CLU: Co-Optimizing Locality and Utility in Thread-Aware Capacity Management for Shared Last Level Caches

Dongyuan Zhan, Hong Jiang, and Sharad C. Seth, *Fellow*, *IEEE*

Abstract—Most chip-multiprocessors nowadays adopt a large shared last-level cache (SLLC). This paper is motivated by our analysis and evaluation of state-of-the-art cache management proposals which reveal a common weakness. That is, the existing alternative replacement policies and cache partitioning schemes, targeted at optimizing either locality or utility of co-scheduled threads, cannot deliver consistently the best performance under a variety of workloads. Therefore, we propose a novel adaptive scheme, called *CLU*, to interactively co-optimize the locality and utility of co-scheduled threads in thread-aware SLLC capacity management. CLU employs lightweight monitors to dynamically profile the LRU (*least recently used*) and BIP (*bimodal insertion policy*) hit curves of individual threads on runtime, enabling the scheme to co-optimize the locality and utility of concurrent threads and thus adapt to more diverse workloads than the existing approaches. We provide results from extensive execution-driven simulation experiments to demonstrate the feasibility and efficacy of CLU over the existing approaches (TADIP, NUCACHE, TA-DRRIP, UCP, and PIPP).

Index Terms—Capacity management, chip multiprocessors, locality and utility co-optimization, shared last level caches

1 Introduction

A shared last level cache (SLLC) organization is commonly used in chip multiprocessor (CMP) designs to simplify cache capacity sharing and coherence support for processing cores. Most commodity CMPs nowadays, whether multi-core (e.g., AMD's Phenomtm II X6 and IBM's Power 7) or many-core (e.g., Tilera's 100-core processors [1]), have large SLLCs to help retain a substantial amount of data on-chip. But a large aggregate capacity alone does not guarantee optimal performance without an effective SLLC management strategy. This is especially true when the cores are running a heterogeneous mix of applications/threads, as is increasingly common with the widespread deployment of CMPs in complex application environments such as virtual machines and cloud computing [2].

Because of its vital importance to the system performance, SLLC capacity management has been extensively studied. We categorize these studies into two groups: those proposing alternatives to the LRU replacement policy [3]–[8] and those proposing cache partitioning schemes [9], [10].

Because the commonly-used LRU replacement policy aims to favor cache access recency (or *temporal locality*[1]) 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) [11]. Alternative replacement policies, such as TADIP [3] and NUCACHE [8], are proposed to overcome the thrashing problem by judiciously assigning and adjusting lifetimes for cached blocks.

The *utility* of a thread represents its ability to reduce misses with a given amount of SLLC capacity [9]). 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 recent studies, such as UCP [9] and PIPP [10], propose to partition the SLLC space among competing threads based on the utility information captured by per-thread LRU-stack profilers, notably improving the performance over the baseline LRU replacement policy.

In our view, the state-of-the-art alternative replacement policies and cache partitioning schemes have fundamentally different working principles. Specifically, the alternative replacement policies (of TADIP and NUCACHE) determine how the competing cores should temporally share the SLLC capacity to accommodate workloads' locality, while the cache partitioning schemes (of UCP and PIPP) decide on how the SLLC resources should be spatially divided among the cores on a utility basis.

Our analysis and evaluation show that alternative replacement policies and cache partitioning schemes represent essentially two independent dimensions of solving the overall shared cache management problem, and that optimizing in just one dimension misses the benefits available from co-optimization in both. Specifically, the alternative replacement policies, lacking a utility monitor, cannot coordinate the best capacity provisioning for all of the co-scheduled threads, while the cache partitioning schemes, fail to realize

1. In our paper, locality is specifically referred to as temporal locality.

• *The authors are with the Department of Computer Science and Engineering, University of Nebraska–Lincoln, Lincoln, NE 68588. E-mail: dyzhan@gmail.com, {jiang, seth}@cse.unl.edu.*

Manuscript received 24 Dec. 2011; revised 25 Oct. 2012; accepted 01 Nov. 2012. Date of publication 25 Nov. 2012; date of current version 27 June 2014. Recommended for acceptance by N. Ranganathan. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TC.2012.277

opportunities for higher utility achievable by individual threads with an replacement policy other than LRU. In order to gain a deeper understanding of this issue, we characterize the locality and utility features for a spectrum of workloads and construct different workload combinations to evaluate the existing solutions. Our observations confirm distinct performance comfort zones for the two categories of existing approaches, neither performing consistently and robustly well under all workloads.

