Locality & Utility Co-optimization for Practical Capacity Management of Shared Last Level Caches

Dongyuan Zhan, Hong Jiang, Sharad C. Seth
Department of Computer Science & Engineering
University of Nebraska - Lincoln
Lincoln, NE 68588, USA
{dzhan, jiang, seth}@cse.unl.edu

ABSTRACT

Shared last-level caches (SLLCs) on chip-multiprocessors play an important role in bridging the performance gap between processing cores and main memory. Although there are already many proposals targeted at overcoming the weaknesses of the *least-recently-used* (LRU) replacement policy by optimizing either locality or utility for heterogeneous workloads, very few of them are suitable for practical SLLC designs due to their large overhead of $\log associativity$ bits per cache line for re-reference interval prediction. The two recently proposed practical replacement policies, TA-DRRIP and SHiP, have significantly reduced the overhead by relying on just 2 bits per line for prediction, but they are oriented towards managing locality only, missing the opportunity provided by utility optimization.

This paper is motivated by our two key experimental observations: (i) the *not-recently-used* (NRU) replacement policy that entails only one bit per line for prediction can satisfactorily approximate the LRU performance; (ii) since locality and utility optimization opportunities are concurrently present in heterogeneous workloads, the co-optimization of both would be indispensable to higher performance but is missing in existing practical SLLC schemes. Therefore, we propose a novel practical SLLC design, called COOP, which needs just one bit per line for re-reference interval prediction, and leverages lightweight per-core locality & utility monitors that profile sample SLLC sets to guide the co-optimization. COOP offers significant throughput improvement over LRU by 7.67% on a quad-core CMP with a 4MB SLLC for 200 random workloads, outperforming both of the recent practical replacement policies at the in-between cost of 17.74KB storage overhead (TA-DRRIP: 4.53% performance improvement with 16KB storage cost; SHiP: 6.00% performance improvement with 25.75KB storage overhead).

Categories and Subject Descriptors

B.3.2 [**Memory Structures/Design Styles**]: Cache Memories

ICS'12, June 25–29, 2012, San Servolo Island, Venice, Italy.

General Terms

Design, Management, Performance

Keywords

Chip Multiprocessors, Shared Last Level Caches, Practical Capacity Management, Locality & Utility Co-Optimization

1. INTRODUCTION

The shared last level cache (SLLC) organization is commonly adopted in chip multiprocessor (CMP) products, such as AMD's Phenom II X6 and Intel's Core i7, to simplify both cache capacity sharing and coherence support for processing cores. Although the SLLC of a CMP is accessible to all cores, its large aggregate capacity alone cannot guarantee optimal performance without good management strategies. This is especially true when the cores are running a heterogeneous mix of threads that have diverse requirements for SLLC resources, which becomes increasingly common with the widespread deployment of CMPs in complex applications, such as cloud computing [1].

Because of their vital role in minimizing the expensive memory traffic, SLLC capacity management schemes have been extensively studied for a long time by the research community. It has been noticed that the *least-recently-used* (LRU) replacement policy becomes less effective for SLLCs due to the diminished *access locality* [1] at the last cache level [2–5] and the uncoordinationed capacity allocation among heterogeneous threads [6, 7]. In response to LRU's limitations, two approaches have emerged in the literature. First, alternative replacement policies, such as TADIP [2], SDBP [3] and NUcache [5], have been proposed to manage locality by temporally assigning and adjusting lifetime for blocks. Second, working with a different principle, cache partitioning schemes, including UCP [6] and PIPP [7], try to optimize *utility* [2] by spatially partitioning the SLLC capacity among concurrent threads to maximize performance.

Although the aforementioned proposals have demonstrated desirable performance improvement over LRU in simulations, they are not practically useful due to the high storage overhead entailed by them. Specifically, they are all based on the assumption that each cache line has $\log A$ bits for its *re-reference interval prediction value* (RRPV) [8] that is used to estimate how soon an accessed block will be reused, where A is the set associativity. But the $\log A$-bit overhead per line is

[1]In this paper, locality specifically refers to temporal locality
[2]Utility is defined as a thread's ability to reduce misses with a given amount of allocated cache capacity.

considered prohibitive for SLLCs according to industry standards [4]. As a result, since the uniprocessor era, commodity processors have relied on lightweight LRU approximations for cache management, such as the *not-recently used* (NRU) replacement policy that requires just one-bit overhead in each cache line. It has been experimentally shown that the lightweight NRU is able to perform almost (99.52% [8]) as well as LRU but still cannot provide optimized performance for CMPs either.

Recent efforts have attempted to bridge the gap between the theoretical cache research and practical SLLC designs. Jaleel et al. [8] propose to use 2 bits in each line's RRPV field for a thread-aware dynamic SLLC replacement policy called TA-DRRIP. TA-DRRIP outperforms the baseline LRU and NRU policies by coordinating locality optimization for all of the co-scheduled threads. Rooted in the same two-bit RRPV substrate, the recent work SHiP [9] further improves over TA-DRRIP by accounting for the variations in locality at the finer-grained memory instruction level for re-reference interval prediction, but incurs more overhead than the thread-level TA-DRRIP approach.

Through our analysis of and experimental study on practical SLLC capacity management solutions, we obtain two important insights that counter the previous research: (*i*) since the minimum-overhead NRU achieves almost the same practical performance as LRU but lacks such theoretical traits as the LRU stack property, it is possible to adopt the minimum-overhead 1-bit RRPV substrate in the entire SLLC and utilize monitors with good theoretical properties yet at a slightly more storage cost for just sample sets, so that the goals of overhead reduction and performance improvement can be achieved at the same time; (*ii*) both locality and utility optimization opportunities are present in heterogeneous CMP workloads, but the practical schemes such as TA-DRRIP and SHiP are oriented only towards locality management, missing performance potentials provided by utility optimization.

Hence, we propose a novel practical SLLC management design, called COOP (an acronym for *locality and utility CO-OPtimization*), which achieves higher performance than both TA-DRRIP and SHiP but at comparable or lower overhead. COOP uses a single bit in each line's RRPV field, and employs a classic LRU-stack hit profiler and a novel *logarithmic-distance* BNRU (a.k.a., *bimodal* NRU, for thrashing prevention) hit profiler to monitor the interleaved locality and utility of each thread. Leveraging the information about all co-scheduled threads, COOP spatially allocates SLLC cache ways among the threads and temporally makes the best use of their partitions in an interactive way, so that the highest utility provided by either NRU or BNRU, whichever is better, can be exploited by locality and utility co-optimization for all of the threads. Our evaluation shows that COOP improves the throughput performance for 200 random workloads by 7.67% on a quad-core CMP with a 4MB SLLC, all at the cost of only 17.74KB storage overhead which is comparable to TA-DRRIP (16KB) but lower than SHiP (25.75KB), while outperforming both of them (compared to TA-DRRIP's 4.53% and SHiP's 6.00% throughput improvements).

The rest of the paper is organized as follows. Section 2 elaborates on essential background and research motivations. Section 3 describes the design and implementation of our proposed solution, followed by a thorough performance evaluation in Section 4. Related work is discussed in Section 5 and the paper concludes in Section 6.

