

# Dynamic and Discrete Cache Insertion Policies for Managing Shared Last Level Caches in Large Multicores

Aswinkumar Sridharan, André Seznec

▶ **To cite this version:**

Aswinkumar Sridharan, André Seznec. Dynamic and Discrete Cache Insertion Policies for Managing Shared Last Level Caches in Large Multicores. Journal of Parallel and Distributed Computing, 2017, 106, pp.215-226. 10.1016/j.jpdc.2017.02.004 . hal-01519650

**HAL Id: hal-01519650**

**https://inria.hal.science/hal-01519650**

Submitted on 9 May 2017

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Dynamic and Discrete Cache Insertion Policies for Managing Shared Last Level Caches in Large Multicores*

Aswinkumar Sridharan & André Seznec

May 9, 2017

INRIA/IRISA
Campus de Beaulieu
35042 Rennes, France
{aswinkumar.sridharan , andre.seznec}@inria.fr

## 1 Introduction

In multi-core processors, the Last Level Cache (LLC) is usually shared by all the cores (threads)[1]. The effect of inter-thread interference due to sharing has been extensively studied in small scale multi-core contexts [4, 3, 12, 6, 7, 8, 1, 5, 9, 11, 2]. However, with advancement in process technology, processors are evolving towards packaging more cores on a chip. Future multi-core processors are still expected to share the last level cache among threads. Consequently, future multi-cores pose two new challenges. Firstly, the shared cache associativity is not expected to increase beyond *around sixteen* due to energy constraints, though there is an increase in the number of applications sharing the last level cache of multi-core processors. For example, Intel Xeon E7-4850 [39] processors host sixteen cores with each core capable of running *two* threads and a 40 MB, 20-way associative shared (L3) last level cache. That is, the last level cache, which is 20-way associative, is shared by thirty-two threads. Hence, we are presented with the scenario of managing shared caches where ($\#applications \geq \#llc_ways$).

Secondly, in large scale multi-core systems, the workload mix typically consists of applications with very diverse memory demands. For efficient cache management, the replacement policy must be aware of such diversity to enforce different priorities across applications. Moreover, in commercial grid systems, the computing resources (in particular, memory-hierarchy) are shared across multiple applications which have different fairness and performance goals. Either the operating system or the hypervisor takes responsibility in accomplishing these goals. Therefore, the hardware must provide scope for the software to enforce different priorities for the applications. Altogether, the cache replacement policy must satisfy two requirements (i) allow enforcing *discrete* priorities across applications and (ii) efficiently capture an application's run-time behavior.

*To appear in JPDC 2017. This article is an extension of the work published at IEEE International Parallel and Distributed Processing Symposium, 2016.

[1]Without loss of generality, we assume one thread/application per core. And, we use thread/application or core interchangeably.

**Problem:** Prior studies [4, 3, 12, 1, 5, 2] have proposed novel approaches to *predict* the reuse behavior of applications and, hence their ability to utilize the cache. The typical approach is to observe the hits/misses it experiences as a consequence of sharing the cache and approximate its behavior. This approach *fairly reflects* an application's ability to utilize the cache when the number of applications sharing the cache is small (2 or 4). However, we observe that this approach *may not necessarily reflect* an application's ability to utilize the cache when it is shared by a large number of applications with diverse memory behaviors. Consequently, this approach leads to incorrect decisions and cannot be used to enforce different priorities across applications. We demonstrate this problem with an example.

**Solution:** Towards this goal, we introduce the metric *Footprint-number*. Footprint-number is defined as the number of unique accesses (cache block addresses) that an application generates to a cache set in an interval of time. Since Footprint-number explicitly approximates the working set size, *and quantifies* the application behavior at run-time, it naturally provides scope for discretely (distinct and more than *two* priorities) prioritizing applications. We propose an *insertion-priority-prediction* algorithm that uses application's *Footprint-number* to *assign* priority to the cache lines of applications during cache replacement (insertion) operations. Since Footprint-number is computed at run-time, dynamic changes in the application behavior are also captured. We further find that probabilistically *de-prioritizing* certain applications during cache insertions (that is, not inserting the cache lines) provides a scalable solution for efficient cache management.

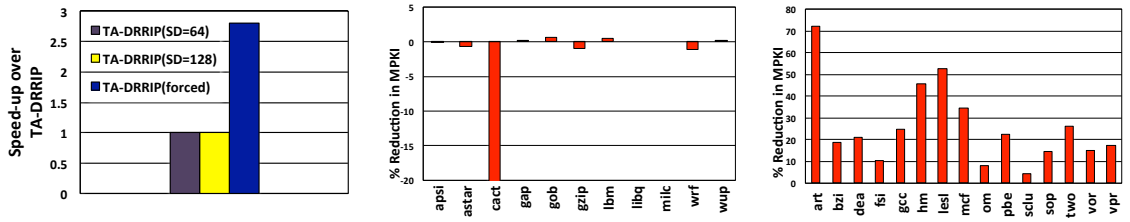
The remainder of the paper is organized as follows: In Section 2, we motivate the need for a new cache monitoring technique followed by detailing the proposed replacement algorithm in Section 3. We describe the experimental setup and evaluate our proposal in Sections 4 and 5, respectively. Related work and our concluding remarks are presented in Sections 6 and 7, respectively.

# 2 Motivation

In this section, we motivate the need for altogether a new mechanism to capture application behavior at run-time and a replacement policy that differently prioritizes applications.

## 2.1 Cache Management in large-scale multi-cores:

A typical approach to approximate an application's behavior is to observe the hits and misses it encounters at the cache. Several prior mechanisms [3, 1, 12, 5, 2] have used this approach: the general goal being to assign cache space (not explicitly but by reuse prediction) to applications that could utilize the cache better. This approach works well when the number of applications sharing the cache is small (2 or 4 cores). However, such an approach becomes suboptimal when the cache is shared by a large number of applications. We explain with set-dueling [4] as an example. ***Set-dueling:*** a randomly chosen pool of sets (Pool A for convenience) exclusively implements one particular insertion policy for the cache lines that miss on these sets. While another pool of sets (Pool B) exclusively implements a *different* insertion policy. A saturating counter records the misses incurred by either of the policies: misses on *Pool A* increment, while the misses on *Pool B* decrement the saturating counter, which is 10-bit in size. The switching threshold between the two policies is 512. They observe that choosing as few as 32 sets per policy is sufficient. Thread-Aware Dynamic ReReference Interval Predictions, TA-DRRIP [1] uses set-dueling to learn between SRRIP and BRRIP insertion policies. SRRIP handles scan (long sequence of no reuse cache lines)




(a) Forced BRRIP priority (b) MPKI for thrashing applications (c) MPKI reduction for others

Fig. 1: Impact of implementing BRRIP policy for thrashing applications.

and mixed (recency-friendly pattern mixing with scan) type of access patterns, BRRIP handles thrashing (larger working-set) patterns.

In the context of large-scale multicore systems, however, TA-DRRIP learns SRRIP policy for all classes of applications. For applications with working-set larger than the cache, BRRIP is the desired policy. However, TA-DRRIP, which uses set-dueling to learn the application behavior, is not able learn BRRIP policy for those applications, thereby allowing them to thrash in the cache. However, by explicitly preventing such applications from competing with the non-thrashing (cache-friendly) applications, performance can be improved. In other words, implementing BRRIP policy for thrashing applications will be beneficial to the overall performance. Fig. 1a confirms this premise. The bar labeled TA-DRRIP(forced) is the implementation where we force BRRIP policy on all the thrashing applications. Performance is normalized to TA-DRRIP. From the figure, we observe the latter achieves speed-up close to 2.8 over the default implementation of TA-DRRIP, which uses set-dueling (recording the misses of the competing policies) to learn the suitable policy. The experiments are performed on a 16MB, 16-way associative cache, which is shared by all sixteen applications. Table 3 shows other simulation parameters. Results in Fig. 1 are averaged from all the 60 16-core workloads. In the figure, bars 1 and 2 shows the result of TA-DRRIP which uses 64 and 128 sets, as opposed to 32 sets in the default implementation, to learn the application behavior, respectively. The two bars confirm that the observed behavior is independent of the number of sets dedicated for policy policy learning.