Motivated by these observations, we propose a novel design, called CLU, to interactively co-optimize the locality and utility of workloads in thread-aware SLLC capacity management. The key design challenge is how to estimate the utility information with a replacement policy other than LRU. Based on the observation that the hit curve of the thrashing-prevention policy BIP (*bimodal insertion policy* [11]) is concave and can be approximated by using logarithmic samples, CLU employs two lightweight runtime monitors for each thread in a CMP workload: a classic LRU stack profiler and a novel *logarithmic-distance-curve-fitting* BIP utility profiler to capture the interleaved locality and utility of the thread. Leveraging the information about all co-scheduled threads, CLU spatially partitions the SLLC cache ways among the threads and temporally makes use of the allocated capacity for individual threads in an interactive way, so that the highest utility provided by the best replacement policies can be exploited. Our evaluation shows that CLU improves the throughput by 24.3%, 45.3% and 43.3% for our simulated dual-core, quad-core and eight-core systems (with 0.26%, 0.27% and 0.53% storage overhead) respectively, outperforming the existing alternative replacement policies and cache partitioning schemes under a wide-range of CMP workloads.

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

2 Background & Motivation

Although the entire SLLC capacity can be accessed by all cores, allowing free accesses with the LRU replacement policy does not necessarily lead to an effective utilization of the SLLC resources. This is because regulating the contention for capacity among co-scheduled threads is beyond the capability of LRU. Therefore, various alternative replacement policies and cache partitioning schemes have been proposed for a better utilization of the SLLC capacity. Here, we briefly describe their working principles and discuss their strengths and weaknesses revealed by our experiments, which motivates us to view the SLLC capacity management from a unique perspective (Section 5 places our research in a broader context of related work).

2.1 SLLC Capacity Management

Locality-Oriented Alternative Replacement Policies: It has been noted that the LRU replacement policy performs quite well when a thread's block-reuse distance is no longer than its cache set associativity [11], or in other words, when the thread has excellent locality. However, LRU can cause a thread with poor locality to thrash its cache space [11] or severely interfere with other co-scheduled threads in capacity use [3]. In general, the thrashing problem can be solved by adaptively assigning the lifetime of a block according to the locality of the thread that brings it into the cache. Following this general principle, *feedback-directed thread-aware dynamic insertion policy* (TADIP-F[2]) [3] differentiates between individual threads' locality characteristics through set-sampling and dueling, and then leverage feedback-directed learning to adopt either LRU to exploit locality or BIP (*bimodal insertion policy*) to prevent thrashing for each thread. BIP [11] inserts an incoming block at the MRU position of the target set with a low probability p and the LRU position with the high probability $1 - p$. In contrast, the *Next-Use Cache* (NUCACHE) scheme [8] makes use of the locality characteristics of individual load instructions (identified by their PCs) and selects a small group of these instructions through a cost-benefit analysis, allowing their blocks to stay longer in the SLLC by a new (FIFO) replacement policy.