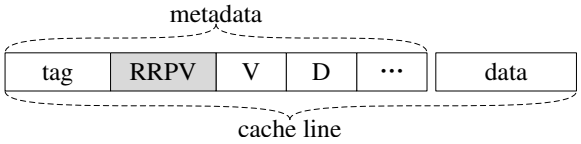


Figure 1: The Structure of a Cache Line

2. BACKGROUND & MOTIVATIONS

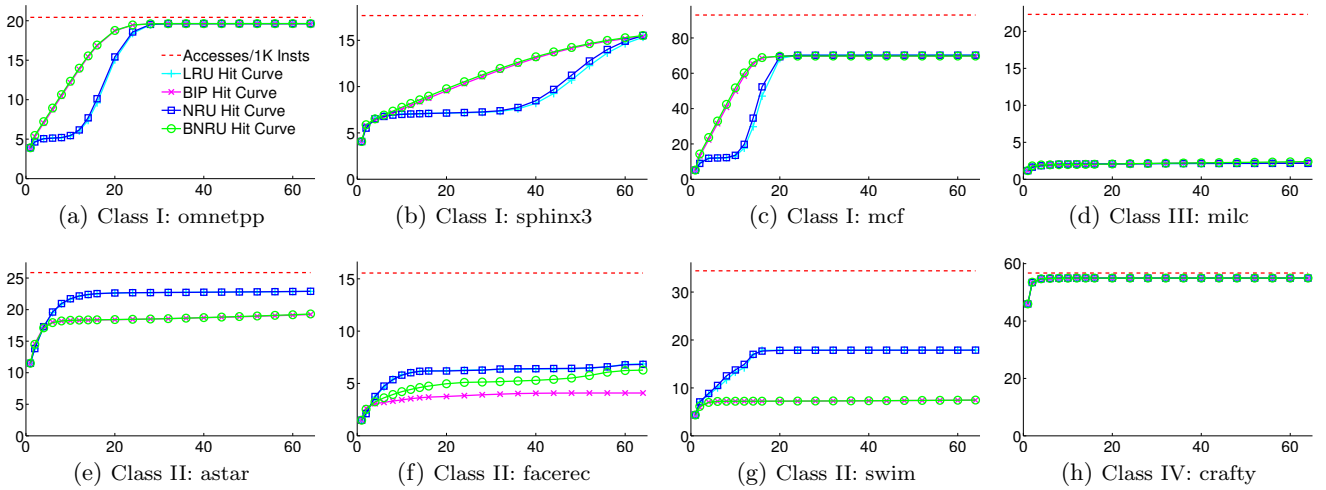
Although the entire SLLC capacity can be accessed by all cores, allowing free use, without constraints, does not lead to efficient utilization of SLLC resources. Therefore, various strategies have been proposed to make the best use of the SLLC capacity. We will briefly describe their working principles here but place our research in a broader context of related work in Section 5. Since practical SLLC management is our main focus in the paper, we will analyze the strengths and weaknesses of the state-of-the-art practical schemes and provide quantitative evidence in support of our conclusions.

2.1 Theoretical SLLC Management Proposals

As is illustrated in Figure 1, a cache line typically consists of two parts, *metadata* and a *data block*. The metadata includes such fields as a tag, a *re-reference prediction value* (RRPV), a valid bit, a dirty bit and other information bits (e.g., coherence bits). Given the set associativity A, the LRU replacement policy adopts $\log A$ bits for each line's RRPV field to indicate its current position in the LRU stack. For instance, the RRPVs of the *most-recently-used* and the *least-recently-used* blocks are 0 and $A-1$ respectively. We note that most previous SLLC management proposals [2, 3, 5–7] are based on the assumption of $\log A$-bit RRPVs. But since this overhead is prohibitive for SLLCs that have large set associativity from the industry's point of view [4], those proposals may arguably only be used for theoretical research. In general, the theoretical proposals can be categorized into either alternative replacement policies for locality management or capacity partitioning schemes for utility optimization, as follows.

Locality-Oriented Alternative Replacement Policies: Because the LRU replacement policy aims to favor cache access recency (or *locality*) only, it can result in thrashing when the working set size of a workload is larger than the cache capacity and the cache access pattern is locality-unfriendly (e.g., a large cyclic working set [10]). Alternative replacement policies, such as TADIP [2], SDBP [3] and NU-cache [5], are proposed to optimize locality by temporally assigning and adjusting lifetimes for cached blocks.

Utility-Oriented Capacity Partitioning Schemes: The *utility* of a thread represents its ability to reduce misses with a given amount of allocated SLLC capacity [6]. Although threads may vary greatly in their utility, an LRU-managed SLLC is oblivious of such differences when threads are co-scheduled and their cache accesses are mixed. In response to this shortcoming, several previous studies, such as UCP [6] and PIPP [7], propose to spatially partition the SLLC among competing threads based on the utility information captured by per-thread LRU-stack profilers, notably



(a) Class I: omnetpp (b) Class I: sphinx3 (c) Class I: mcf (d) Class III: milc

(e) Class II: astar (f) Class II: facerec (g) Class II: swim (h) Class IV: crafty

Figure 2: The HPKIs (*hits per thousand instructions*) of LRU, BIP, NRU and BNRU as a function of the SLLC capacity for the SPEC benchmarks. The x-axis shows the SLLC capacity measured in the number of ways (given that the number of sets and line size are fixed), while the y-axis represents *HPKIs*. The dotted roofline in each figure indicates the total number of SLLC accesses per 1k instructions (independent of the SLLC capacity). The 8 benchmarks are divided into four classes according to their locality and utility characteristics.

improving the performance over the baseline LRU replacement policy.

2.2 Practical SLLC Management Schemes

The LRU approximations, such as the *not-recently-used* (NRU) replacement policy, are practically adopted in commodity processors because its RRPV filed requires just a single bit. In NRU, a 0-valued (or 1-valued) RRPV indicates that the block was recently used (or not recently used). Upon a cache hit, NRU updates the block's RRPV bit to 0. In order to select a victim block for eviction followed by a cache miss, there are two possible cases: (*i*) if there are any blocks with 1-valued RRPVs in the target set, the first such block found by scanning the set will be selected as the replacement candidate; (*ii*) otherwise, NRU increments all blocks' RRPVs and then repeats the process (*i*) to find the replacement candidate. Upon a cache fill, the RRPV of the newly-inserted block is set to 0. According to extensive experimental statistics [8], NRU achieves a desirable 99.52% performance approximation to LRU. But since LRU is not performance-effective for CMPs, the NRU replacement policy that closely approximates LRU is not performance-effective for the CMP SLLC management either.

Recently, Jaleel et al. [8] have proposed a high-performance practical replacement policy called RRIP (an acronym for *Re-Reference Interval Prediction*). With 2 bits in the RRPV field, a block can have any of the three different categories of re-reference intervals: *near* (RRPV=0 or 1), *long* (RRPV=2) and *distant* (RRPV=3). RRIP always predicts a long re-reference interval for incoming blocks in an effort to prevent the cache pollution due to a subset of incoming blocks being dead-on-fill. Additionally, the bimodal variant of RRIP (called BRRIP) can prevent thrashing by predicting a distant (or a long) re-reference interval for an incoming block with a high (or a complementarily low) probability. TA-DRRIP is a thread-aware extension of RRIP to CMPs with SLLCs by coordinating either RRIP or BRRIP for individual threads under set-dueling and feedback control. Rooted in the same 2-bit RRPV substrate, SHiP, proposed in the most recent work [9], assigns either a distant or a long re-reference interval to an incoming block, depending on whether or not it is predicted to be dead-on-fill. Specifically, SHiP leverages a history table and sample sets to dynamically learn which memory instructions (identified by their PC signatures) tend to insert dead-on-fill blocks, and predicts a distant (or a long) re-reference interval for new blocks if they are inserted by those PCs (or otherwise).