Fig. 1b and Fig. 1c show the MPKIs of individual applications when thrashing applications are forced to implement BRRIP insertion policy. For thrashing applications, there is little change in their MPKIs, except *cactusADM*. cactusADM suffers close to 40% increase in its MPKI and 8% reduction in its IPC while other thrashing applications show a very marginal change in their IPCs. However, non-thrashing applications show much improvement in their MPKIs and IPCs. For example, in Fig. 1c, *art* saves up to 72% of its misses (in MPKI) when thrashing applications are forced to implement BRRIP insertion policy. Thus, thrashing applications implementing BRRIP as their insertion policy is beneficial to the overall performance. However, in practice, TA-DRRIP does not implement BRRIP for thrashing applications and loses out on the opportunity for performance improvement. Similarly, SHiP[5] which learns from the hits and misses of cache lines at the shared cache, suffers from the same problem. Thus, we *infer* that observing the hit/miss results of cache lines to approximate application behavior is not efficient in the context of large-scale multi-cores.

## 2.2 Complexity of other approaches :

While the techniques that predict the individual application's behavior from its shared behavior are not efficient, reuse distance based mechanisms [41, 32, 33, 44, 35, 36, 34] provide a fair approximation of the application's behavior. However, to accurately predict the reuse behavior of individual cache accesses of the applications involves significant overhead due to storage and their related bookkeeping operations. Further, these techniques are either dependent on the replacement policy [35, 36] or require modifying the cache tag arrays [32, 44, 34]. Similarly, some cache partitioning techniques do not scale with the number of cores while others scale but incur significant overhead due to larger (up to 128/256-way) LRU managed shadow tag structures [29, 28, 10, 30].

*From these discussions, we see that, in large scale multi-core systems, a simple and efficient monitoring mechanism to approximate application behavior and which in turn allows the replacement policy to enforce discrete priorities across applications is required.*

## 2.3 A case for discrete application prioritization:

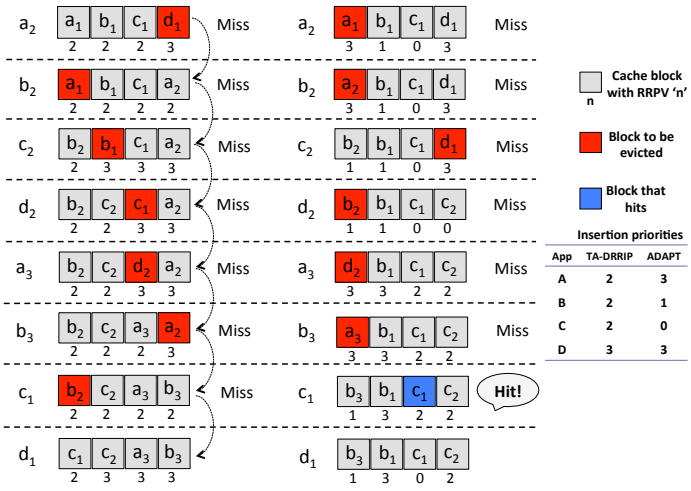
Before presenting our proposed mechanism, we present an example to show the benefit of discretely prioritizing application in the context of large multi-cores, where the number of applications or threads sharing in the cache matches/exceeds the associativity of the shared cache.

Like TA-DRRIP, ADAPT also uses 2-bit RRPV counter per cache line to store the reuse predictions. On cache replacements, both the policies evict the cache line with RRPV 3 to accommodate the new block. However, the difference is that on insertions ADAPT makes *discrete* (four) predictions[2] while TA-DRRIP inserts either with RRPV 2 or RRPV 3.

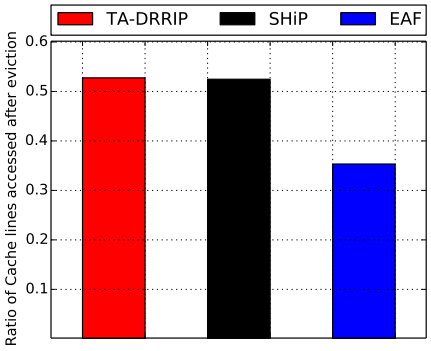
We assume *four* applications share a *4-way associative* cache. Let $A : \{a1, a2, a3, a4\}^{k1}, B : \{b1, b2, b3\}^{k2}$, $C : \{c1, c2\}^{k3}$ and $D : \{d1, d2, d3, d4, d5, d6\}^{k4}$ be the sequences of accesses to cache blocks by applications A, B, C and D, respectively, during a certain interval of time. Let k1=3, k2=1, k3=3 and k4=4 denote the number of reuses to the given access sequence. Assuming a fair scheduling policy, we have the following combined accesses sequence in the order : S1 :{a1,b1,c1,d1}, S2 :{a2,b2,c2,d2}, S3 :{a3,b3,c1,d3}, S4 :{a4,-,c2,d4} etc. Also, let us assume all accesses update their RRPV on hits.

Fig. 2a shows our example. The boxes represent cache tag storing the cache block address and the number below each box represents the block's RRPV. TA-DRRIP inserts cache lines of applications A, B and C with RRPV 2 and cache lines of application D with RRPV 3. On the other hand, ADAPT inserts the cache lines with *discrete* priorities[3]. Cache lines of applications A and D are inserted with RRPV 3 and, cache lines of application B and C are inserted with RRPV 1 and 0, respectively. From the figure, we see cache line c1, which is inserted with RRPV 0 is able to survive until its next use. However, TA-DRRIP is not able to preserve cache block c1 of C, which is inserted with RRPV 2, until its next use. From experiments, we observe that significant fraction of cache lines suffer from such early evictions. Fig. 2b shows that close to 52% of reuses (that miss) in the cache fall within the first 256 misses in the set under TA-DRRIP and SHiP algorithms, while close to 36% of reuses fall within the first 256 misses under Evicted Address Filter (EAF) [2] algorithm.

[2]ADAPT presented in this example is only for illustrative purpose. The next section describes our actual design.
[3]Assignment of priorities by ADAPT is the subject of next section.




(a) TA-DRRIP vs ADAPT (discrete prioritization).




(b) Ratio of accesses that fall within the first 256 misses in the set after their eviction.

Fig. 2: a) Benefit of discrete prioritization. b) Ratio of Early Evictions.

# 3 Adaptive Discrete and de-prioritized Application Priori-Tization

Adaptive Discrete and de-prioritized Application PrioriTization, ADAPT, consists of two components: (i) the monitoring mechanism and (ii) the insertion-priority algorithm. The first component monitors the cache accesses (block addresses) of each application and computes its Footprint-number, while the second component infers the insertion priority for the cache lines of an application using its *Footprint-number*. Firstly, we describe the design, operation and cost of the monitoring mechanism. Then, we describe in detail the insertion-priority algorithm.

## 3.1 Collecting Footprint-number

**Definition:** Footprint-number of an application is the number of unique accesses (block addresses) that it generates to a cache set. However, during execution, an application may exhibit change in its behavior and hence, we define its *Sliding Footprint-number*[4] as the number of unique accesses it generates to a set in *an interval of time*. We define this interval in terms of the number of misses at the shared last level cache since only misses trigger *new* cache blocks to be allocated. However, sizing of this interval is critical since the combined misses of all the applications at the shared cache could influence their individual (sliding) Footprint-number values. A sufficiently large interval mitigates this effect on Footprint-number values. To fix the interval size, we perform experiments with 0.25M, 0.5M, 1M, 2M and 4M interval sizes. Among, 0.25, 0.5 and 1M misses, 1M gives the best results. And, we do not observe any significant difference in performance between 1M and 4M interval sizes. Further, 1 Million misses on average correspond to 64K misses per application and

[4]However, we just use the term Footprint-number throughout.

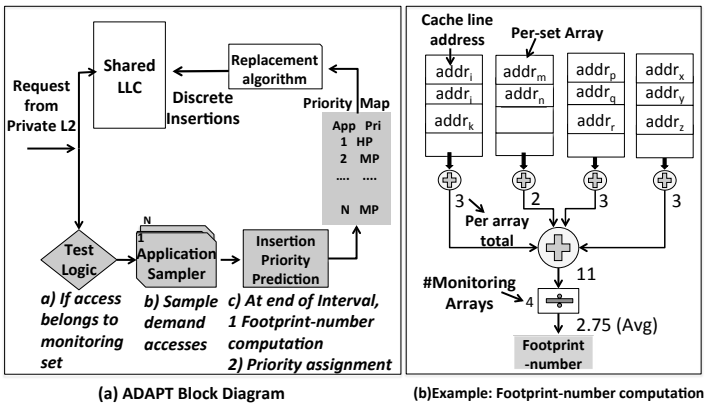



Fig. 3: (a) ADAPT Block Diagram and b) example for Footprint-number computation.

are roughly *four* times the total number of blocks in the cache, which is sufficiently large. Hence, we fix the interval size as 1M last level cache misses.

Another point to note is that Footprint-number can only be computed approximately because (i) cache accesses of an application are not uniformly distributed across cache sets. (ii) Tracking all cache sets is impractical. However, a prior study [6] has shown that the cache behavior of an application can be approximated by sampling a small number of cache sets (as few as 32 sets is enough). We use the same idea of sampling cache sets to approximate Footprint-number. From experiments, we observe that sampling 40 sets are sufficient.