Utility-Oriented Capacity Partitioning Schemes: The *utility* is defined as the ratio of the SLLC hit count to the SLLC capacity that is required to maintain the hit count for a thread under the LRU replacement policy. The miss-driven nature of the LRU-based SLLC capacity management implicitly partitions the SLLC capacity among co-scheduled threads in a way that a thread incurring more misses will be allocated a greater amount of SLLC capacity by default. But the miss-driven capacity allocation is oblivious of a thread's efficiency of utilizing the SLLC resources for performance delivery, exemplified by the pathological case where a streaming thread occupies a large amount of capacity with little performance contribution. The work by Suh et al. [12] is the first proposal that uses marginal-gain counters in LRU to count the hits at different stack positions and estimates the utility information for each application. Later, the *utility-based cache partitioning* (UCP) [9] and *pseudo insertion/promotion partitioning* (PIPP) [10] schemes are proposed to partition the SLLC space among co-scheduled threads according to their utility. With the utility information profiled for all co-scheduled threads by *utility monitors* (UMON) which are based on *Mattson's LRU stack algorithm* [13], UCP partitions SLLC space in the form of cache ways among threads and favors those with higher utility under LRU, while PIPP achieves the same effect by relying on a combination of insertion and promotion policies.

2.2 Our Perspective and Supporting Experimental Data

In our view, the aforementioned alternative replacement policies and cache partitioning schemes have fundamentally different working principles: the replacement alternatives aim to temporally optimize the sharing of SLLC capacity for co-scheduled threads mainly by preventing those with poor locality from thrashing the SLLC, while the cache partitioning schemes are targeted at spatially provisioning SLLC resources among competing threads according to their LRU-based utility characteristics. Unfortunately, the replacement policies are unable to coordinate the best capacity provisioning for all co-scheduled threads due to the lack of utility monitors, while the existing cache partitioning schemes cannot estimate or

