R. E., WENISCH, T. F., FALSAFI, B., AND HOE, J. C. 2003. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA*.