Discrete Cache Insertion Policies for Shared Last Level Cache Management on Large Multicores

Aswinkumar Sridharan, André Seznec

Discrete Cache Insertion Policies for Shared Last Level Cache Management on Large Multicores

Aswinkumar Sridharan & André Seznec
INRIA/IRISA
Campus de Beaulieu
35042 Rennes, France
{*aswinkumar.sridharan , andre.seznec*}*@inria.fr*

***Abstract*—Multi-core processors employ shared Last Level Caches (LLC). This trend will continue in the future with large multi-core processors (16 cores and beyond) as well. At the same time, the associativity of LLC tends to remain in the order of sixteen. Consequently, with large multicore processors, the number of cores that share the LLC becomes larger than the associativity of the cache itself. LLC management policies have been extensively studied for small scale multi-cores (4 to 8 cores) and associativity degree in the 16 range. However, the impact of LLC management on large multi-cores is essentially unknown, in particular when the associativity degree is smaller than the number of cores. In this study, we introduce Adaptive Discrete and deprioritized Application PrioriTization (ADAPT), an LLC management policy addressing the large multi-cores where the LLC associativity degree is smaller than the number of cores. ADAPT builds on the use of the Footprint-number metric. We propose a monitoring mechanism that dynamically samples cache sets to estimate the Footprint-number of applications and classifies them into discrete (distinct and more than two) priority buckets. The cache replacement policy leverages this classification and assigns priorities to cache lines of applications during cache replacement operations. We further find that de-prioritizing certain applications during cache replacement is beneficial to the overall performance. We evaluate our proposal on 16, 20 and 24-core multi-programmed workloads and discuss other aspects in detail.**

***Keywords*-Footprint-number; Discrete Priorities; More cores than associativity;**

I. 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 [1]–[9], [11], [12]. 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 cores sharing the cache in multi-core processors. Hence, we are presented with the scenario of managing shared caches where $(\#cores \geq \#llc\_ways)$. 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[2].

Prior studies [1]–[5], [12] 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.

Therefore, a new mechanism that efficiently captures the application behavior and its ability to utilize the cache while simultaneously allowing the replacement algorithm to enforce different priorities across applications is required. 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

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

[2]In this paper, we discuss only hardware based solution.

efficient cache management. Altogether, we propose *Adaptive Discrete and De-prioritized Application PrioriTization (ADAPT)* replacement policy for efficient management of large multi-core shared caches and make the following contributions:

•We consider cache replacement in shared caches in the context of $(\#cores \geq \#llc\_ways)$: we find that observing the hit/miss pattern of applications to approximate their cache utility is not an efficient approach when the cache is shared by a large number of applications, and a new mechanism is required.

•We propose a new metric *Footprint-number* to approximate application behavior at run-time and propose an insertion-priority- prediction algorithm that uses Footprint-number to assign discrete (more than two) priorities for applications. Our evaluation across 60 16-core multi-programmed workloads shows that ADAPT provides 4.7% improvement over TA-DRRIP algorithm on the weighted speed-up metric. Further evaluations show that ADAPT is scalable with respect to the number of applications sharing the cache and also with larger shared cache sizes.

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

II. Motivation

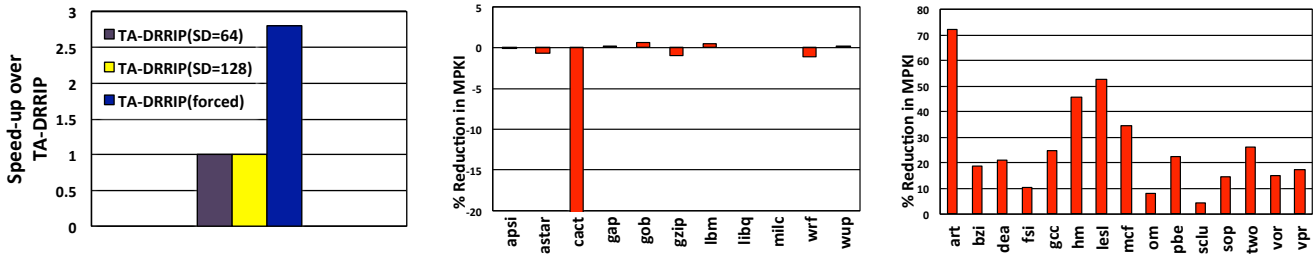
We motivate the need for a new approach to cache management by (i) demonstrating the inefficient learning of application behavior in large-scale multi-cores and (ii) the complexity of other approaches that fairly isolate the application behavior.