2. In our paper, TADIP is TADIP-F by default.

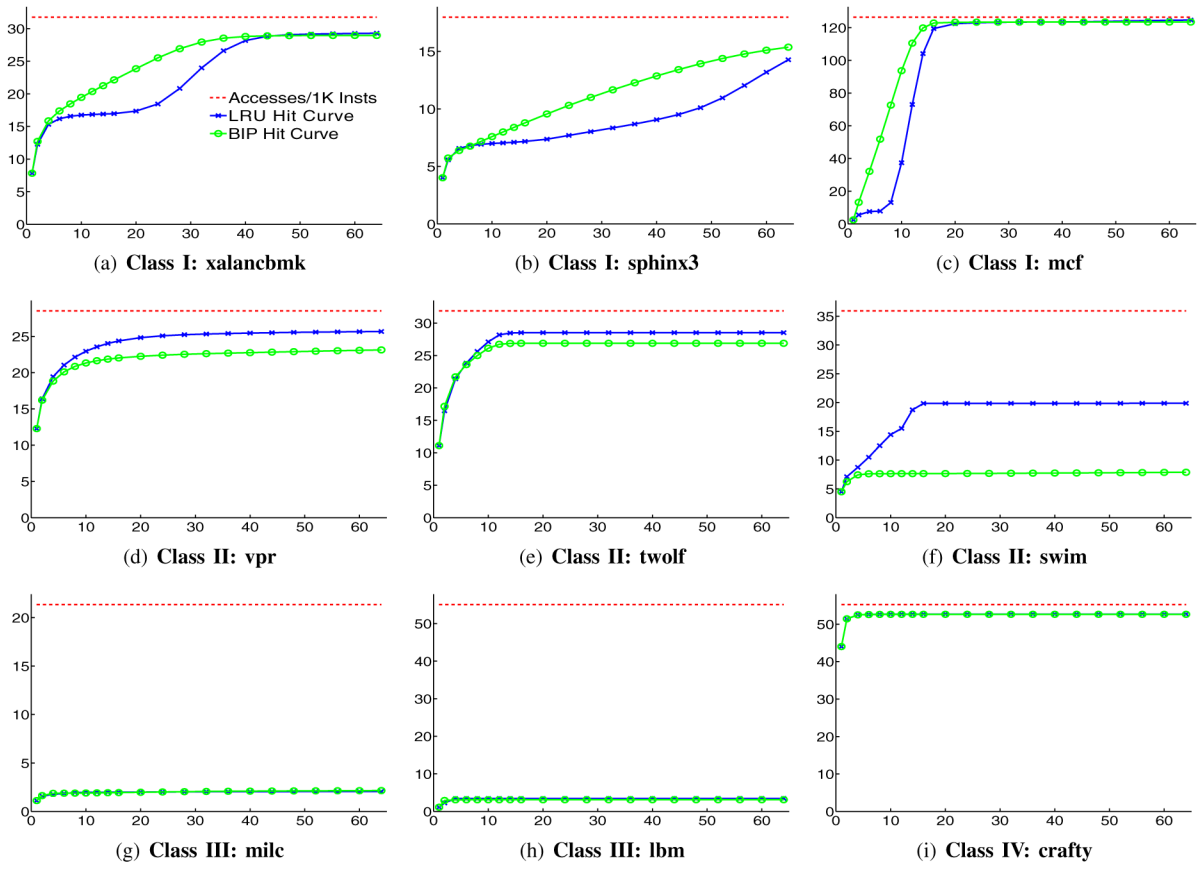


Fig. 1. The HPKIs of LRU and BIP as a function of the LLC capacity for the SPEC benchmarks. The x-axis shows the LLC capacity measured in the number of ways (given that the number of sets and line size are fixed), while the y-axis represents *hits per 1 K instructions*. The dotted roofline in each figure indicates the total number of LLC accesses per 1k instructions (independent of the LLC capacity). The 9 benchmarks are divided into four classes according to their locality and utility characteristics.

exploit the utility information for a replacement policy alternative to LRU. As a result, focusing on optimizing locality or utility alone in SLLC capacity management, the two categories of approaches miss delivering robust performance under a variety of workloads. In the following, we elaborate on why it is beneficial to treat locality and utility concurrently and interactively. Our argument is based on workload characterization as well as an evaluation of the two categories of approaches on the workloads that expose their performance comfort and discomfort zones.

Fig. 1 illustrates the LLC performance for 9 of the benchmarks in our study as a function of assigned cache capacity, managed by LRU and BIP respectively (see Sections 3 and 4 for more details). Here, with fixed 2048 sets and 64B lines assumed, we can measure the capacity in terms of the associativity. The 9 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 allocation of extra capacity, but they differ in their LRU vs. BIP 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 Fig. 1(a)–(c), benchmarks *xalancbmk*, *sphinx3* and *mcf* all have inferior locality because their LRU curves are significantly below the BIP curves within a certain capacity range (e.g., from associativity 2 to 20 for *mcf*). If any of them runs in a mix of co-scheduled threads on a CMP, an alternative replacement policy such as TADIP can potentially apply an alternative replacement decision better than LRU to improve the SLLC hit performance. In contrast, existing cache partitioning schemes like UCP are oblivious of locality due to their LRU-based utility monitors. For instance, if a cache partitioning scheme decides to allocate 8 cache ways to *mcf*, without the locality information, the scheme will never realize that *mcf*'s hit performance can still be improved by 4.5x ($\approx \frac{72.6-13.1}{13.1}$) with the same capacity allocation by simply altering the replacement policy from LRU to BIP.

In contrast, the workloads in the second class, represented by applications *vpr*, *twolf* and *swim* (illustrated in Fig. 1(d)–(f)), show good locality since their LRU curves are never below the BIP curves. However, they can still be set apart from each other with respect to their utility. For instance, when assigned 16 ways, *twolf* has a higher utility than *swim* in that it can yield 28.5 hits per 1 K instructions (HPKI) (corresponding to a hit rate of 95.2%) while *swim* can deliver only 19.8 HPKI (with a hit ratio of 55.2%). Further, if *twolf* and *swim* are running concurrently and compete for the SLLC resources such as the 16-way SLLC, an alternative replacement policy like TADIP-F will detect LRU's better hit performance than BIP (especially for *swim*) and thus adopt the LRU module for both of them. But since *swim* inherently has many more misses than *twolf* (e.g., the ratio between the MPKIs of *swim* and *twolf* are 5.0 and 11.5 at associativity 8 and 16 respectively), *swim* will occupy much greater capacity than *twolf* due to the underlying