2.3 Our New Perspective and Its Supporting Experimental Evidence

If we apply the same categorization in Section 2.1 to TA-DRRIP and SHiP, they are both classified as alternative replacement policies but for practical use in that they aim to optimize locality for SLLCs. While the alternative replacement policies excel in locality management, they are likely unable to coordinate the best capacity provisioning among all co-scheduled threads for utility optimization. This is due to their lack of *utility monitors* [6], a critical component in judicious capacity partitioning, which estimate how many SLLC hits each thread would deliver with various capacity allocated. Therefore, one of our research motivation lies in the question of whether or not locality optimization alone can provide high enough performance for practical SLLC capacity management. The answer to this question, to be shortly backed with workload characterization and performance comparison, is *no*, suggesting that locality and utility co-optimization is indispensable to the best utilization of SLLC resources. Further, if locality and utility co-optimization is out of necessity, another key question is whether or not the minimum-overhead 1-bit RRPV substrate is sufficient for such a purpose. The experiments de-

tailed in the following provide an affirmative answer to this question.

2.3.1 Workload Characterization

Before elaborating on the workload characterization, we supplement a little more background on the *bimodal insertion policy* (BIP) [10] that is a thrashing-prevention replacement alternative to LRU. Based on the log A-bit RRPV substrate, BIP assigns $A-1$ (or 0) to the RRPV of an incoming block with a high (or complementarily low) probability, in contrast to LRU's constant 0-RRPV assignment for new blocks. LRU and BIP are the two basic optional replacement policies used in *thread-aware dynamic insertion policy* (TADIP) [2] that functions for locality optimization. Given that NRU is shown to closely approximate LRU, we are motivated to propose a new practical replacement policy, called *bimodal NRU* (BNRU), which approximates BIP by filling 1 (or 0) into the RRPV of an incoming block with a high (or complementarily low) probability.

Figure 2 illustrates the SLLC performance in terms of *hits per thousand instructions* (HPKIs), for 8 of the benchmarks in our study, as a function of assigned cache capacity, managed by LRU, BIP, NRU and BNRU respectively (see Section 3 and Section 4 for more details of experimental setups). Here, with fixed 2048 sets and 64B lines assumed, we can measure the capacity in terms of the associativity. Looking at the performance aspects of the 4 policies in Figure 2, we can make the following observations: (1) the NRU and LRU hit curves overlap each other nearly completely for all of the 8 figures, indicating that NRU approximates LRU almost perfectly regardless of the SLLC capacity and associativity configurations; (2) the BNRU and BIP hit curves also match each other very well, except for the benchmark *facerec*, in a variety of SLLC configurations; (3) BNRU is as capable as BIP for thrashing prevention, as evidenced in Figure 2 (a)-(c) where the BNRU and BIP hit curves are higher than the NRU and LRU hit curves for the three benchmarks *omnetpp*, *sphinx3* and *mcf* within certain ranges of SLLC capacity configurations. In addition, *the three observations also hold for other benchmarks in our study* (see Table 2).

From the perspective of benchmark characteristics, the 8 benchmarks can be divided into four classes depending on their locality and utility features. The first two classes represent the cases where the performance can be improved with extra capacity allocated, but they differ in their NRU vs. BNRU utility. The last two classes saturate in performance after a minimal allocation of capacity, but with very different hit rates. In the first class, as indicated in Figure 2 (a)-(c), benchmarks *omnetpp*, *sphinx3* and *mcf* all have inferior locality because their NRU curves are significantly below the BNRU curves within certain capacity ranges (e.g., from associativity 2 to 20 for *mcf*). If any of them runs in a CMP with a mix of co-scheduled threads, an alternative replacement policy such as BNRU can potentially improve the SLLC hit performance over NRU. These are the cases where locality optimization can come into more prominent play for SLLC management. For example, with the replacement policy simply altered from NRU to BNRU for the same allocated capacity of 8 cache ways, the SLLC hit performance of *mcf* can be improved by 2.5x ($\approx \frac{42.4-12.2}{12.2}$).

In contrast, the workloads in the second class, represented by applications *astar*, *facerec* and *swim* (illustrated in Figure 2 (e)-(g)), show good locality since their NRU curves are never below the BNRU curves. However, they can still be set apart from each other with respect to their utility. For instance, when assigned 16 ways, *astar* has a higher utility than *facerec* in that *astar* can yield a 22.53 HPKI (corresponding to a hit rate of 87.3%) while *facerec* can deliver only a 6.2 HPKI (with a hit ratio of 40.0%). If a CMP workload consists of applications all from this category, much less room is available for locality improvement that alternative replacement policies are good at, while utility optimization is still likely to make a difference in performance by favoring threads with higher utility in SLLC capacity partitioning (e.g., preferring *astar* to *facerec*).

Figure 2 (d) and (h) illustrate the third and the fourth classes whose applications require very few SLLC resources. In particular, *milc* is a streaming application due to its high miss rate regardless of the amount of allocated SLLC capacity, while *crafty* is CPU-bound and can yield very high hit rates given a small amount of allocated SLLC capacity.

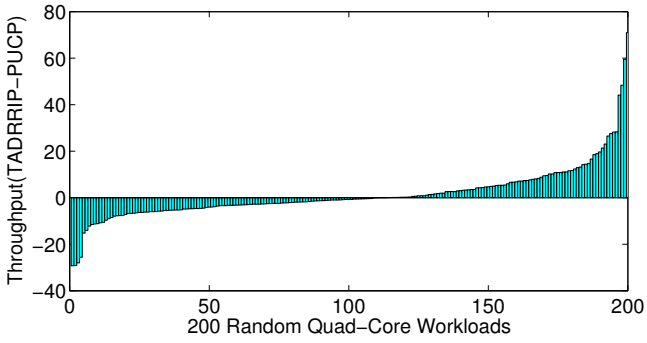


Figure 3: The difference between TA-DRRIP and PUCP in throughput normalized to that of LRU for individual quad-core workloads. Overall, TA-DRRIP and PUCP improve the throughput over LRU by 4.53% and 3.56% respectively. But, TA-DRRIP and PUCP have different performance comfort and discomfort zones due to their fundamentally distinct working principles and optimization objectives of locality vs. utility.

2.3.2 Performance Comparison

The workload characterization experiments above indicate that (i) NRU and its bimodal variant BNRU are competent for favoring good locality and preventing thrashing respectively, and (ii) locality optimization alone cannot work consistently well for heterogeneous CMP workloads consisting of threads with various locality and utility features. To quantitatively demonstrate the limitation of practical alternative replacement policies that are oriented towards locality optimization only, we compose a simple *practical utility-based cache partitioning* (PUCP) scheme and compare it against the locality-oriented TA-DRRIP on 200 random quad-core workloads (see Section 4 for experimental details). In essence, PUCP makes cache-way partitioning decisions for co-scheduled threads by relying on the per-core sampling-based LRU utility monitors (the same as in [6]) and then leverages the NRU replacement policy (instead of LRU in [6]) to manage each thread's allocated ways. More related details will be presented in Section 3. The final performance comparison result is that TA-DRRIP and PUCP improve the throughput

performance over LRU by 4.53% and 3.56% on average respectively, which indicates TA-DRRIP is better than PUCP overall. However, if we investigate Figure 3 that illustrates the difference in throughput normalized to that of LRU between TA-DRRIP and PUCP for individual workloads (sorted in ascending order), we can clearly see that there are 112 out of the 200 workloads for which TA-DRRIP underperforms PUCP. Since PUCP does nothing beyond utility optimization with its LRU-based utility monitors and the 1-bit NRU substrate, the detailed view of performance difference in Figure 3 verifies our speculation on the limitation of practical alternative replacement policies with locality management alone. But utility optimization alone as is provided by PUCP is not sufficient either, since TA-DRRIP outperforms PUCP for the remaining 88 workloads, which also contributes to TA-DRRIP's better overall performance in spite of its performance disadvantage for the 112 workloads. In summary, we can infer from this motivational experiment that neither locality nor utility optimizations alone can consistently perform well under a variety of workloads for practical SLLC capacity management due to their different working principles and optimization objectives.