**Design and Operation:** Fig. 3a shows the block diagram of a cache implementing ADAPT replacement algorithm. In the figure, the blocks shaded with gray are the additional components required by ADAPT. The *test logic* checks if the access (block address) belongs to a monitored set and if it is a demand access[5], and then it passes the access to the application sampler. The *application sampler* samples cache accesses (block addresses) directed to each monitored set. There is a storage structure and a saturating counter associated with each monitored set. The storage structure is essentially an array which operates like a typical tag-array of a cache set.

First, the cache block address is searched. If the access does not hit, it means that the cache block is a *unique* access. It is added into the array and the counter, which indicates the number of unique cache blocks accessed in that set, is incremented.On a hit, only the recency bits are set to 0. Any policy can be used to manage replacements. We use SRRIP policy. All these operations lie outside the critical path and are independent of the hit/miss activities on the main cache. Finally, it does not require any change to the cache tag array except changing the insertion priority.

**Example:** Fig. 3b shows an example of computing Footprint-number. For simplicity, let us assume we sample 4 cache sets and a single application. In the diagram, each array belongs to a separate monitored set. An entry in the array corresponds to the block address that accessed the set. We approximate Footprint-number by computing the average from all the sampled sets. In this example, the sum of all the entries from all the *four* arrays is 11. And, the average is 2.75. This is the Footprint-number for the application. In a multi-core system, there are as many instances of this component as the number of applications in the system.

[5]Only demand accesses update the recency state.

### 3.2 Footprint-number based Priority assignment

Like prior studies [1, 5, 2, 38], we use 2 bits per cache line to represent re-reference prediction value (RRPV). RRPV '3' indicates the line will be reused in the distant future and hence, a cache line with RRPV of 3 is a candidate for eviction. On hits, only the cache line that hits is promoted to RRPV 0, indicating that it will be reused immediately. On insertions, unlike prior studies, we explore the option of assigning different priorities (up to 4) for applications leveraging the Footprint-number metric.

We propose an *insertion-priority-prediction* algorithm that statically assigns priorities based on the Footprint-number values. Footprint-number ranges ideally depend on the number of applications that share the cache and their access characteristics, associativity of the last level cache, and the interval size. Therefore, a dynamic approach that takes these parameters into account is desirable[6]. However, in this work, we empirically study the footprint-number ranges assuming a 16-core system with each core running *one* application and having a shared last level cache of 16-way associativity. To fix the Footprint-number ranges, by fixing the low-priority range fixed to (8,16), we vary the high-priority range between [0,3] and [0,8] (6 different ranges). Similarly, by fixing the high-priority range to [0,3], we vary the low-priority range between (7,16) to (12,16) (6 different ranges). In total, from 36 different experiments we fix the priority-ranges. From the experimentally fixed ranges, we assign priorities as follows:

**High Priority:** All applications in the Footprint-number range [0,3] (both included) are assigned high-priority. When the cache lines of these applications miss, they are inserted with RRPV 0.

*Intuition:*Applications in this category have working sets that fit perfectly within the cache. Typically, the cache lines of these applications have high number of reuses. Also, when they share the cache, they do not pose problems to the co-running applications. Hence, they are given high-priority. Inserting with priority 0 allows the cache lines of these applications to stay in the cache for longer periods of time before being evicted.

**Medium Priority:** All applications in the Footprint-number range (3,12] (3 excluded and 12 included) are assigned medium priority. Cache lines of the applications in this category are inserted with value 1 and rarely inserted with value 2.

*Intuition:* Applications under this range of Footprint-number have working set larger than the high-priority category however, fit within the cache. From analysis, we observe that the cache lines of these applications generally have moderate reuse except few applications. To balance mixed reuse behavior, one out of the sixteenth insertion goes to low priority 2, while inserted with medium priority 1, otherwise.

**Low Priority:** Applications in the Footprint-number range (12,16) are assigned low priority. Cache lines of these applications are generally inserted with RRPV 2 and rarely with medium priority 1 (1 out of 16 cache lines).

*Intuition:* Applications in this category typically have mixed access patterns : $(\{a1, a2\}^k$ $\{s1, s2, s3..sn\}^d)$ with $k$ and $d$ sufficiently small and $k$ slightly greater than $d$, as observed by TA-DRRIP [1]. Inserting the cache lines of these applications with *low* priority 2 ensures (i) cache lines exhibiting low or no reuse at all get filtered out quickly and (ii) cache lines of these applications have higher probability of getting evicted than high and medium priority applications[7].

[6] We defer this dynamic approach to future work.

[7] It means that transition from 2 to 3 happens quicker than 0 to 3 or 1 to 3 thereby allowing HP and MP applications to stay longer in the cache than LP applications.

Table 1: Insertion Priority Summary

| **Priority Level** | **Insertion Value** |
|---|---|
| High (HP) | 0 |
| Medium (MP) | 1 but 1/16th insertion at LP |
| Low (LP) | 2 but 1/16th at MP |
| Least (LstP) | Bypass but insert 1/32nd at LP |

Table 2: Cost on 16MB,16-way LLC

| **Policy** | **Storage cost** | **N=24 cores** |
|---|---|---|
| TA-DRRIP | 16-bit/app | 48 Bytes |
| EAF-RRIP | 8-bit/address | 256KB |
| SHiP | SHCT table&PC | 65.875KB |
| ADAPT | 865 Bytes/app | 24KB approx. |

**Least Priority:** Applications with Footprint-number range (>= 16) are assigned least priority. Only one out of thirty-two accesses are installed at the last level cache with least priority 3. Otherwise, they are bypassed to the private Level 2. *Intuition:* Essentially, these are applications that either exactly fit in the cache (occupying all *sixteen* ways) or with working sets larger than the cache. These applications are typically memory-intensive and when run along with others cause thrashing in the cache. Hence, both these types of applications are candidates for least priority assignment. The intuition behind bypassing is that when the cache lines inserted with least priority are intended to be evicted very soon (potentially without reuse), bypassing these cache lines will allow the incumbent cache lines to better utilize the cache. Our experiments confirm this assumption. In fact, bypassing is not just beneficial to ADAPT. It can be used as a performance booster for other algorithms, as we show in the evaluation section.

## 3.3 Hardware Overhead

The additional hardware required by our algorithm is the application sampler and insertion priority prediction logic. The application sampler consists of an array and a counter. The size of the array is same as the associativity. From Section 3.2, recall that we assign the same priority(least) to applications that exactly fit in the cache as well as the thrashing applications because, on a *16-way* associative cache, both classes of applications will occupy a minimum of 16 ways. Hence, tracking 16 (tag) addresses per set is sufficient. The search and insertion operations on the array are very similar to that of a cache set. The difference is that we store only the *most significant 10 bits* per cache block. **Explanation:** the probability of two different cache lines having all the 10 bits same is very low: $(1/2^x)/(2^{10}/2^x)$, where $x$ is the number of tag bits. That is, $1/2^{10}$. Even so, there are separate arrays for each monitoring set. Plus, applications do not share the arrays. Hence, 10 bits are sufficient to store the tag address. 2 bits per entry are used for bookkeeping. Additionally, 8 bits are required for head and tail pointers (4 bits each) to manage search and insertions. Finally, a 4-bit counter is used to represent Footprint-number.

Storage overhead per set is 204 bits and we sample 40 sets. Totally, 204 bits $\times 40 = 8160 bits$. To represent an application's Footprint-number and priority, two more bytes (1 byte each) are needed.

To support probabilistic insertions, three more counters each of size one byte are required. Therefore, storage requirement per application sampler is $[8160\ bits + 40\ bits] = 8200 bits/application$. In other words, 1KB (approximately) per application.

Table 2 compares the hardware cost of ADAPT with others. Though ADAPT requires more storage compared to TA-DRRIP [1], it provides higher performance improvement and is better compared to EAF [2] and SHiP [5] in both storage and performance.

### 3.4 Monitoring in a realistic system:

In the paper, we assume that one thread per core. Therefore, we can use the core ID for the thread. On an SMT machine, the thread number/ID would have to be transmitted with the request from the core for our scheme to work properly.

If an application migrates (on a context-switch) to another core, the replacement policy applied for that application during the next interval will be incorrect. However, the interval is not long (1Million LLC misses). The correct Footprint-number and insertion policy will be re-established in the following monitoring interval onward. In data-centers or server systems, tasks or applications are not expected to migrate often. A task migrates only in exceptional cases like shutdown or, any power/performance related optimization. In other words, applications execute(spend) sufficient time on a core for the heuristics to be implemented. Finally, like prior works [4, 3, 1, 2, 5, 6], we target systems in which LLC is organized as multiple banks with uniform access latency.