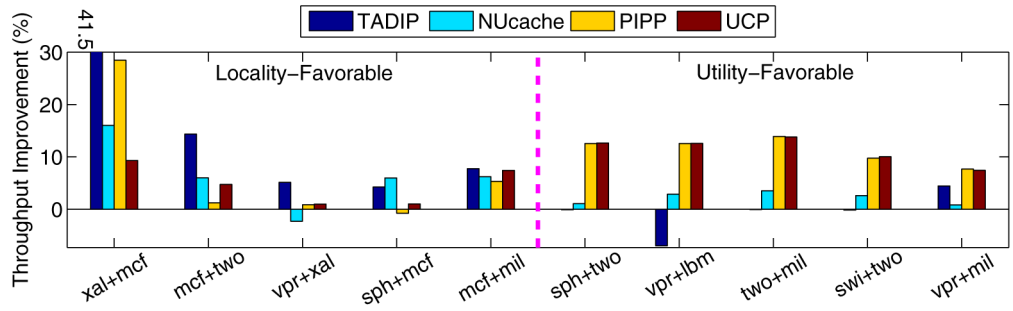


Fig. 2. Motivational experiments: an alternative replacement policy like TADIP or NUCACHE can outperform a cache partitioning scheme such as PIPP or UCP under locality-favorable workloads, while a cache partitioning scheme is better than an alternative replacement policy under utility-favorable workloads.

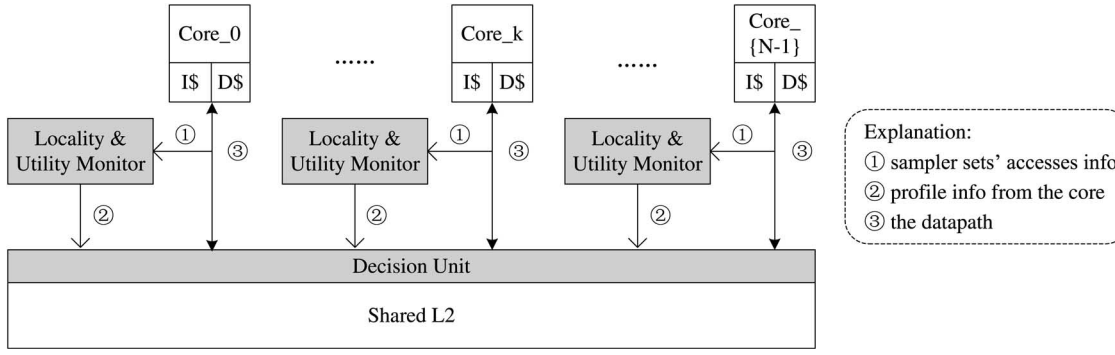


Fig. 3. The CLU architecture: a *locality & utility monitor* captures the interplays between the locality and the utility optimizations for each thread, while the *decision unit* determines and enforces the co-optimized space partitions and replacement policies for all of the co-scheduled threads. I$ and D$ within each core represent L1 instruction and data caches respectively. The grey boxes are the major extra hardware logic required by CLU on top of the baseline LRU-based SLLC.

miss-driven capacity allocation by LRU. A cache partitioning scheme such as UCP or PIPP, being utility-aware, can do a better space partitioning in this case by favoring *twolf*.

Fig. 1(g)–(i) illustrate the third and the fourth classes whose applications require very few SLLC resources. In particular, *milc* and *lbm* are both streaming applications due to their high miss rates, while *crafty* is CPU-bound and can yield very high hit rates given a small amount of SLLC capacity.