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 [1]–[3], [5], [12] 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) and mixed (recency-friendly pattern mixing with scan) type of access patterns, BRRIP handles thrashing (larger working-set) patterns.

TA-DRRIP learns SRRIP policy for all classes of applications, including the ones with working-set size larger than the cache. However, applications with working-set size larger than the cache cause thrashing when they share the cache with other (cache-friendly) applications. Based on this observation, by explicitly preventing thrashing applications from competing with the non-thrashing (or, cache friendly) applications for cache space, performance can be improvement. In other words, implementing BRRIP policy for these thrashing applications will be beneficial to the overall performance. Figure 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 records the number of misses caused by the competing policies and making inefficient decisions on application priorities. The experiments are performed on a 16MB, 16-way associative cache, which is shared by all sixteen applications. Table III shows other simulation parameters. Results in Figure 1 are averaged from all the 60 16-core workloads. Also, from bars 1 and 2, we see that the observed behavior of TA-DRRIP is not dependent on the number of sets dedicated for policy learning.

Figures 1b and 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 Figure 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.

Complexity in other approaches: On the other hand, reuse-distance based techniques [33], [35]–[37] explicitly compute the reuse distance values of cache lines. However, they involve significant overhead due to storage and related bookkeeping operations. Further, these techniques are either dependent on the replacement policy [36], [37] or



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

Figure 1: Impact of implementing BRRIP policy for thrashing applications

require modifying the cache tag arrays [33], [35]. Similarly, cache partitioning techniques [10], [29]–[31] incur significant overhead due to larger associative (up to 128/256-way) tag structures, or require modification to the replacement policies to adapt to their needs [31] [30]. From these discussions, we see that a simple, efficient and scalable cache monitoring mechanism is required. Further, recall that cache replacement policy in large multi-core processors is required to be application-aware, and enforce different priorities. Therefore, an efficient cache management technique must augment a cache monitoring mechanism that conforms to the two goals.

III. Adaptive Discrete and de-prioritized Application PrioriTization

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.

A. 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*[3] 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 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: Figure 2a 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[4], 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: Figure 2b 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.

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

[4]Only demand accesses update the recency state

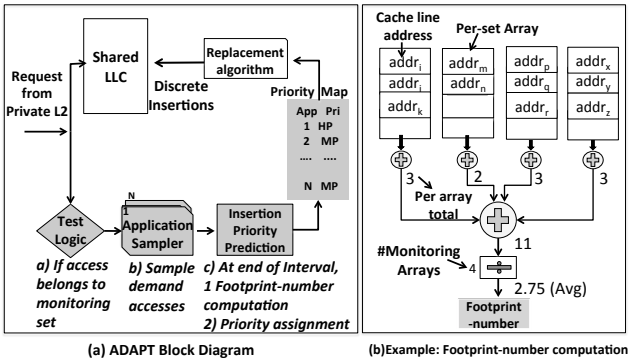


Figure 2: (a) ADAPT Block Diagram and b) example for Footprint-number computation

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.

B. Footprint-number based Priority assignment

Like prior studies [1], [2], [5], [39], 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. The algorithm assumes that the LLC associativity is *16*. However, it still works for larger associative caches as we show later. Table I summarizes the insertion priorities for each classification. Experiments are performed by varying the high-priority range between [0,3] and [0,8] (6 different ranges), keeping the low-priority range unaffected. Similarly, by keeping the high-priority range [0,3] constant, we change the low-priority range between (7,16) to (12,16) (6 different ranges). In total, from 36 different experiments we fix the priority-ranges. A dynamic approach that uses run-time information to assign priorities is more desirable. We defer this approach to future work. Priority assignments are 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 fits 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.

Table I: 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

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[5].

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 type 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.

C. Hardware Overhead

The additional hardware required by our algorithm is the application sampler and insertion priority prediction logic.

[5]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 II: 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 appx

The application sampler consists of an array and a counter. The size of the array is same as the associativity. From Section III-B, 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 $\times 40 = 8160bits$. 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 $[8160\ bits + 40\ bits] = 8200bits/application$. In other words, 1KB (approximately) per application.