With multi-threaded applications there are two interesting scenarios based on how threads interfere (constructive or destructive) with each other. If threads constructively interfere, it would be better to aggregate them as a single application since they usually operate on the same working set doing similar work. In this case, replacement decisions could be augmented with the sharing characteristics of data [43, 42] among threads. On the other hand, if they destructively interfere, it would be better to treat each thread as an independent application.

## 4 Experimental Study

### 4.1 Methodology

For our study, we use BADCO [19] cycle-accurate x86 CMP simulator. Table 3 shows our baseline system configuration. We do not enforce inclusion in our cache-hierarchy and all our caches are write-back caches. LLC is 16MB and organized into 4 banks. We model bank-conflicts, but with fixed latency for all banks like prior studies [1, 5, 2]. A Virtual Private Cache [7] based arbiter schedules requests from L2 to LLC. Our DRAM model is similar to the one in Evicted Address Filter [2].

### 4.2 Benchmarks

We use benchmarks from SPEC 2000 and 2006 and PARSEC benchmark suites, totaling 36 benchmarks (31 from SPEC and 4 from PARSEC and 1 Stream benchmark). Table 5 shows the classification of all the benchmarks and Table 4 shows the empirical method used to classify memory intensity of a benchmark based on its Footprint-number and L2-MPKI when run alone on a 16MB, 16-way set-associative cache. In Table 5, the column Fpn(A) represents Footprint-number value

Table 3: Baseline System Configuration

| | |
|---|---|
| Processor Model | 4-way OoO, 128 entry ROB, 36 RS, 36-24 entries LD-ST queue |
| Branch pred. | TAGE, 16-entry RAS |
| IL1 & DL1 | 32KB; LRU; next-line prefetch;I$:2-way;D$:8-way; 64 bytes line |
| L2 (unified) | 256KB,16-way, 64 bytes line, DRRIP, 14-cycles, 32-entry MSHR and 32-entry retire-at-24 WB buffer |
| LLC (unified) | 16MB, 16-way, 64 bytes line, TA-DRRIP, 24 cycles, 256-entry MSHR and 128-entry retire-at-96 WB buffer |
| Main-Memory (DDR2) | Row-Hit:180 cycles, Row-Conflict:340 cycles, 8 banks,4KB row, XOR-mapped [27] |

Table 4: Empirical Classification Methodology

| **FP-num** | **L2 MPKI** | **Memory Intensity** |
|---|---|---|
| $< 16$ | $< 1$ | VeryLow (VL) |
| | $[1, 5)$ | Low (L) |
| | $> 5$ | Medium (M) |
| **FP-num** | **L2 MPKI** | **Memory Intensity** |
| $>= 16$ | $< 5$ | Medium (M) |
| | $[5, 25)$ | High (H) |
| | $> 25$ | VeryHigh (VH) |

Table 5: Benchmark classification based on Footprint-number and L2-MPKI.

| **Name** | **Fpn(A)** | **Fpn(S)** | **L2-MPKI** | **Type** | **Name** | **Fpn(A)** | **Fpn(S)** | **L2-MPKI** | **Type** |
|---|---|---|---|---|---|---|---|---|---|
| black | 7 | 6.9 | 0.67 | VL | bzip | 4.15 | 4.03 | 25.25 | M |
| calc | 1.33 | 1.44 | 0.05 | VL | gap | 23.12 | 23.35 | 1.28 | M |
| craf | 2.2 | 2.4 | 0.61 | VL | gob | 16.8 | 16.2 | 1.28 | M |
| deal | 2.48 | 2.93 | 0.5 | VL | hmm | 7.15 | 6.82 | 2.75 | M |
| eon | 1.2 | 1.2 | 0.02 | VL | lesl | 6.7 | 6.3 | 20.92 | M |
| fmine | 6.18 | 6.12 | 0.34 | VL | mcf | 11.9 | 12.4 | 24.9 | M |
| h26 | 2.35 | 2.53 | 0.13 | VL | omn | 4.8 | 4 | 6.46 | M |
| nam | 2.02 | 2.11 | 0.09 | VL | sopl | 10.6 | 11 | 6.17 | M |
| sphnx | 5.2 | 5.4 | 0.35 | VL | twolf | 1.7 | 1.6 | 16.5 | M |
| tont | 1.6 | 1.5 | 0.75 | VL | wup | 24.2 | 24.5 | 1.34 | M |
| swapt | 1 | 1 | 0.06 | VL | apsi | 32 | 32 | 10.58 | H |
| gcc | 3.4 | 3.2 | 1.34 | L | astar | 32 | 32 | 4.44 | H |
| mesa | 8.61 | 8.41 | 1.2 | L | gzip | 32 | 32 | 8.18 | H |
| pben | 11.2 | 10.8 | 2.34 | L | libq | 29.7 | 29.6 | 15.11 | H |
| vort | 8.4 | 8.6 | 1.45 | L | milc | 31.42 | 30.98 | 22.31 | H |
| vpr | 13.7 | 14.7 | 1.53 | L | wrf | 32 | 32 | 6.6 | H |
| fsim | 10.2 | 9.6 | 1.5 | L | cact | 32 | 32 | 42.11 | VH |
| sclust | 8.7 | 8.4 | 1.75 | L | lbm | 32 | 32 | 48.46 | VH |
| art | 3.39 | 2.31 | 26.67 | M | STRM | 32 | 32 | 26.18 | VH |

Table 6: Workload Design.

| Study | #Workloads | Composition | #Instructions |
|---|---|---|---|
| 4-core | 120 | Min 1 thrashing | 1.2B |
| 8-core | 80 | Min 1 from each class | 2.4B |
| 16-core | 60 | Min 2 from each class | 4.8B |
| 20-core | 40 | Min 3 from each class | 6B |
| 24-core | 40 | Min 3 from each class | 7.2B |

obtained by using all sets while the column Fpn(S) denotes Footprint-number computed by sampling. Only *vpr* shows $> 1$ difference in Footprint-number values. Only to report the upper-bound on the Footprint-numbers, we use 32-entry storage. In our study, we use only 16-entry array. We use a selective portion of 500M instructions from each benchmark. We warm-up all hardware structures during the first 200M instructions and simulate the next 300M instructions. If an application finishes execution, it is re-executed until all applications finish.

### 4.3 Workload Design

Table 6 summarizes our workloads. For 4 and 8-core workloads, we study with 4MB and 8MB shared caches while 16, 20 and 24-core workloads are studied with a 16MB cache since we target caches where $\#applications \geq \#llcassociativity$. In all these studies, the last level cache associativity is kept at sixteen.

## 5 Results and Analysis

### 5.1 Performance on 16-core workloads

Fig. 4 shows performance on the weighted-speedup metric over the baseline TA-DRRIP and three other state-of-the-art cache replacement algorithms. We evaluate two versions of ADAPT: one which inserts all cache lines of least priority applications (referred as *ADAPT_ins*) and the version which mostly *bypasses* the cache lines of least priority applications (referred as *ADAPT_bp32*). Our best performing version is the one that bypasses the cache lines of thrashing applications. Throughout our discussion, we refer to ADAPT as the policy that implements bypassing. From Fig. 4, we observe that ADAPT consistently outperforms other cache replacement policies. It achieves up to 7% improvement with 4.7% on average. As mentioned in Section 2, with set-dueling, applications with working-set larger than the cache, implement SRRIP policy, which causes higher contention and thrashing in the cache. Similarly, SHiP learns the reuse behavior of region of cache lines (grouped by their PCs) depending on the hit/miss behavior. A counter records the hits (indicating *intermediate*) and misses (indicating *distant*) reuse behavior for the region of cache lines. Since SHiP implements SRRIP, it observes similar hit/miss pattern as TA-DRRIP for thrashing applications. Consequently, like TA-DRRIP, it implements SRRIP for all applications. Only 3% of the misses are predicted to have *distant* reuse behavior. The marginal drop in performance (1.1%approx.) is due to *inaccurate* distant predictions on certain cache-friendly applications. Overall, ADAPT uses *Footprint-number* metric to efficiently distinguish across applications.

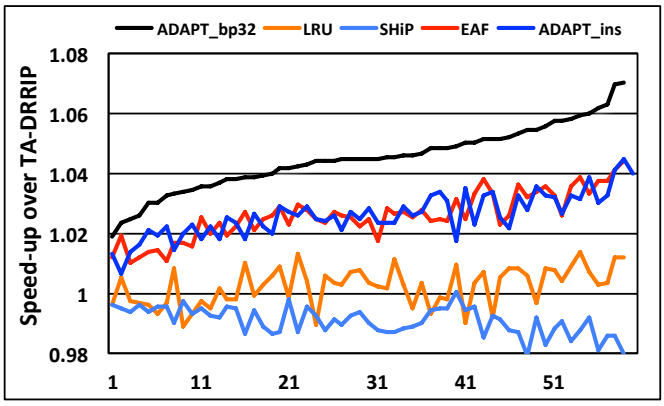



Fig. 4: Performance of 16-core workloads.