To better understand the performance impact of the different working principles between alternative replacement policies and cache partitioning schemes, we construct 10 simple dual-core CMP workloads by pairing some of the benchmarks illustrated in Fig. 1 to expose their performance gaps. We then use the workloads to evaluate the alternative replacement policies TADIP and NUCACHE, as well as the cache partitioning schemes PIPP and UCP, on a dual-core CMP with a 16-way 2 MB SLLC in terms of their throughput improvement over the baseline LRU (see the experimental setup details in Section 4). Fig. 2 shows that, based on their throughput performance, the ten workloads can be divided into two categories, namely *locality-favorable* and *utility-favorable*. For a locality-favorable workload that consists of at least one of the benchmarks with inferior locality, e.g., *xalancbmk+mcf*, an alternative replacement policy like TADIP can greatly optimize the temporal capacity-sharing behavior for co-scheduled threads, which a cache partitioning scheme often fails to do. On the other hand, a utility-favorable workload consists of benchmarks with significantly diverse utility (e.g., *swim+twolf*) such that a cache partitioning scheme can make a better decision on space partition, yielding a better performance than an alternative replacement policy. We also note that an alternative replacement policy like TADIP performs much worse in certain utility-favorable workloads like *sphinx3+twolf*, even though *sphinx3* presents opportunities for locality improvement. This is because *twolf* begins to interfere with *sphinx3* when they are managed by LRU and BIP respectively, due to the lack of a dedicated space partition for performance isolation in TADIP. In summary, we can infer from this motivational experiment that neither alternative replacement policies nor cache partitioning schemes can consistently perform well under a variety of workloads due to their different working principles.

3 Design & Implementation

CLU is designed to achieve three specific goals: (i) to be thread-aware, which means that it should be able to differentiate between the diverse features of individual threads; (ii) to dynamically profile both utility and locality of co-scheduled threads and fully exploit the interactions between the two dimensions for co-optimization; and (iii) to decide on the optimal management policy by taking into account the locality and utility characteristics of all the threads.

3.1 The CLU SLLC Architecture

Fig. 3 depicts an architectural view of CLU. On an N-core CMP, a *locality & utility monitor* is associated with each core and dynamically captures both the utility and locality information of the SLLC access sequence from its host core. In particular, the locality & utility monitor consists of an LRU

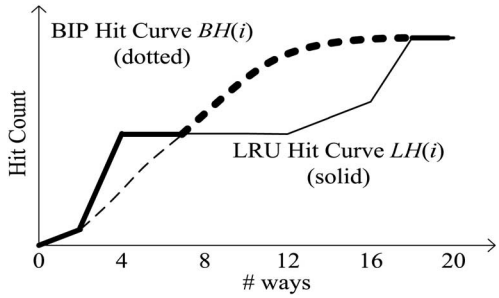


Fig. 4. Deriving a *composite hit curve* (the bold/higher segments): $DH(i) = \max\{LH(i), BH(i)\}$, where $1 \leq i \leq A$ and A is the associativity. Specifically, $LH(i)$ and $BH(i)$ are the values of LRU (solid) and BIP (dotted) hit curves at way count i respectively.

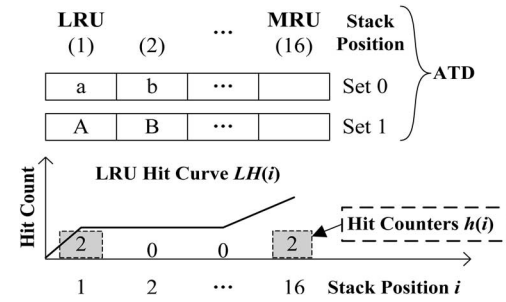


Fig. 5. Profiling a *LRU hit curve*: using an A-way *auxiliary tag array* and applying *Mattson's stack algorithm*.