3. DESIGN & IMPLEMENTATION

Our practical scheme, called COOP (an acronym for *locality and utility CO-OPtimization*) is designed to achieve three specific goals: (*i*) to base the SLLC capacity management on the 1-bit RRPV substrate; (*ii*) to be aware of locality and utility features of individual threads by profiling their NRU and BNRU hit curves; and (*iii*) to decide on the SLLC optimal management policy by conducting interactive locality and utility co-optimization for all co-scheduled threads.

3.1 The Overall Architecture

Figure 4 depicts an architectural view of COOP. In the SLLC, every cache line has a single bit in its RRPV field. Associated with each core, there is a *locality & utility monitor* that dynamically profiles both NRU and BNRU hit curves from the core's SLLC reference sequence. In particular, the NRU and the BNRU hit curves are captured by an LRU profiler and a BNRU profiler respectively, both of which are based on *auxiliary tag directories* (ATD) and the *set sampling* technique [10]. On every time interval boundary, the profilers feed the locality and utility information about sampler sets to the *decision unit*. Based on the information, the decision unit makes and enforces the capacity management decisions of cache-way partitions and replacement policies for all co-scheduled threads during the next time period.

3.2 The Locality & Utility Monitor

The *locality & utility monitor* counts the SLLC hits that a thread would contribute if it were running alone, while the amount of space assigned to it and the replacement policy (NRU vs. BNRU) adopted to manage its allocated space are both varied. By doing so, the monitor attempts to capture the runtime interplay between locality and utility optimizations in the SLLC management. Assuming that an SLLC has an associativity of 64, for example, the monitor counts the number of hits a thread would contribute if it were allocated 1-, 2-, ..., or 64-way SLLC space, being managed by NRU and BNRU respectively. Consequently, the monitor is able to deduce both NRU and BNRU hit curves as functions of cache ways, as illustrated in Figure 2 and generalized in Figure 5. The two curves can jointly convey two critical pieces of information:

- Which replacement policy should be adopted under a given capacity quota for the thread. As depicted in Figure 5, if the thread can get 4 cache ways, then it should apply the NRU replacement policy to manage the given amount of space, since the NRU hit curve (solid) is above the BNRU curve (dotted) when the way count equals 4; but if the assigned way-count is 12, the thread should alter the policy to BRNU that can help it obtain far more hits. Therefore, with the two curves, COOP can implicitly derive a *composite hit curve* (bold) which consists of the higher segments of the NRU or BNRU curves, as illustrated in Figure 5.
- What the preferred utility is under the best replacement policy. For instance, if the hit counts of the derived composite hit curve at the way counts of 10 and 12 are assumed to be 100 and 110 respectively, then we know that the utility of 10 ways is better than that of 12 ways because $\frac{100}{10} > \frac{110}{12}$. In this way, COOP fully exploits the interactions between locality and utility dimensions.

As described below, we apply two different profiling mechanisms to deduce the NRU and the BNRU hit curves of a thread respectively because of their distinct features.

Profiling an NRU Hit Curve: Since NRU and LRU hit curves have been experimentally shown to be almost identical, as exemplified in Figure 2, we approximate an NRU hit curve with its corresponding LRU hit curve, which can be easily obtained by the well-established *LRU utility monitor* (LRU UMON) [6]. In an LRU UMON, an *auxiliary tag directory* (ATD) with an associativity A and a size-A array of *stack-hit counters* are adopted to implement Mattson's *LRU stack algorithm* [11], where A is also the SLLC's set associativity. Here, an ATD structure, with each of its entries containing only a hashed tag, a valid bit and an $\log A$-bit RRPV field mimics the LRU stack of a small group of sampler SLLC sets, as if the monitored thread were exclusively occupying the whole space of these sampler sets. Upon every hit on ATD, it reports the LRU-stack position where the hit takes place so that the corresponding stack-hit counter $h(i)$ can be incremented by one. As a result of the LRU stack property, the values of the NRU and the LRU hit curves at way count i, denoted $NH(i)$ and $LH(i)$ respectively, can be expressed by Equation 1, where $h(k)$ is the hit counter at LRU-stack position k and $1 \leq k \leq i \leq A$.

$$NH(i) \approx LH(i) = \sum_{1 \leq k \leq i} h(k) \quad (1)$$

Profiling a BNRU Hit Curve: The profiling of a BNRU hit curve, on the other hand, is more challenging because BNRU violates the stack property with its nondeterministic 0-valued or 1-valued RRPV assignment for incoming blocks. Thus, the simple stack algorithm cannot be applied to deducing a BNRU hit curve. To resolve this issue, we first propose an exact but complex approach and follow it with an approximate but practical solution.

The *exact approach* is also based on set sampling, and uses a number A of ATD structures representing the A different associativities from 1 to A. Therefore, in the exact approach, we use a group of A ATD structures, {ATD(1), ATD(2),

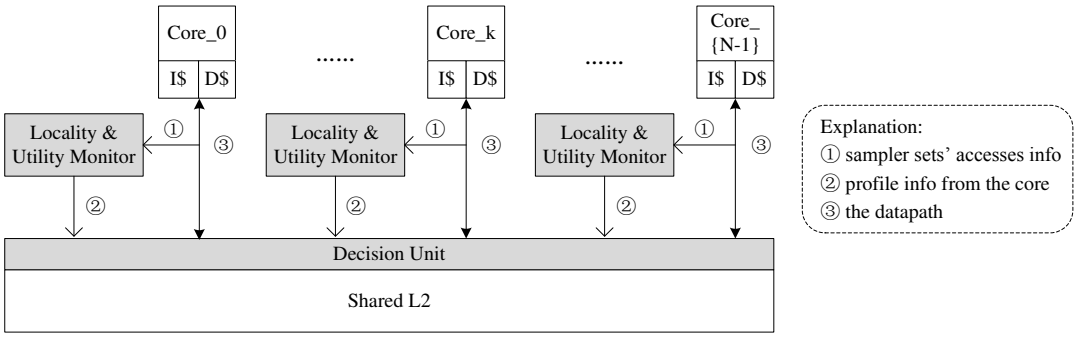


Figure 4: The COOP architecture: the grey boxes are the major extra hardware logic required by COOP on top of the SLLC with 1-bit RRPVs.

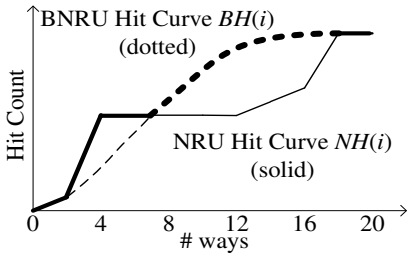


Figure 5: Deriving a *composite hit curve* (bold): the higher segments of either NRU (solid) or BNRU (dotted) hit curves.

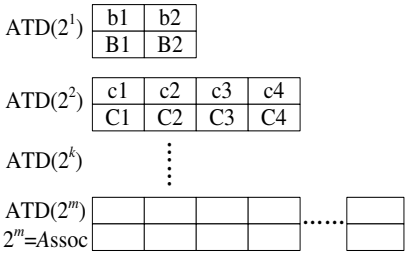


Figure 6: Approximately profiling a BNRU hit curve: a 2^k-way ATD structure ATD(2^k) is used for monitoring at each logarithmic way-count point 2^k, where $1 \leq k \leq \log A$.