LRU inserts the cache lines of all applications at MRU position. However, cache-friendly applications only *partially* exploit such longer most-to-least transition time because the MRU insertions of thrashing applications pollute the cache. On the other hand, ADAPT efficiently distinguishes applications. It assigns least priority to thrashing applications and effectively filter out their cache lines, while inserting recency-friendly applications with higher priorities, thus achieving higher performance.

The EAF algorithm filters recently evicted cache addresses. On a cache miss, if the missing cache line is present in the filter, the cache line is inserted with *near-immediate* reuse (RRPV 2). Otherwise, it is inserted with *distant* reuse (RRPV 3). In EAF, the size of the filter is such that it is able to track as many misses as the number of blocks in the cache (that is, working-set twice the cache). Hence, any cache line that is inadvertently evicted from the cache falls in this filter and gets *intermediate* reuse prediction. Thus, EAF achieves higher performance compared to TA-DRRIP, LRU and SHiP. Interestingly, EAF achieves performance comparable to ADAPT_ins. On certain workloads, it achieves higher performance while on certain workloads it achieves lesser performance. This is because, with ADAPT (in general), applications with smaller Footprint-number are inserted with RRPV 0 or 1. However, when such applications have poor reuse, EAF (which inserts with RRPV 2 for such applications) filters out those cache lines. On the contrary, applications with smaller Footprint-number but moderate or more number of reuses, gain from ADAPT's discrete insertions. Nevertheless, ADAPT (with bypassing) consistently outperforms EAF algorithm. We observe that the presence of thrashing applications causes the filter to get full frequently. As a result EAF is only able to partially track the application's (cache lines). On the one hand, some cache lines of non thrashing (recency-friendly) that spill out of the filter get assigned a *distant* (RRPV 3). On the other hand, cache lines of the thrashing applications that occupy filter positions get *intermediate* (RRPV 2) assignment.

## 5.2 Impact on Individual Application Performance

We discuss the impact of ADAPT on individual application's performance. The results are averaged from all the sixty 16-core workloads. Only applications with change (>= 3%) in MPKI or IPC are reported. From Fig. 5 & Fig. 6, we observe that discretely prioritizing (ADAPT) significantly

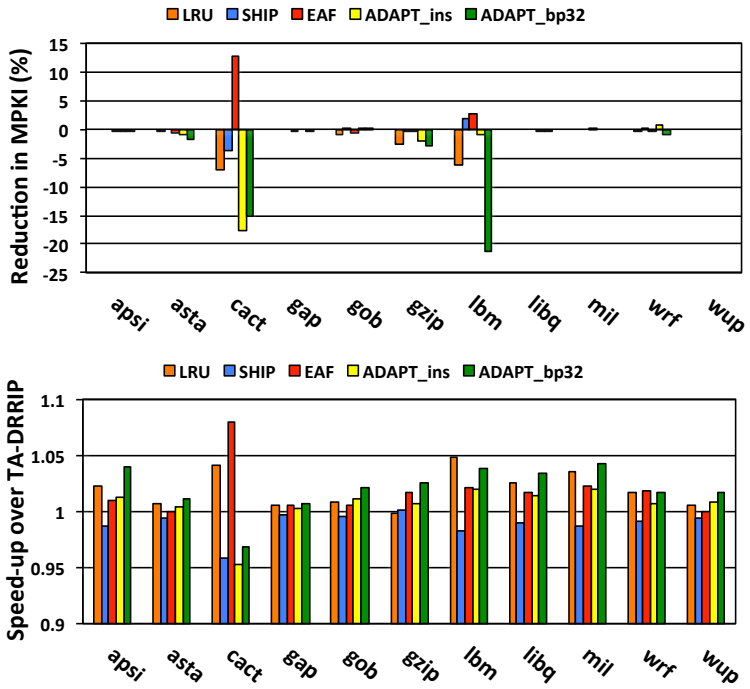



Fig. 5: MPKI(top) and IPC(below) of thrashing applications.

improves the performance of non-thrashing applications (which are assigned high and medium priorities by ADAPT). The idea of bypassing further increases the performance of many non-thrashing applications. This is because, as we bypass the cache lines (31/32 times) of the least-priority applications (instead of inserting), the cache state is *not disturbed* most of the times: cache lines which could benefit from staying in the cache remain longer in the cache without being removed by cache lines of the thrashing applications. ADAPT affects only *cactusADM* because some of its cache blocks are reused immediately after insertion. Since ADAPT assigns least priority, such cache blocks are evicted early under ADAPT. However, for *cactusADM* the bypass version of ADAPT performs better than the insertion version for the same reason. For *gzip and lbm*, though MPKI increases, they do not suffer slow-down in IPC. Because, an already memory-intensive application with high memory-related stall time, which when further delayed, does not experience much slow-down [20].

## 5.3 Impact of Bypassing on cache replacement policies

In this section, we show the impact of bypassing distant priority cache lines instead of inserting them on all replacement policies. Since LRU policy inserts all cache lines with MRU (high) priority, there is no opportunity to implement bypassing. From Figure 7, we observe that bypassing achieves higher performance for replacement policies except SHiP. As mentioned earlier, SHiP predicts *distant* reuse only for 3% of the cache lines. Of them, 69% (on average) are miss-predictions. Hence, there is

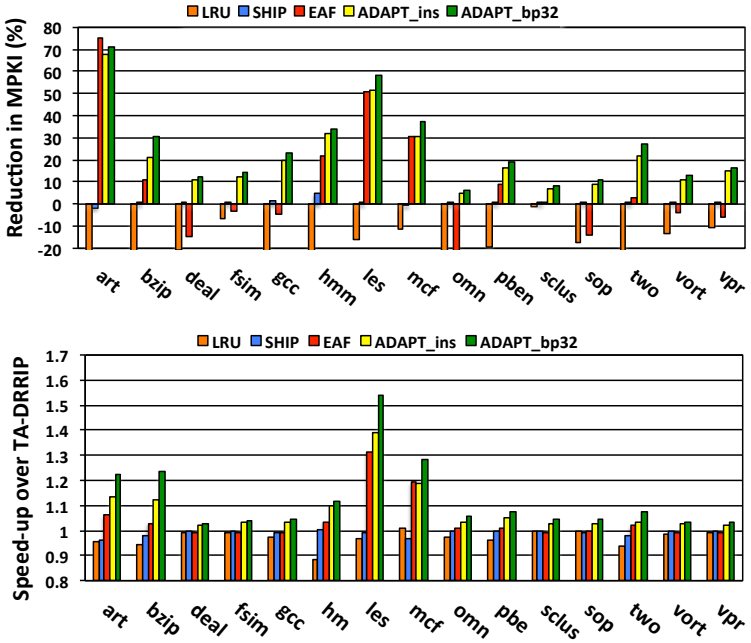



Fig. 6: MPKI(top) and IPC(below) of non-thrashing applications.

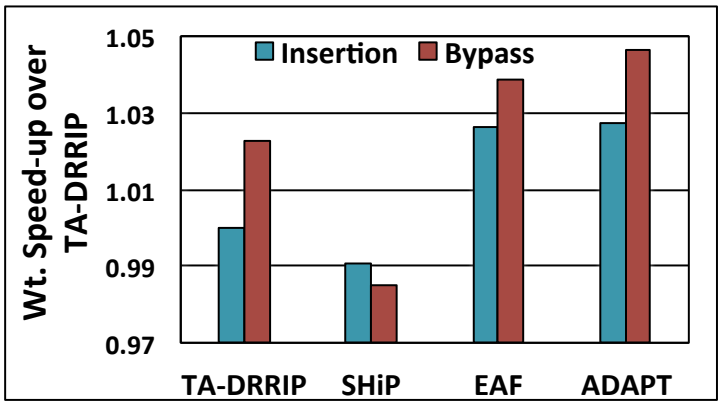



Fig. 7: Impact of Bypassing on replacement policies.