profiler and a BIP profiler, both of which are based on the set sampling technique [11]. Therefore, only a small group of sampler sets out of all SLLC sets are monitored and the samplers' information is used to deduce the characteristics of the entire SLLC. On every time interval boundary, the profilers feed the information back to the *decision unit* that uses it to determine the space partitioning and replacement policy for all of the 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 it is assigned and the replacement policy (LRU vs. BIP) adopted to manage the allocated space are both varied. By so doing, the monitor attempts to capture the runtime interplay between the locality and the utility optimizations in 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 LRU and BIP respectively. Consequently, the monitor is able to deduce both the LRU and BIP hit curves that are a function of cache ways respectively, as illustrated in Fig. 1 and generalized in Fig. 4. 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 Fig. 4, if the thread can get 4 cache ways, then it should apply the LRU replacement policy to manage the given amount of space, since the LRU hit curve (solid) is above the BIP curve (dotted) when the way count equals 4; but if the assigned way-count is 12, the thread should alter the policy to BIP that can help it obtain far more hits. Therefore, with the two curves, CLU can implicitly derive a *composite hit curve* (bold) which consists of the higher segments of the LRU or BIP curves.
- 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, CLU fully exploits the interactions between the locality and the utility dimensions.

To be detailed next, we apply two different profiling mechanisms to respectively deduce the LRU and BIP hit curves of a thread, since LRU satisfies the stack property [13] while BIP does not. Specifically, the stack property stipulates that the blocks that would be in an A-way associative cache should be subsumed by those that would be in an $(A+1)$-way associative cache.

3.2.1 Profiling the LRU Hit Curve

To obtain the LRU hit curve, we leverage the well-established profiling technique [9] that leverages the *set sampling* strategy and *Mattson's stack algorithm* [13]. Specifically, an *auxiliary tag directory* (ATD) with an associativity A and a size-A array of *stack-hit counters* are adopted to implement Mattson's stack algorithm, where A is also the SLLC's set associativity, as shown in Fig. 5. Here, an ATD structure, with each of its entries containing only the tag 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 stack property of LRU, the value of the LRU hit curve at way count i, denoted $LH(i)$, can be expressed by Equation (1) and (2) shown at the bottom of the page.

3.2.2 Profiling the BIP Hit Curve

The profiling of the BIP hit curve, on the other hand, is more challenging because BIP violates the stack property by placing incoming blocks at the LRU position of any cache set with a high probability or at the MRU position with the complementary (low) probability. Thus, the simple stack algorithm cannot be applied to deducing the BIP hit curve. To resolve this

$$LH(i) = \sum_{1 \leq k \leq i} h(k), \text{where } h(k) \text{ is the hit counter at LRU-stack position } k \text{ and } 1 \leq k \leq i \leq A, \tag{1}$$

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

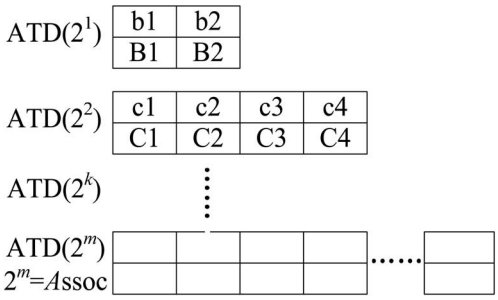


Fig. 6. Practically but approximately profiling a *BIP hit curve*: a 2^k-way ATD structure ATD(2^k) is used for monitoring at each logarithmic way-count point 2^k, where $1 \leq k \leq m = \log_2 A$.

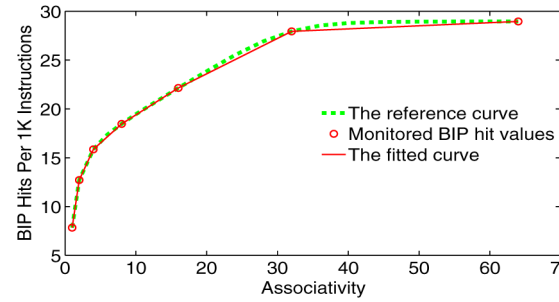


Fig. 7. An example of applying the logarithmic-distance monitoring and curve-fitting approach for profiling the BIP hit curve of benchmark *xalancbmk* (by approximating the exact reference curve).