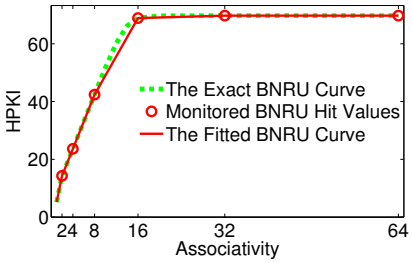


Figure 7: Applying the *logarithmic-distance monitoring & curve fitting* approach to profiling the BNRU hit curve for benchmark *mcf* (by approximating the exact curve).

ATD(3) ..., ATD($A-1$) and ATD(A)}, to mimic BNRU's operations on the sampler SLLC sets with an associativity ranging from 1 to A respectively. Although this approach provides an exact measure of the BNRU curve, it requires a significant number A of ATD structures, rendering the implementation prohibitively expensive when A is large.

The *practical solution* is based on four key observations derived from an analysis of the BNRU hit curves for the benchmarks in our study (exemplified in Figure 2): (*i*) the BNRU hit curve is monotonically non-decreasing with respect to the assigned way count; (*ii*) the BNRU hit curve is a concave function, which means that the curve's gradient is always non-increasing as the way count increases. *The intuition behind concave BNRU curves is that, with the high-probability 1-valued RRPV assignment for an incoming block, the block will most likely be victimized due to its 1-valued RRPV upon a subsequent cache miss in the same set, thus preventing other blocks from getting evicted* (namely, *stationary*).; (*iii*) the BNRU hit curve has a long flat tail as the way count approaches a high value; and (iv) it is provable that BNRU and NRU hit curves have the same value at way count 1 ($BH(1) = NH(1)$), since neither of them lets an incoming line bypass the cache. Therefore, it is sufficient to monitor the BNRU hit values at a small number of discrete logarithmic way-count points by using a dedicated ATD for each of these points, and then apply the *curve fitting* technique to deducing the entire BNRU hit curve. Specifically, as illustrated in Figure 6, we employ $m = \log A$ ATD structures {ATD(2^1), ATD(2^2), ..., ATD(2^m)} to capture the BNRU hit counts {$BH(2^1)$, $BH(2^2)$, ..., $BH(2^m)$} in a small number of way-count cases {2^1, 2^2, ..., 2^m}. We carry out curve fitting based on the m discrete BNRU hit values by linearly interpolating between the two monitored BNRU curve counts (2^k, $BH(2^k)$) and (2^{k+1}, $BH(2^{k+1})$) for $1 \leq k \leq m-1$. Then, the $BH(i)$ value can be calculated iteratively by Equation 2. Figure 7 shows an example of applying our *logarithmic-distance monitoring* & *curve fitting* approach with up to 64 ways for benchmark *mcf*.

$$BH(i) = \begin{cases} BH(2^k), \text{ where } i = 2^k \text{ and } BH(2^k) \\ \qquad \text{monitored by ATD}(2^k) \\ BH(i+1) - \Delta, \ \Delta = \frac{BH(2^{k+1}) - BH(2^k)}{2^k} \\ \qquad \text{and} \quad 1 \leq 2^k < i < 2^{k+1} \leq 2^m = A \end{cases} \tag{2}$$

The specific design choice of monitoring at logarithmic way-count points stems from our empirical observations mentioned above, suggesting a denser number of monitoring points to more accurately profile the high-gradient portion of the BNRU hit curve when A is small, which is also a property of a logarithmic/geometric series. As a result, the practical solution needs only $m = \log A$, instead of A, BNRU-managed ATD structures at the associativities of 2, 4, ..., $\frac{A}{2}$ and A respectively, as well as m BNRU-hit counters. It is worth remarking that the storage overhead (measured in the total number of ATD ways) required by the practical BNRU profiling is $2 + 4 + 8 + \cdots + \frac{A}{2} + A = \frac{2\times(A-1)}{2-1} = 2 \times (A-1) < 2 \times A$, *which is less than twice the storage overhead required by a single A-way ATD structure for the NRU/LRU hit curve profiling and makes our solution very practical in hardware implementation.* It must be noted

Table 2: Selected Benchmarks & Classification

Class	Descriptor	Benchmarks
I	Poor Locality	galgel, libquantum, mcf, omnetpp, sphinx3, xalancbmk
II	Good Utility	astar, ammp, bzip2, calculix, facerec, GemsFDTD, swim, twolf, vpr
III	Streaming	lucas, milc
IV	CPU-Bound	crafty, eon, fma3d

that, (i) the RRPV field of each BNRU-managed ATD entry has only a single bit, and (ii) upon an access to one sampler SLLC set, the LRU-managed ATD and the m BNRU-managed ATDs are operated concurrently to profile both NRU and BNRU hit curves.

3.3 The Decision Unit

With the locality and utility characteristics of co-scheduled threads profiled during each time interval, the decision unit will periodically determine the optimal space partitions and replacement policies for individual threads. Since the space partitioning logic of COOP is also utility-based and targeted at maximizing the overall performance, we adopt but with modification the framework of the *lookahead utility-based cache partitioning algorithm* [6] in the decision unit. The original algorithm evaluates every potential partitioning decision and provisions cache ways to a thread that currently has the highest utility of these ways based on its LRU hit curve. We modify the algorithm to determine the best utility-based partitioning of the cache ways according to the *composite hit curve*, which is composed of the higher segments of the NRU and the BNRU hit curves.

In the decision unit, there is an m-bit *partition quota counter*, an A-bit *global replacement mask* [12] and a single *locality management* bit associated with each core, where $m = \log A$ and A is the associativity. On each time interval boundary, the SLLC's space partitioning result for $Core_i$ is kept in the *partition quota counter*, denoted as Q_i, where $0 \leq i \leq N-1$. Assuming that A is greater than N, COOP also guarantees that at least one way is provisioned to every core. The *global replacement mask* is used to specify which cache ways are currently allocated to the corresponding core. For example, if a core is allocated with two cache ways, say, Way 0 and Way 1, only the first and the second bits on its *global replacement mask* are set to one. A core can access any lines in an SLLC set but is only allowed to replace a line in its own allocated ways. The *locality management* bit, LM_i, is utilized to indicate whether the NRU (LM_i=0) or BNRU (LM_i=1) policy is adopted for the core to manage its allocated SLLC cache ways. LM_i can be determined by examining the difference between the NRU and BNRU curves at the way count k that is also the value of Q_i: the bit is set 0 if $NH(k) \geq BH(k)$ or 1 otherwise.

4. EVALUATION

In this section, we first briefly describe our simulation-based experimental methodology and then present and analyze the evaluation results.

4.1 Evaluation Methodology

Simulation Setup: We simulate our scheme using the cycle-accurate M5 full system simulator [13] with the key configuration parameters listed in Table 1. We model a quad-core CMP with two levels of on-chip caches. The L1 instruction and data caches adopt a coupled tag & data store organization. For the shared L2 cache, we model decoupled tag and data stores for each L2 slice and also account for the NoC latency when calculating the L2 access time. The SLLC capacity management schemes in comparison include LRU (baseline), NRU, PUCP, TA-DRRIP, SHiP and our proposed COOP scheme. PUCP and COOP make management decisions periodically every 5M cycles [3], and their profiler hit counters are reset upon each periodic decision boundary. We have not found any overflow problems with these 16-bit hit counters in our experiments.