Table II 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. It should be noted that ADAPT does not require dedicating some cache sets for policy learning. Regarding energy consumption, we empirically conjecture that the monitoring system will consume approximately 1/25th of the power of the main cache tag array. (40 sets per application and 16 applications results in 1/25th of accesses directed to the monitoring cache).

D. 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 [1]–[6], we target systems in which LLC is organized as multiple banks with uniform access latency.

IV. Experimental Study

A. Methodology

For our study, we use BADCO [19] cycle-accurate x86 CMP simulator. Table III 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], [2], [5].A VPC [7] based arbiter schedules requests from L2 to LLC. We use DRAM model similar to [2].

Table III: 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 [28]

B. Benchmarks

Table IV: Empirical Classification of Applications

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

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 V shows the classification of all the benchmarks and Table IV 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 V, the column Fpn(A) represents Footprint-number value 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

Table V: Benchmark Characteristics

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

and simulate the next 300M instructions. If an application finishes execution, it is re-executed until all applications finish.

C. Workload Design

Table VI 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 $\#applications \geq \#llcassociativity$.

Table VI: 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

V. Results and Analysis

A. Performance on 16-core workloads

Figure 3 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 Figure 3, we observe that ADAPT consistently outperforms other cache replacement policies. It achieves up to 7% improvement with 4.7% on average.

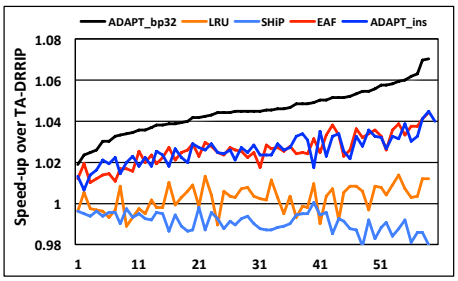


Figure 3: Performance of 16-core workloads

As mentioned in SectionII, 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 *near-immediate*) 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%appx) is due to *inaccurate* distant predictions on certain cache-friendly applications. Overall, ADAPT uses *Footprint-number* metric to efficiently distinguish across applications.

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 *near-immediate* 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. But, 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

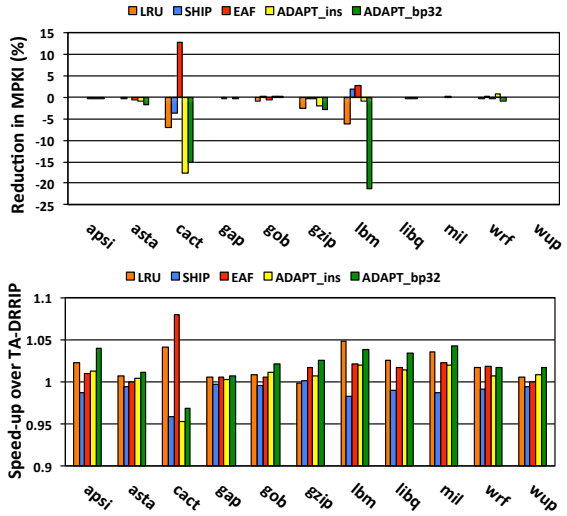


Figure 4: MPKI(top) and IPC(below) of thrashing applications

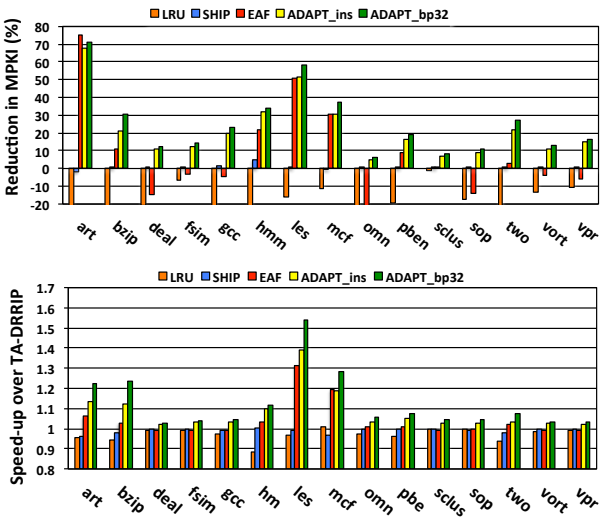


Figure 5: MPKI(top) and IPC(below) of non-thrashing applications

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 *near-immediate* (RRPV 2) assignment.