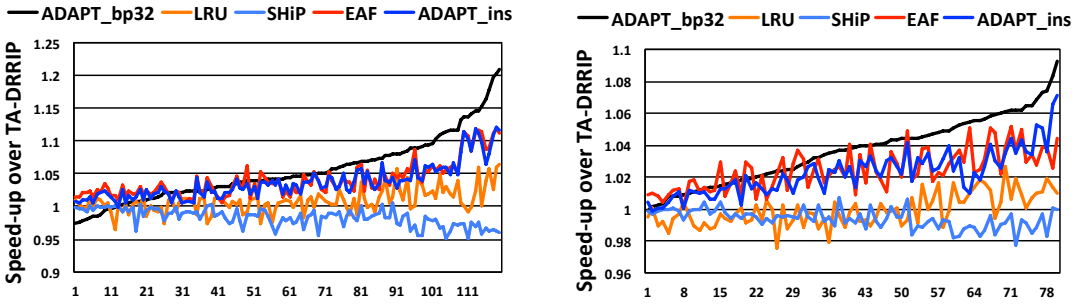
minor drop in performance.

On the contrary, TA-DRRIP, which implements *bi-modal* (BRRIP) on certain cache sets, bypasses the *distant* priority insertions directly to the private L2 cache, which is beneficial. Consequently, it learns BRRIP for the thrashing applications. Similarly, EAF with bypassing achieves higher performance. EAF, on average, inserts 93% of its cache lines with *distant* reuse prediction providing more opportunities to bypass. However, we observe that 33% (approx.) of *distant* reuse predictions

are incorrect[8]. Overall, from Fig. 7, we can make two conclusions: first, our intuition of bypassing *distant* reuse cache lines can be applied to other replacement policies. Second, Footprint-number is a reliable metric to approximate an application's behavior: using Footprint-number, ADAPT distinguishes thrashing applications and bypasses their cache lines. Results presented in this figure is normalized from all the 60 16-core workloads.

## 5.4 Scalability with respect to number of applications

In this section, we study how well ADAPT scales with respect to the number of cores sharing the cache. Fig. 8 and Fig. 9 show the s-curves of weighted speed-up for 4,8,20 and 24-core workloads. ADAPT outperforms prior cache replacement policies. For 4-core workloads, ADAPT yields average performance improvement of 4.8%, and 3.5% for 8-core workloads. 20 and 24-core workloads achieve 5.8% and 5.9% improvement, on average, respectively. We study 4-core and 8-core workloads with 4MB and 8MB caches, respectively, with 16-way associativity. For 20 and 24-core workloads, we study 16MB, 16-way associative cache. Recall our proposition : ($\#applications \geq associativity$).




(a) 4-core (120 workloads)

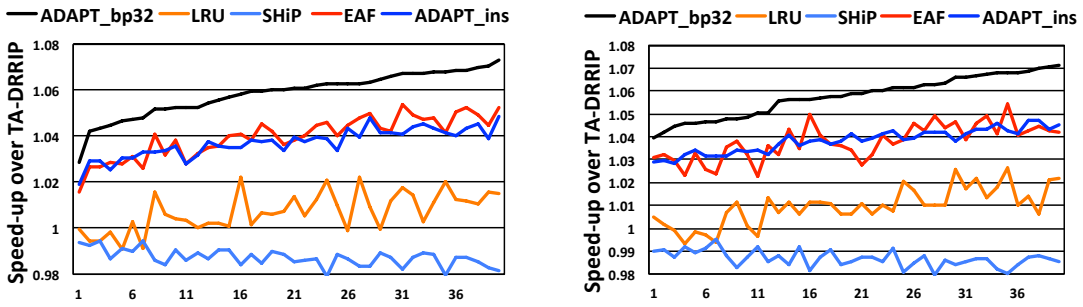
(b) 8-core (80 workloads)

Fig. 8: Performance of ADAPT with respect to number of applications for 4 and 8-cores.

## 5.5 Sensitivity to Cache Configurations

In this section, we study the impact of ADAPT replacement policy on systems with larger last level caches. In particular, the goal is to study if Footprint-number based priority assignment designed for 16-way associative caches applies to larger associative ($> 16$) caches as well. For 24MB and 32MB caches, we increase only the associativity of the cache set from 16 to 24 and 16 to 32, respectively. Certain applications still exhibit thrashing behaviors even with larger cache sizes which ADAPT is able to manage and achieve higher performance on the weighted Speed-up metric (Fig. 10).

[8]Miss-predictions are accounted by tracking distant priority (RRPV 3) insertions which are not reused while staying in the cache, but referenced (within a window of 256 misses per set) after eviction. Here, we do not account distant priority insertions that are reused while staying in the cache because such miss-predictions do not cause penalty.




(a) 20-core (40 workloads) (b) 24-core (40 workloads)

Fig. 9: Performance of ADAPT with respect to number of applications for 20 and 24-cores.

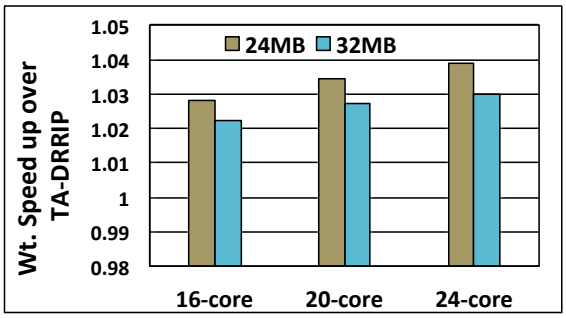



Fig. 10: Performance on Larger Caches.

# 6 Related Work

Cache management techniques can be broadly categorized as cache replacement and cache partitioning techniques. Numerous studies have been proposed in the past under each category but most of the studies have focused only on small scale multi-core processors. In this article, we address cache management in many-core processors, where the number of applications sharing the cache is larger than the associativity of the shared last level cache. We summarize some of the prior studies.

## 6.1 Insertion priority prediction

Adapive and Dynamic Insertion Policy (DIP) [4] is one of the foremost proposals (i) to view cache replacement as separate steps involving evicting a cache line and inserting the missing cache line and (ii) to alter the *insertion priority* of cache lines. In particular, they observe that the LRU policy thrashes for applications with working set size larger than the cache. Typically, the LRU policy evicts the *least recently used* cache line and inserts the missing cache line with the *most recently used* priority. DIP observes that for thrashing applications, retaining only a fraction of the working set (instead of the whole set) avoids thrashing. The insertion policy is altered to achieve this effect. The inserted cache lines are updated to the MRU priority only probabilistically (1/32

times). In all other cases, the priority of the cache lines remain at LRU, eventually getting filtered out on subsequent misses and avoiding thrashing. The variant policy is referred to as *Bi-modal insertion policy* (BIP). However, for *recency-friendly* applications, LRU is still the best policy. They porpose *set-dueling* to dynamically learn the best policy for a given application. TADIP [3] proposes a thread-aware methodology to dynamically learn the insertion priorities in a shared cache environment.

Dynamic Re-reference Interval Prediction (DRRIP) [1] proposes to predict the reuse behavior of cache lines as re-reference interval buckets. RRIP consists of *SRRIP* and *BRRIP* policies to manage different workload patterns. While SRRIP handles mixed (recency-friendly pattern mixing with scan) and scan type of access patterns, BRRIP handles thrashing patterns. As with DIP, RRIP uses set-dueling to dynamically learn the best of the two polices for an application. It uses the same methodology [3] to manage applications in shared cache environment.

While DRRIP relies on set-dueling (number of misses incurred by the competing policies) to learn the best insertion policy for an application, Signature-based Hit Predictor (SHiP) [5] and Evicted Address Filter (EAF) [2] add intelligence to the predictions made during cache line insertions. SHiP uses Program-counter, Instruction sequence and Memory region signatures (each is a separate mechanism) to predict different priorities (SRRIP or BRRIP) for regions of accesses corresponding to the signature. They observe that PC based approach yields the best results. EAF further enhances the prediction granularity to individual cache lines. A filter decides the SRRIP/BRRIP priority of cache lines based on its presence/absence in the filter. The goal is to avoid premature evictions from the cache.

All these approaches use only binary (SRRIP or BRRIP) insertion policies. Moreover, as discussed in the motivation section, they cannot be adapted to enable discrete prioritization. On the contrary, ADAPT is able to classify applications into discrete priority buckets and achieve higher performance. SHiP and EAF predict priorities at the granularity of individual or regions of cache lines and appear as finer classification mechanisms[9]. However, in commercial designs [37][40], which use a software-hardware co-designed approach to resource management, the system software decides fairness or performance objectives only at an application granularity. Hence, it is desirable that the cache management also performs application level performance optimizations.

## 6.2 Reuse distance prediction

The techniques discussed above make only *qualitative* estimate (predictions) on the reuse behavior of cache lines and classify them as to have either *intermediate* or *distant* reuse. ADAPT is one such technique. However, some studies [41, 32, 33, 44, 35, 36, 34] explicitly compute the reuse distance value of cache lines at run-time and perform replacements using the explicit value of reuse distance. Since the reuse distances of cache lines can take wider range of values, measuring reuse distance at run-time is typically complex, requires significant storage and modifies the cache tag arrays to store reuse distance values for cache lines. Here, we describe some of the important works in this domain.