Performance Metrics: We adopt two standard metrics, *throughput* and *fair speedup*, to quantify the CMP performance. Specifically, *throughput* measures the utilization of a system, while *fair speedup* balances both performance and fairness. Let IPC_i be the *instructions per cycle* performance of the ith thread when it is co-scheduled with other threads and $SingleIPC_i$ be the IPC of the same thread when it executes in isolation. Then, for a system where N threads execute concurrently, the formulas for the two metrics are shown in Equation 3 and Equation 4 respectively.

$$\text{throughput} = \sum_{i=1,2,\cdots,N} IPC_i \tag{3}$$

$$\text{fair speedup} = \frac{N}{\sum_{i=1,2,\cdots,N} SingleIPC_i / IPC_i} \tag{4}$$

Workload Construction: As listed in Table 2, we select 20 benchmarks from the SPEC CPU 2000 and 2006 benchmark suites and categorize them into four classes by investigating their locality and utility features via experiments similar to Section 2.3.1. Class I is a collection of benchmarks that exhibit poor locality and can benefit from judicious replacement policies. The benchmarks in Class II have excellent utility and need dedicated SLLC space partitions. Class III is a group of streaming applications that require little SLLC capacity and need to be prevented from polluting the SLLC. Finally, Class IV benchmarks are CPU-bound with small working sets in the SLLC. From the four classes of benchmarks, 200 random quad-core workloads are generated by randomly selecting 200 4-benchmark combinations out of the 20 individual benchmarks.

Simulation Control: In the experiments, all threads under a given workload are executed starting from a checkpoint that has already had the first 10 billion instructions bypassed. They are cache-warmed with 1 billion instructions and then simulated in detail until all threads finish another 1 billion instructions. Performance statistics are reported for a thread when it reaches 1 billion instructions. If one thread completes the 1 billion instructions before others, it continues to run so as to still compete for the SLLC capacity, but its extra instructions are not taken into account in the final performance report. This is in conformation with the standard practice in CMP cache research [2, 3, 5–9].

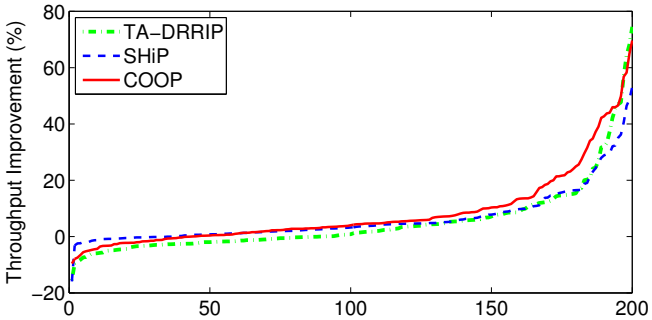
4.2 Performance Comparison

Figure 8 shows the geometric mean of throughput performance for NRU, PUCP, TA-DRRIP, SHiP and COOP,

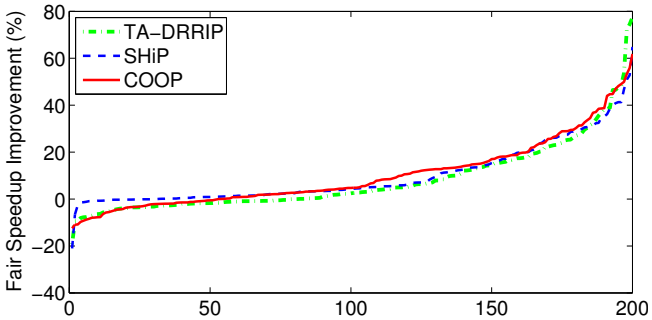
[3]The period is experimentally tuned.

Table 1: Major Configuration Parameters

Core	four cores, Alpha ISA, in-order, IPC=1 except for memory accesses, 128/128 I/D TLBs, 44-bit physical addresses
L1	2-way, 32KB, 1-cycle delay, 16 MSHRs, write back & 4 write buffer entries for L1D, LRU-managed
L2	16-way, 4MB, 6/8-cycle tag/data store delay, totally 1024 MSHRs, write back & totally 256 write buffer entries
	mesh NoC topology ($1 \times 2, 2 \times 2, 2 \times 4$) with 1 cycle delay per hop; 300-cycle off-chip memory access delay
Schemes	#sample_sets = $\frac{1}{32} \times$ #SLLC_sets for all sampling-based schemes; BNRU's low probability = $\frac{1}{32}$; PUCP/COOP's LRU-managed ATD: 16-bit hit counters, 10-bit hashed tags, 4-bit RRPVs, 1 valid bit; TA-DRRIP: 10-bit saturating counters; SHiP: 16K 3-bit saturating counters in the history table, 14-bit PC signatures + 1 reuse bit in each sampled cache line; COOP's BNRU-managed ATDs: 16-bit hit counters, 10-bit hashed tags, 1-bit RRPVs, 1 valid bit.

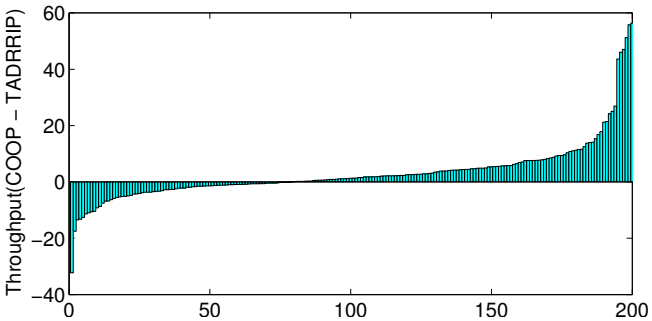


(a) **4-Core Throughput**

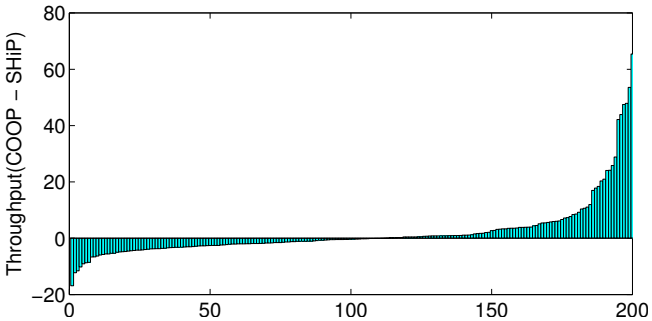


(b) **4-Core Fair Speedup**

Figure 9: Detailed views of the quad-core throughput and fair speedup improvement for TA-DRRIP, SHiP and COOP. The performance values on each curve are sorted in ascending order.



(a) **Comparison between COOP and TA-DRRIP**



(b) **Comparison between COOP and SHiP**

Figure 10: The difference between COOP and the two state-of-the-art practical replacement policies in throughput normalized to that of LRU for individual quad-core workloads.

normalized to the baseline (LRU) on the simulated quad-core configuration. Averaged over all of the 200 random workloads, COOP provides a throughput improvement of 7.67%, which is noticeably higher than the improvements achieved by the practical replacement policies (TA-DRRIP: 4.53%, SHiP: 6.00%) and the simple utility-oriented scheme (PUCP: 3.56%). The NRU replacement policy degrades the throughput performance by 0.24%, which indicates that both NRU and LRU are clearly inadequate for CMPs with SLLCs. If we look closer at the details of throughput improvement in Figure 9(a), we can find that the worst case performance of COOP is -9.57% and its best improvement is up to 69.67%, while the (worst, best) performance margins for TA-DRRIP and SHiP are (-15.47%, 74.49%) and (-15.90%, 53.21%) respectively. Figure 9(a) also shows that the throughput performance curve of COOP is almost always above that of TA-DRRIP except for a very small fraction of the 200 workloads (i.e., a very small x-axis range at the end). COOP's curve is also above that of SHiP except for a small fraction of the 200 workloads (i.e., a small x-axis range at the beginning), which means that COOP offers a consistent and robust throughput performance that is generally better than the two existing practical replacement policies.