B. 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 Figures 4 & 5, we observe that bypassing does not cause slow-down (except *cactusADM*) on least priority applications and provides substantial improvement on high and medium priority applications. Therefore, our assumption that *bypassing* most of the cache lines of the least-priority applications to be beneficial to the overall performance is confirmed. 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. For most of the applications bypassing provides substantial improvement in MPKI and IPC, as shown in Figure 5. Bypassing affects only *cactusADM*. This is because some of their cache lines are reused immediately after insertion. 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].

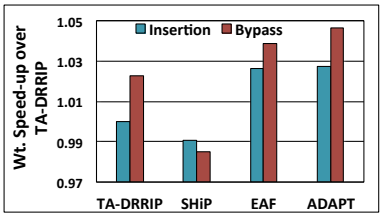


Figure 6: Impact of Bypassing on replacement policies

C. 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 6, 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 minor drop in performance. On the contrary, TA-DRRIP, which implements *bimodal*(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% (appx) of *distant* reuse predictions are incorrect[6]. Overall, from Figure 6, 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

[6]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.

to approximate an application's behavior: using Footprint-number, ADAPT distinguishes thrashing applications and bypasses their cache lines.

D. 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. Figure 8 shows 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. Here, 20 and 24-core workloads are studied on 16MB,16-way associative cache. Recall our proposition : $(\#cores \geq associativity)$.

E. Sensitivity to Cache Configurations

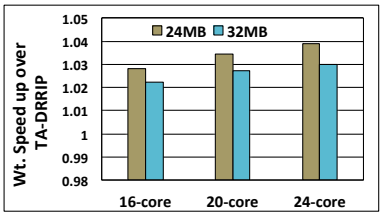


Figure 7: Performance on Larger Caches

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(Figure 7).

F. Other Multi-core Metrics

Table VII shows the performance of ADAPT on the Harmonic Mean of the Normalized IPCs [41] and the Harmonic, Geometric and Arithmetic Means of IPCs [27]. ADAPT shows consistent improvement under other metrics as well.

Table VII: Performance of ADAPT over TA-DRRIP

Metric	4-core	8-core	16-core	20-core	24-core
Wt.Speed-up	5.48%	3.65%	4.67 %	5.86%	5.79%
Norm. HM	5.34%	4.31%	6.66%	8.06%	8.35%
GM of IPCs	5.43%	3.98%	5.34%	6.8%	6.95%
HM of IPCs	5.17%	3.94%	5.43%	7.29%	7.77%
AM of IPCs	5.27%	3.27%	4.82%	5.85%	5.63%

VI. Related Work

Numerous studies have proposed novel ideas to manage multi-core shared caches. Here, we summarize some of them in the context of large-scale multi-core caches.

A. Insertion priority prediction

DIP [4] is one of the foremost proposals to alter the *insertion priority* of cache lines. In particular, they observe applications with working-set size larger than the cache to thrash under LRU and propose *Bimodal* policy for such workloads. DRRIP [1] predicts the reuse behavior of cache lines into re-reference interval buckets. RRIP consists of *SRRIP* and *BRRIP* policies. SRRIP handles mixed (recency-friendly pattern mixing with scan) and scan type of access patterns. BRRIP handles thrashing patterns. SHiP [5] and EAF [2] add further intelligence to the insertion predictions. SHiP uses Program-counter, Instruction sequence and Memory region signatures (separate mechanisms) to predict different priorities (SRRIP or BRRIP) for regions of accesses corresponding to the signature. 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.