Inter-reference Gap Distribution Replacement (IGDR) [41] proposed by Tagaki et al. predicts the inter reference gap of cache lines and assigns them as weights. Their work is based on the observation that the inter reference gap distribution of a cache block takes only few discrete values and that cache blocks with same re-reference counts have same inter reference gap distribution.

[9]Finer classification we mention here should not to be confused with discrete classification that we propose. We refer to discrete as having ($> 2$) priorities across applications.

Thus, cache blocks are grouped (classified) based on the number of re-references. Essentially, classification of blocks signifies different inter reference gaps (or, priorities). During its lifetime, a cache block moves from one class to another before being evicted. Prior works [32, 44] capture the reuse distance of cache blocks using the PCs that access them. They observe only few PCs to contribute to the most of the cache accesses. However, these techniques apply to single-thread context. Since reuse distance computation for all cache blocks incurs significant overhead, some studies have proposed to sample cache accesses and compute the reuse distance for select cache blocks [35, 36, 34].

Schuff et al. [36] observe that in multi-threaded environments, the timing of interactions between the threads does not affect stack distance computation. Hence, stack distance can be computed in parallel for all threads. They use sampling to track the reuse distance of individual threads. In their approach, the stack distance of cache lines receiving invalidation (on coherence update) from another thread is approximated to have very distant reuse (maximum reuse distance value). This is because such a cache line would receive a coherence miss anyway. This assumption may not be optimal in certain cases. For example, assume certain threads share a cache block and frequently access/update the cache block. Threads which read the updated block gets a coherence miss and subsequently, gets the valid cache block from the other thread. Forcing a *distant* reuse on such cache blocks could make the replacement policy to inadvertently give low priority and cause early evictions resulting in frequent off-chip accesses. Further, their approach may be counter-productive when combined with any on-chip cache management technique as in [15].

NUCache [31] propose a novel cache organization that builds on the idea of delinquent PCs. Cache is logically partitioned as main-ways and deli-ways. The idea is to store the cache lines (of delinquent PCs) evicted from the main-ways into deli-ways and retain the cache lines for duration beyond their eviction. The drawbacks with their approach is that caches need to have larger associativity, which adds significant energy overhead. Secondly, when there are large number of applications sharing the cache, finding the optimal set of delinquent PCs across all applications and assign deli-ways among them becomes complex.

Duong et al. [34] propose the *protecting distance* (PD) metric to protect cache lines until certain number of accesses. Also, they propose a hit-rate model which dynamically checks if inserting a cache line would improve the hit-rate. If not, the cache line is bypassed. They extend the hit-rate model to decide per-thread PD that maximizes the hit-rate of the shared cache. Computing protecting distance is quite complex and incurs significant hardware overhead in terms of logic and storage. For large number of applications, computing optimal PDs may require searching across a large reuse distance space. Conversely, ADAPT requires only tracking a limited number of accesses (sixteen) per set and simple logic to compute Footprint-number.

## 6.3 Eviction priority prediction

Victim selection techniques try to predict cache lines that are either *dead* or very unlikely to be re-used soon [21, 22, 23, 24]. A recent proposal, *application-aware cache replacement* [24] predicts cache lines with very long re-use distance using hit-gap counters. Hit-gap is defined as the number of accesses to a set between two hits to the same cache line. Precisely, the hit-gap gives the maximum duration for which the cache line should stay in the cache. On replacements, a cache line residing closer to/beyond this hit-gap value is evicted. In many-cores, under their approach, certain recency-friendly applications could get hidden behind memory-intensive applications and would suffer more misses. However, ADAPT would be able to classify such applications and retain

their cache lines for longer time. Further, this mechanism requires expensive look-up operations and significant modifications to the cache tag array.

## 6.4 Cache Bypassing

Bypassing cache lines was proposed in many studies [13, 14, 18, 17, 12, 16, 15]. Run-time Cache Bypassing [14] proposes to bypass cache lines with low-reuse behaviors while few others try to address conflict misses by bypassing cache lines that could pollute the cache. All these techniques either completely bypass or insert all requests. For thrashing applications, retaining a fraction of the working set is beneficial to the application [4]. However, in many-core caches, such an approach is not completely beneficial. Inserting cache lines of thrashing applications with least-priority still pollutes the cache. Instead, bypassing most of their cache lines is beneficial both to the thrashing application as well as the overall performance. As we showed in the evaluation section, bypassing least-priority cache lines is beneficial to other replacement policies as well.

Segmented-LRU [12] proposes probabilistic bypassing of cache lines. The *tag* of the bypassed cache line and the tag of the *victim* cache line (which is actually not evicted) are each held in separate registers. If an access to the virtual victim cache line is found to occur ahead of the bypassed cache line, the bypass is evaluated to be useful. This mechanism functions well in single-core context and small-scale multi-cores. However, in many-cores, as demonstrated in the motivation section, observing the hits and misses on the shared cache is not an efficient way to decide on policies as they may lead to incorrect decisions. On the contrary, ADAPT decides to bypass cache lines based on Footprint-number of applications which do suffer from the actual activity of the shared cache. Gaur et al. [16] propose bypass algorithm for exclusive LLCs. While they study bypassing of cache blocks based on its L2 use behavior and L2-LLC trip counts, our bypass decisions are based on the working-set size of applications.

Kurian et al. [15] study data locality aware management of Private L1 caches for latency and energy benefits. An on-chip mechanism detects locality (spatial and temporal) of individual cache lines. On an L1 miss, only lines with high locality are allocated at L1 while the cache lines with low locality are not allocated at L1 (just accessed from L2). Thus, caching low locality data only at the shared cache (L2) avoids polluting the private L1 cache and saves energy by avoiding unnecessary data movement within on-chip hierarchy. The principal difference from our approach is that they manage *private caches* by forcing exclusivity on select data while we manage *shared caches* by forcing exclusivity on select application cache lines.

## 6.5 Cache partitioning techniques

Cache partitioning techniques [7, 8, 6, 10] focus on allocating fixed-number of ways per set to competing applications. Typically, a shadow tag structure [6] monitors the application's cache utility by using counters to record the number of hits each recency-position in the LRU stack receives. Essentially, the counter value indicates the number of misses saved if that *cache way* were not allocated to that application. The allocation policy assigns cache ways to applications based on their relative margin of benefit. The shadow tag mechanism exploits the stack property of LRU [25].

While these studies are constrained by the number of cache ways in the last level cache and hence, suffer from scalability with number of cores, some studies have proposed novel approaches to fine-grained cache partitioning [29, 28, 30] that breaks the partitioning-associativity barrier. These

mechanisms achieve fine-grained (at cache block level) through adjusting the eviction priorities. Jigsaw [30] shares the same hardware mechanism as Vantage [29], but uses a novel software cache allocation policy. The policy is based on the insight that miss-curves are typically non-convex and the convex-hull of the miss-curves provides scope for efficient and a faster allocation algorithm. Vantage, however, uses the same lookahead allocation policy ($O(N^2)$ algorithm) as in UCP. PriSM [28] proposes a pool of allocation policies which are based on the miss-rates and cache occupancies of individual applications.

Essentially, these mechanisms require quite larger associative shared cache. For tracking per-application utility, 256-way associative, LRU managed shadow tags are required [29][30]. Further, these techniques require significant modification to the existing cache replacement to adapt to their needs. On the contrary, ADAPT is simple and does not require modification to the cache states. Only the insertion priorities are altered.

# 7 Conclusion

Future multi-core processors will continue to employ shared last level caches. However, their associativity is expected to remain in the order of *16* consequently posing two new challenges: (i) the ability to manage more cores (applications) than associativity and (ii) the replacement policy must be application aware and allow to *discretely* ($> 2$) prioritize applications. Towards this end we make the following contributions: Firstly, we identify that existing approach of observing hit/miss pattern to approximate applications' behavior is not efficient. Then, we introduce the *Footprint-number* metric to dynamically capture the working-set size of applications. We propose Adaptive Discrete and de-prioritized Application PrioriTization (ADAPT), a new cache replacement algorithm, which consists of a monitoring mechanism and an insertion-priority-prediction algorithm. The monitoring mechanism dynamically captures the Footprint-number of applications on an interval basis. The prediction algorithm computes insertion priorities for applications from the Footprint-numbers under the assumption that smaller the Footprint-number, better the cache utilization. From experiments we show ADAPT is efficient and scalable ($\#applications \geq \#associativity$).