Figure 10(a) and Figure 10(b) show the head-to-head comparison between COOP and the two state-of-the-art schemes for individual workloads. COOP outperforms the two practical replacement policies for the majority of the workloads, and in many cases significantly. However, it does not do so in all cases. We speculate that it is because COOP must strike a balance between both locality and utility optimizations. COOP may not optimize locality fully when a workload heavily or exclusively favors locality optimization but does show stronger performance improvement over both of the state-of-the-arts (indicated by the portions of bars above the 0/equal line in Figure 10(a) and Figure 10(b)).

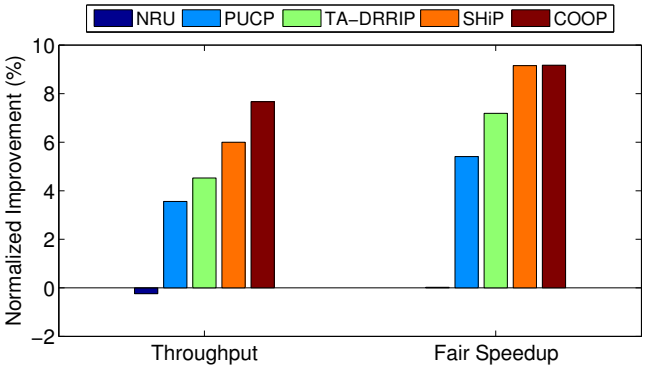


Figure 8: The 4-Core Configuration: the throughput and the fair speedup improvement (over LRU, in geometric mean) averaged over the 200 random quad-core workloads for all of the schemes in comparison.

If we specifically compare Figure 10(a) against Figure 3 in Section 2.3.2, we can make the following two observations: (*i*) COOP significantly boosts the performance beyond capacity partitioning provided by PUCP by additionally adapting to the better replacement policy between NRU and BNRU; (*ii*) it relies on its partitioning module besides the alternative replacement mechanism, which is based on 1-bit RRPVs, to surpass a TA-DRRIP-like replacement policy which only conducts locality optimization even with 2-bit RRPVs. Therefore, COOP is arguably capable of bridging the performance gap between the locality and the utility optimization schemes via its co-optmization strategies.

In terms of the fair speed improvement, COOP improves over LRU by 9.17%, which matches the 9.16% performance improvement of SHiP. Other schemes, NRU, PUCP and TA-DRRIP, improve the fair speedup metric by 0.02%, 5.41%, 7.19% respectively. Figure 9(b) depicts the detailed result of fair speedups for individual workloads. While the curves of COOP and SHiP are both above that of TA-DRRIP in most cases, the difference between the curves of the two bests seems to be minor.

However, we notice in both Figure 9(a) and Figure 9(b) that SHiP's curves are barely below the zero horizontal line, while COOP's curves have a small portion below. This may suggest that SHiP is slightly more robust in the sense that it seldom underperforms LRU. But this robustness may come at the cost of its weakened ability to exploit the highest performance when available, which is evident in Figure 9(a) because the curves of both COOP and TA-DRRIP are above that of SHiP at the end of the x-axis range. TA-DRRIP seems to be able to exploit the highest performance, but it underperforms LRU in quite a few cases. In this sense, COOP also shows its ability to strike a reasonable balance between the exploitation of the highest possible performance and keeping performance robust in the worst cases.

4.3 Overhead Analysis

Table 3 gives a detailed comparison of hardware overhead for the various schemes in this paper (see their specific configurations in Table 1). For all of the schemes, the total number of their RRPV bits as well as the extra monitoring logic is counted into their total hardware costs. In particular, for COOP, since it requires an LRU profiler and an BNRU profiler for per-core locality & utility monitoring, the ATD structures will dominate hardware overhead in its design. But the hashing function that has shown provably low collision rate in [14] and the *logarithmic-distance monitoring* technique in the BNRU profiler contribute to much less cost in the per-core locality & utility monitor. Most importantly, COOP is based on the minimum-overhead 1-bit RRPV substrate, which can greatly help reduce the overhead from the 2-bit RRPV substrate in both TA-DRRIP and SHiP in spite of COOP's extra monitoring logic. As a result, COOP's 3.88KB LRU profiler, 5.86KB BNRU profiler and 8KB RRPVs contribute to its 17.74KB total overhead. In contrast, the recently-proposed practical scheme SHiP, requires more hardware resources due to the large prediction history table (6KB) and also additional PC signature stores (3.75KB), in addition to the 16KB 2-bit RRPVs. Overall, COOP can provide better performance improvement than both TA-DRRIP and SHiP but at a comparable cost of storage overhead to TA-DRRIP and a lower cost than SHiP.

Table 3: Overhead & Throughput

	LRU	NRU	TA-DRRIP	SHiP	COOP
RRPVs	32KB	8KB	16KB	16KB	8KB
Monitors	0	0	5B	9.75KB	9.74KB
Total Overhead	32KB	8KB	16KB	25.75KB	17.74KB
Throughput	1	0.9976	1.0453	1.0600	1.0767

5. RELATED WORK

As our paper focuses on the SLLC capacity management, in what follows, we briefly review the representative work related to the shared LLC organization.

5.1 SLLC Capacity Management

Theoretical Alternative Replacement Policies: As LRU is ineffective in handling workloads with inferior locality, alternative replacement policies have been proposed to adapt management decisions to workloads' specific locality characteristics, by means of sophisticated block insertion, promotion or victimization. The *thread-aware dynamic insertion policy* (TADIP) [2] identifies the locality of individual threads through set-sampling and dueling, and then coordinates locality optimization for all of the co-scheduled threads under feedback control. The *next-use cache* (NUcache) [5] scheme differentiates the locality of individual memory instructions, selects a small group of instructions with superior locality through a cost-benefit analysis and allows their inserted blocks to stay longer in the SLLC by the FIFO replacement policy. Based on LRU-managed set samples and skewed dead-block prediction tables, the *sampling dead block prediction* (SDBP) scheme [3] learns which memory instructions (identified by their PC signatures) tend to access cache blocks that immediately become "dead", victimizes the blocks touched by those PCs prior to default replacement candidates and also bypasses predicted dead-on-fill blocks.

Theoretical Capacity Partitioning Schemes: The commonly used LRU policy implicitly divides the SLLC capacity among competing threads on a miss-driven basis, which is also ineffective in that a thread may occupy much capacity by bringing into the cache a number of missed blocks but without re-referencing them. SLLC capacity partitioning is targeted

at allocating LLC resources to threads on a *utility*, *fairness* or *quality-of-service* (QoS) basis. The *utility-based cache partitioning* (UCP) [6] and the *pseudo insertion/promotion partitioning* (PIPP) schemes are proposed to partition the SLLC space among co-scheduled threads to maximize the throughput performance. With the thread-level utility information profiled by *LRU utility monitors* (LRU UMON) that are based on Mattson's *LRU stack algorithm* [11], UCP partitions SLLC space in the form of assigning cache ways to threads and favors those with higher utility under LRU, while PIPP achieves a similar effect by relying on a combination of insertion and promotion policies. In [15], Kim et al. propose fairness-based cache partitioning so that all threads are slowed down equally from where each thread monopolizes the SLLC. In [16], Nesbits et al. introduce the notion of virtual private caches to facilitate QoS guarantees. Zhou et al. [17] argue that the SLLC miss count, which is a commonly used metric in the literature, is inadequate for QoS considerations. They propose to take into account the measures of both miss count and miss penalty of each thread when performing QoS-oriented SLLC partitioning.