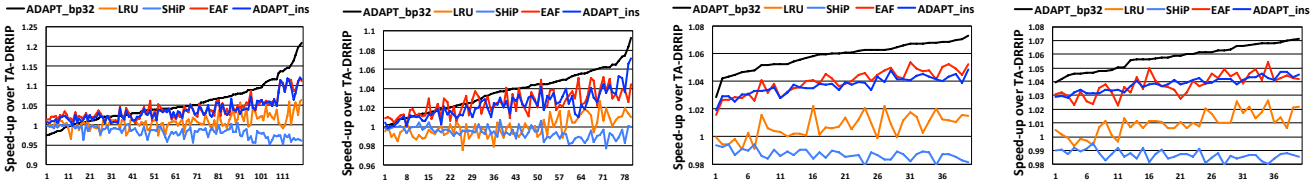
All these approaches use only two (SRRIP or BRRIP) insertion policies. Moreover, as discussed in the motivation section, they cannot be adapted to enable discrete prioritization. On the contrary, ADAPT is able to classify applications into discrete priority buckets and achieve higher performance. SHiP and EAF predict priorities at the granularity of (regions of) cache lines. However, in commercial designs [38] [40], which use SW-HW co-designed approach to resource management, the system software decides fairness or performance goals only at an application granularity. Hence, it is desirable that the cache management also performs application level performance optimizations.

B. Reuse distance prediction

Some studies [34]–[37] compute the reuse distance values of cache lines at run-time to perform cache replacements. 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 modifying the cache tag arrays to store reuse distance values of cache lines. Schuff et al. [37] proposes a sampling and a parallel approach to measure the reuse distance of multi-threaded applications. NUCache [32] 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.

C. Eviction priority prediction

Victim selection techniques try to predict cache lines that are either *dead* or very unlikely to be re-used soon [21]–[25]. A recent proposal, *application-aware cache replacement* [24] predicts cache lines with very long re-use distance



(a) 4-core (120 workloads) (b) 8-core (80 workloads) (c) 20-core (40 workloads) (d) 24-core (40 workloads)

Figure 8: Performance of ADAPT with respect to number of applications

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 around the hit-gap value is evicted. In large multi-cores, under this approach, certain cache-friendly applications could get hidden behind the memory-intensive ones, and suffer more misses. However, ADAPT would be able to classify such applications and retain their cache lines for longer time. Further, this policy requires expensive look-up operations and significant modifications to the cache tag array.

D. Cache Bypassing

Many studies have proposed bypassing of cache lines [12]–[14], [17], [18]. 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 larger multi-cores, 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 show in Section V-C, bypassing least-priority cache lines is beneficial to other replacement policies as well. Segmented-LRU [12] learns the benefit of bypassing on randomly selected cache lines by observing the relative timing of the *victim/non-bypassed* or *retained/by-passed* cache lines. However, observing the hits/misses on the shared cache is not an efficient way to decide on policies as they may lead to inefficient decisions. Gaur et al. [16] propose bypass algorithm for exclusive LLCs. While they study bypassing of cache blocks based on its L2 use and L2-LLC trip counts, our bypass decisions are based on the working-set size of applications. [15] uses data locality to manage placement of cache lines between Private L1 and Shared L2. Only high locality lines are inserted at L1 while cache lines with low locality are not allocated. 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.

E. Cache partitioning techniques

Cache partitioning techniques [6]–[8], [10] focus on allocating fixed-number of ways per set to competing applications. Typically, a shadow tag structure (that exploits stack property of LRU [26]) [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 allocated to that application. The allocation policy assigns cache ways to applications based on their relative margin of benefit. While these studies suffer from scalability with number of cores, some studies have proposed novel approaches to fine-grained cache partitioning [29]–[31] that break the partitioning-associativity barrier. These mechanism achieve fine-grained (at cache block level) through adjusting the eviction priorities. Jigsaw [30] leverages Vantage [31] for the cache hardware, but uses a novel software cache allocation policy based on the insight that miss-curves are typically non-convex and this property provides scope for efficient and a faster allocation algorithm. PriSM [29] 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 caches. For tracking per-application utility, 256-way LRU managed shadow tags are required [30] [31]. Further, these techniques require significant modification to the existing cache replacement to adapt to their needs. Contrastingly, ADAPT does not require modifying the cache states. Only the insertion policies are altered.

VII. Summary and Future Work

Future multi-core processors will continue to employ shared last level caches. However, their associativity is expected to remain in the order of *sixteen* 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:

•We identify that existing approach of observing hit/miss pattern to approximate applications' behavior is not efficient.
•We introduce the *Footprint-number* metric to dynamically capture the working-set size of applications. We propose Adaptive Discrete and de-prioritized Application Priori-Tization (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 ($(\#cores \geq \#associativity)$). Commercial processors typically employ mid-level (L2) cache prefetching. Our present study does not include L2 prefetching. We intend to study large multi-core shared caches with L2 prefetchers in the future.