# Acknowledgment

The authors would like to thank the members of the ALF team for their feedback on this work. The authors also acknowledge the partial support provided by ERC Advanced Grant DAL No. 267175.

# References

[1] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In Proceedings of the 37th annual international symposium on Computer architecture (ISCA '10). ACM, New York, NY, USA, 60-71.

[2] Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, and Todd C. Mowry. 2012. The evicted-address filter: a unified mechanism to address both cache pollution and thrashing. In Proceedings of the 21st international conference on Parallel architectures and compilation techniques (PACT '12). ACM, New York, NY, USA, 355-366.

[3] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. 2008. Adaptive insertion policies for managing shared caches. In Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT '08). ACM, New York, NY, USA, 208-219.

[4] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. In Proceedings of the 34th annual international symposium on Computer architecture (ISCA '07). ACM, New York, NY, USA, 381-391.

[5] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. 2011. SHiP: signature-based hit predictor for high performance caching. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44). ACM, New York, NY, USA, 430-441.

[6] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39). IEEE Computer Society, Washington, DC, USA, 423-432.

[7] Kyle J. Nesbit, James Laudon, and James E. Smith. 2007. Virtual private caches. In Proceedings of the 34th annual international symposium on Computer architecture (ISCA '07). ACM, New York, NY, USA, 57-68.

[8] Yuejian Xie and Gabriel H. Loh. 2009. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09). ACM, New York, NY, USA, 174-183.

[9] Ravi Iyer. 2004. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In Proceedings of the 18th annual international conference on Supercomputing (ICS '04). ACM, New York, NY, USA, 257-266.

[10] A. Gupta, J. Sampson and M. Bedford Taylor, "TimeCube: A manycore embedded processor with interference-agnostic progress tracking," 2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), Agios Konstantinos, 2013, pp. 227-236.

[11] Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney, and Joseph Nuzman. 2012. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In Proceedings of the 21st international conference on Parallel architectures and compilation techniques (PACT '12). ACM, New York, NY, USA, 293-304.

[12] Hongliang Gao, Chris Wilkerson. A Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing. Joel Emer. JWAC 2010 - 1st JILP Worshop on Computer Architecture Competitions: cache replacement Championship, Jun 2010, Saint Malo, France. 2010.

[13] Jamison D. Collins and Dean M. Tullsen. 1999. Hardware identification of cache conflict misses. In Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture (MICRO 32). IEEE Computer Society, Washington, DC, USA, 126-135.

[14] Teresa L. Johnson, Daniel A. Connors, Matthew C. Merten, and Wen-mei W. Hwu. 1999. Run-Time Cache Bypassing. IEEE Trans. Comput. 48, 12 (December 1999), 1338-1354.

[15] George Kurian, Omer Khan, and Srinivas Devadas. 2013. The locality-aware adaptive cache coherence protocol. In Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13). ACM, New York, NY, USA, 523-534.

[16] Jayesh Gaur, Mainak Chaudhuri, and Sreenivas Subramoney. 2011. Bypass and insertion algorithms for exclusive last-level caches. In Proceedings of the 38th annual international symposium on Computer architecture (ISCA '11). ACM, New York, NY, USA, 81-92.

[17] Antonio Gonzlez, Carlos Aliagas, and Mateo Valero. 1995. A data cache with multiple caching strategies tuned to different types of locality. In Proceedings of the 9th international conference on Supercomputing (ICS '95). ACM, New York, NY, USA, 338-347.

[18] Scott McFarling. 1992. Cache replacement with dynamic exclusion. In Proceedings of the 19th annual international symposium on Computer architecture (ISCA '92). ACM, New York, NY, USA, 191-200.

[19] Ricardo A. Velasquez, Pierre Michaud, Andr Seznec. BADCO: Behavioral Application-Dependent Superscalar Core Model. SAMOS XII: International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, Jul 2012, Samos, Greece. 2012.

[20] Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08).

[21] Wei-Fen Lin and Steven K. Reinhardt,2002, Predicting Last-Touch References under Optimal Replacement. In Technical Report CSE-TR-447-02, University of Michigan, 2002

[22] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. 2008. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41). IEEE Computer Society, Washington, DC, USA, 222-233.

[23] An-Chow Lai, Cem Fide, and Babak Falsafi. 2001. Dead-block prediction and dead-block correlating prefetchers. In Proceedings of the 28th annual international symposium on Computer architecture (ISCA '01). ACM, New York, NY, USA, 144-154.

[24] Tripti S. Warrier, B. Anupama, and Madhu Mutyam. 2013. An application-aware cache replacement policy for last-level caches. In Proceedings of the 26th international conference on Architecture of Computing Systems (ARCS'13), Hana Kubtov, Christian Hochberger, Martin Dank, and Bernhard Sick (Eds.). Springer-Verlag, Berlin, Heidelberg, 207-219.

[25] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. IBM Syst. J. 9, 2 (June 1970), 78-117. DOI=http://dx.doi.org/10.1147/sj.92.0078.

[26] Pierre Michaud. 2013. Demystifying multicore throughput metrics. IEEE Comput. Archit. Lett. 12, 2 (July 2013), 63-66. DOI=http://dx.doi.org/10.1109/L-CA.2012.25.

[27] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. 2000. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture (MICRO 33). ACM, New York, NY, USA, 32-41. DOI=http://dx.doi.org/10.1145/360128.360134.

[28] R Manikantan, Kaushik Rajan, and R Govindarajan. 2012. Probabilistic shared cache management (PriSM). In Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12). IEEE Computer Society, Washington, DC, USA, 428-439.

[29] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: scalable and efficient fine-grain cache partitioning. In Proceedings of the 38th annual international symposium on Computer architecture (ISCA '11). ACM, New York, NY, USA, 57-68. DOI=http://dx.doi.org/10.1145/2000064.2000073.

[30] Nathan Beckmann and Daniel Sanchez. 2013. Jigsaw: scalable software-defined caches. In Proceedings of the 22nd international conference on Parallel architectures and compilation techniques (PACT '13). IEEE Press, Piscataway, NJ, USA, 213-224.

[31] R Manikantan, Kaushik Rajan, and R Govindarajan. 2011. NUcache: An efficient multicore cache organization based on Next-Use distance. In Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11). IEEE Computer Society, Washington, DC, USA, 243-253.

[32] G. Keramidas, P. Petoumenos and S. Kaxiras, "Cache replacement based on reuse-distance prediction," 2007 25th International Conference on Computer Design, Lake Tahoe, CA, 2007, pp. 245-250.

[33] Mazen Kharbutli, Yan Solihin, "Counter-Based Cache Replacement and Bypassing Algorithms", IEEE Transactions on Computers, April 2008.

[34] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. 2012. Improving Cache Management Policies Using Dynamic Reuse Distances. In Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45). IEEE Computer Society, Washington, DC, USA, 389-400. DOI=http://dx.doi.org/10.1109/MICRO.2012.43.

[35] David Eklov, David Black-Schaffer, and Erik Hagersten. 2011. Fast modeling of shared caches in multicore systems. In Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC '11). ACM, New York, NY, USA, 147-157. DOI=http://dx.doi.org/10.1145/1944862.1944885.

[36] Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai.Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. 2010. Accelerating multicore reuse distance analysis with sampling and parallelization. In Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT '10). ACM, New York, NY, USA, 53-64. DOI=http://dx.doi.org/10.1145/1854273.1854286.

[37] https://software.intel.com/en-us/blogs/2014/06/18/benefit-of-cache-monitoring

[38] http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/

[39] http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon E7-4850 v4.html

[40] http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html

[41] Masamichi Takagi and Kei Hiraki. 2004. Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches. In Proceedings of the 18th annual international conference on Supercomputing (ICS '04). ACM, New York, NY, USA, 20-30. DOI=http://dx.doi.org/10.1145/1006209.1006213.

[42] B. Panda and S. Balachandran, "CSHARP: Coherence and SHaring Aware Cache Replacement Policies for Parallel Applications," 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing, New York, NY, 2012, pp. 252-259. doi: 10.1109/SBAC-PAD.2012.27

[43] R. Natarajan and M. Chaudhuri, "Characterizing multi-threaded applications for designing sharing-aware last-level cache replacement policies," 2013 IEEE International Symposium on Workload Characterization (IISWC), Portland, OR, 2013, pp. 1-10. doi: 10.1109/IISWC.2013.6704665

[44] Pavlos Petoumenos, Georgios Keramidas, and Stefanos Kaxiras. 2009. Instruction-based reuse-distance prediction for effective cache management. In Proceedings of the 9th international conference on Systems, architectures, modeling and simulation (SAMOS'09), Walid Najjar and Michael J. Schulte (Eds.). IEEE Press, Piscataway, NJ, USA, 49-58.