5.2 Hit-Latency Reduction

In the context of *non-uniform cache architecture* (NUCA) [18], the SLLC organization can incur higher hit latencies, because the block being accessed by a core may reside in a non-local SLLC slice. *Victim replication* (VM) [19] reduces SLLC hit latency by retaining/replicating each core's L1 victim blocks in its local SLLC slice if a victim's home slice is non-local. Noticing that blind replication would diminish the effective SLLC capacity and in turn adversely affect performance, the *adaptive selective replication* (ASR) [20] mechanism selectively replicates shared read-only data and adaptively determines the replication level that minimizes hit latency without an obvious increase in cache misses.

5.3 Page-Coloring Management

Page coloring is an OS-based approach to cache management by manipulating an overlapped section (namely the *page color*) between a physical page index and an SLLC set index. First proposed in [21], by using the physical page index rather than the set index in determining a block's home LLC slice, OS-assisted SLLC management is shown to have a direct and flexible impact on CMP's SLLC hit latency and sharing degrees. Lin et al. [22] propose to partition the SLLC capacity to optimize throughput, fairness or QoS for CMPs by allocating page colors to different cores. Awasthi et al. [23] propose to utilize shadow address bits in migrating shared pages to optimal locations. Chaudhuri [24] devises a HW mechanism to support the decision-making on when and where to migrate an entire page so as to amortize the performance overhead. A recent SLLC design called *reactive NUCA* (R-NUCA) [25] classifies SLLC accesses into distinct categories and adapts data placement, replication and migration to individual categories via page coloring.

6. CONCLUSION

The research community has introduced a substantial volume of theoretical proposals to optimize either locality or utility in the SLLC capacity management. But their high storage overhead for re-reference interval prediction discourages the industry from adopting them in practical CMP designs. Although there are already two practical replacement policies TA-DRRIP and SHiP that significantly reduce overhead by relying on a 2-bit RRPV substrate, their performance is suboptimal due to their single-pronged approach of locality optimization. Different from the existing studies, our proposed COOP design (*i*) combines the strengths of both the minimum-overhead 1-bit RRPV substrate and the profilers with good theoretical traits and, importantly, (*ii*) carries out locality and utility co-optimization in capacity management. By employing lightweight monitors that profile both NRU (approximated by LRU) and BNRU hit curves (curve-fitted with *logarithmic-distance monitoring*), our proposed design can exploit the co-optimized locality and utility of concurrent threads and thus effectively manage the SLLC capacity for CMP workloads with heterogeneous resource requirements. Our execution-driven simulation shows that the proposed scheme improves the throughput performance over the baseline LRU for 200 random workloads by 7.67% on a quad-core CMP with a 4MB SLLC, outperforming both TA-DRRIP (4.53%) and SHiP (6.00%), all at the cost of only 17.74KB storage overhead that is slightly higher than TA-DRRIP (16KB) but lower than SHiP (25.75KB).

The present work opens up opportunities for future research. For instance, our current proposed design is evaluated using multiprogrammed workloads for which the heterogeneous capacity requirements of co-scheduled threads are the most important concern. But for multithreaded workloads, the interactions between data sharing and capacity sharing among threads need to be taken into account in the SLLC management [26]. In our future work, the performance impact of COOP on multithreaded workloads will be investigated, for the purpose of devising an amended scheme that incorporates enhancements to promote both data and capacity sharing.

7. ACKNOWLEDGMENTS

We would like to thank the Holland Computing Center of the University of Nebraska - Lincoln for providing the supercomputing facilities. We also greatly appreciate the constructive comments and suggestions from the anonymous reviewers. This work was supported in part by NSF under Grants CCF-0937993, IIS-0916859, CNS-1016609 and CNS-1116606.

References

[1] Iñigo Goiri, Josep Oriol Fitó, Ferran Julià, Ramon Nou, Josep Lluis Berral, Jordi Guitart, and Jordi Torres. Multifaceted Resource Management for Dealing with Heterogeneous Workloads in Virtualized Data Centers. In *Proceedings of the 11th ACM/IEEE International Conference on Grid Computing*, pages 25–32, October 2010.

[2] Aamer Jaleel, William Hasenplaugh, Moinuddin K. Qureshi, Julien Sebot, Simon C. Steely Jr., and Joel S. Emer. Adaptive Insertion Policies for Managing Shared Caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 208–219, October 2008.

[3] Samira M. Khan, Daniel A. Jiménez, Doug Burger, and Babak Falsafi. Using Dead Blocks as a Virtual Victim Cache. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 489–500, September 2010.

[4] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 151–162, December 2010.

[5] R. Manikantan, K. Rajan, and R. Govindarajan. NUcache: An Efficient Multicore Cache Organization Besed on Next-Use Distance. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, pages 243–254, February 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 *Proceedings of the 39th IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, December 2006.

[7] Yuejian Xie and Gabriel H. Loh. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. In *Proceedings of the 36th International Symposium on Computer Architecture*, pages 174–183, June 2009.

[8] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel Emer. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *Proceedings of the 37th International Symposium on Computer Architecture*, pages 60–71, June 2010.

[9] 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 *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 430–441, December 2011.

[10] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., and Joel Emer. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 381–391, June 2007.

[11] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM System Journal*, 9(2):78–117, June 1970.

[12] Derek Chiou, Srinivas Devadas, Larry Rudolph, Boon S. Ang, Derek Chiouy, Derek Chiouy, Larry Rudolphy, Larry Rudolphy, Srinivas Devadasy, Srinivas Devadasy, Boon S. Angz, and Boon S. Angz. Dynamic Cache Partitioning via Columnization. In *MIT CSAIL Computation Structures Group Memo 430*, pages 1–22, November 1999.

[13] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, July 2006.

[14] M. V. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient Hardware Hashing Functions for High Performance Computers. *IEEE Transactions on Computers*, 46:1378–1381, December 1997.

[15] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, September 2004.

[16] Kyle J. Nesbit, James Laudon, and James E. Smith. Virtual Private Caches. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 57–68, June 2007.

[17] Xing Zhou, Wenguang Chen, and Weimin Zheng. Cache Sharing Management for Performance Fairness in Chip Multiprocessors. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 384–393, September 2009.

[18] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated on-Chip Caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, March 2002.

[19] Michael Zhang and Krste Asanovic. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 336–345, June 2005.

[20] Bradford M. Beckmann, Michael R. Marty, and David A. Wood. ASR: Adaptive Selective Replication for CMP Caches. In *Proceedings of the 39th IEEE/ACM International Symposium on Microarchitecture*, pages 443–454, Decebmer 2006.

[21] Sangyeun Cho and Lei Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *Proceedings of the 39th IEEE/ACM Internationl Symposium on Microarchitecture*, pages 455–468, December 2006.

[22] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In *Proceedings of the 14th International Symposium on High Performance Computer Architecture*, pages 367–378, Febuary 2008.

[23] Manu Awasthi, Kshitij Sudan, Rajeev Balasubramonian, and John B. Carter. Dynamic Hardware-Assisted Software-Controlled Page Placement to Manage Capacity Allocation and Sharing within Large Caches. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, pages 250–261, February 2009.

[24] Mainak Chaudhuri. PageNUCA: Selected Policies for Page-Grain Locality Management in Large Shared Chip-Multiprocessor Caches. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, pages 227–238, February 2009.

[25] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *Proceedings of the 36th International Symposium on Computer Architecture*, pages 184–195, June 2009.

[26] Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen. Does Cache Sharing on Modern CMP Matter to the Performance of Contemporary Multithreaded Programs? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 203–212, January 2010.