ACKNOWLEDGMENT

The authors would like to thank the members of the ALF team for their suggestions. This work is partially supported by ERC Advanced Grant DAL No. 267175.

REFERENCES

[1] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). In ISCA 2010.

[2] Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, and Todd C. Mowry. The evicted-address filter: a unified mechanism to address both cache pollution and thrashing. In PACT 2012.

[3] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. Adaptive insertion policies for managing shared caches. In PACT 2008.

[4] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In ISCA 2007.

[5] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. SHiP: signature-based hit predictor for high performance caching. In MICRO 2011.

[6] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In MICRO 2006.

[7] Kyle J. Nesbit, James Laudon, and James E. Smith. Virtual private caches. In ISCA 2007.

[8] Yuejian Xie and Gabriel H. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In ISCA 2009.

[9] Ravi Iyer. 2004. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In ICS 2004.

[10] Gupta, A.; Sampson, J.; Taylor, M.B., TimeCube: A many-core embedded processor with interference-agnostic progress tracking, In SAMOS 2013.

[11] Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney, and Joseph Nuzman. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In PACT 2012.

[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.

[13] Jamison D. Collins and Dean M. Tullsen. Hardware identification of cache conflict misses. In MICRO 1999.

[14] Teresa L. Johnson, Daniel A. Connors, Matthew C. Merten, and Wen-mei W. Hwu. Run-Time Cache Bypassing. in IEEE TC, 1999.

[15] George Kurian, Omer Khan, and Srinivas Devadas. The locality-aware adaptive cache coherence protocol. In ISCA 2013.

[16] Jayesh Gaur, Mainak Chaudhuri, and Sreenivas Subramoney. Bypass and insertion algorithms for exclusive last-level caches. In ISCA 2011.

[17] Antonio Gonzlez, Carlos Aliagas, and Mateo Valero. A data cache with multiple caching strategies tuned to different types of locality. In ICS 1995.

[18] Scott McFarling. Cache replacement with dynamic exclusion. In ISCA 1992.

[19] Velasquez, R. A and Michaud, P. and Seznec, A. BADCO: Behavioral Application-Dependent Superscalar Core model. In SAMOS 2012.

[20] Onur Mutlu and Thomas Moscibroda.Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In ISCA 2008.

[21] Wei-Fen Lin and Steven K. Reinhardt,2002, Predicting Last-Touch References under Optimal Replacement

[22] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In MICRO 2008

[23] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In ISCA 2001.

[24] Tripti S. Warrier, B. Anupama, and Madhu Mutyam. An application-aware cache replacement policy for last-level caches. In ARCS 2013.

[25] Samira M. Khan, Yingying Tian, and Daniel A. Jimenez. Sampling Dead Block Prediction for Last-Level Caches. In MICRO 2010.

[26] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. IBM Syst. J. 9, 2 (June 1970),

[27] Pierre Michaud. Demystifying multicore throughput metrics. IEEE Comp. Arch. Letters, June 2013.

[28] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In MICRO 2000.

[29] R Manikantan, Kaushik Rajan, and R Govindarajan. Probabilistic shared cache management (PriSM). In ISCA 2012.

[30] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In ISCA 2011.

[31] Nathan Beckmann and Daniel Sanchez. Jigsaw: scalable software-defined caches. In PACT 2013.

[32] R Manikantan, Kaushik Rajan, and R Govindarajan. NUcache: An efficient multicore cache organization based on Next-Use distance. In HPCA 2011.

[33] Keramidas, G.; Petoumenos, P.; Kaxiras, S., "Cache replacement based on reuse-distance prediction," in ICCD 2007.

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

[35] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. In MICRO 2012.

[36] David Eklov, David Black-Schaffer, and Erik Hagersten. Fast modeling of shared caches in multicore systems. In HiPEAC 2011.

[37] Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai.Accelerating multicore reuse distance analysis with sampling and parallelization. In PACT 2010.

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

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

[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] Kun Luo; Gummaraju, J.; Franklin, M.,"Balancing thoughput and fairness in SMT processors," in ISPASS 2001.