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), ..., ATD($A-1$) and ATD(A)}, to mimic BIP's operations on the sampler SLLC sets with an associativity ranging from 1 to A respectively, where ATD(k) stands for an ATD structure with an associativity of k. By monitoring any ATD(k) structure, the corresponding $BH(k)$, namely the value of the BIP hit curve at way count k, can be determined as the total hit count of ATD (k) under BIP. Although this approach provides an exact measure of the BIP curve, it requires a significant number A of ATD structures, which makes the implementation prohibitively expensive when A is large, even when a single ATD structure is lightweight [9], [10].

The practical solution is based on four key observations derived from an analysis of the BIP hit curves for the benchmarks in our study (exemplified in Fig. 1): (i) the BIP hit curve is monotonically non-decreasing with respect to the assigned way count; (ii) the BIP 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 BIP curves is that, at non-LRU stack positions, the blocks hardly get evicted by incoming blocks (namely, stationary) and are also ranked from MRU to descending positions based on recency*; (iii) the BIP hit curve has a long flat tail as the way count approaches a high value; and (iv) it is provable that the LRU and BIP hit curves have the same value at way count 1 ($BH(1) = LH(1)$), since the LIP (LRU insertion policy) module in BIP does not let an incoming line bypass the cache [11]. Therefore, it is sufficient to monitor the BIP 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 deduce the entire BIP hit curve. As illustrated in Fig. 6, we employ m ATD structures {ATD(2^1), ATD(2^2), ..., ATD(2^m)} to capture the BIP hit counts {$BH(2^1)$, $BH(2^2)$, ..., $BH(2^m)$} in a small number of way-count cases $\{2^1, 2^2, \ldots, 2^m\}$}, where $m = \log_2 A$. We carry out curve fitting based on the m discrete BIP hit values by linearly interpolating between the two monitored BIP curve counts (2^k, $BH(2^k)$) and (2^{k+1}, $BH(2^{k+1})$). Then, the $BH(i)$ value can be calculated iteratively by Equation 2. Fig. 7 shows an example of applying our logarithmically discrete monitoring and curve-fitting approach with up to 64 ways for the benchmark *xalancbmk*. 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 BIP hit curve when A is small, which is also a property of a logarithmic/geometric series.

As described above, the practical solution needs only $m = log_2 A$, instead of A, BIP-managed ATD structures at the associativities of $2, 4, \ldots, \frac{A}{2}$ and A respectively, as well as m BIP-hit counters. It is worth remarking that the storage overhead (measured in the total number of ATD ways) required by the practical BIP 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 LRU profiling and makes our solution very practical in hardware implementation.* It needs to be noted that, upon an access to one sampler SLLC set, the LRU-managed ATD and the m BIP-managed ATDs are operated concurrently for both the LRU- and BIP-curve profiling.

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 partition and replacement policy for individual threads by leveraging on all their locality and utility information fed by the monitors. Since the space partitioning logic of CLU is also utility-based, aimed at maximizing the overall performance, we adopt the framework of the *lookahead utility-based cache partitioning algorithm* [9]. Here is a brief description of how the algorithm works. Assume that n threads are sharing A cache ways, and thread $0, 1, \cdots,$ and $n-1$ have already got $allocated\_capacity[0], allocated\_capacity[1], \ldots$ and $allocated\_capacity[n-1]$ respectively. Then, the remaining $A - \sum_{k=0}^{n-1} allocated\_capacity[k]$ ways need to be partitioned: for thread i, the lookahead algorithm will evaluate the peak marginal utility as well as the corresponding number of ways within the integer range $[allocate\_capacity[i], allocate\_capacity[i] + A - \sum_{k=0}^{n-1} allocated\_capacity[k]]$, where $0 \leq i \leq n-1$, and allocate the exact number of ways to one of the n threads that has the highest peak marginal utility; afterwards, the algorithm continues to allocate the remaining space iteratively until there are no available cache ways. The original algorithm works based on the LRU hit curve because of its stack property. We modify the algorithm to determine the

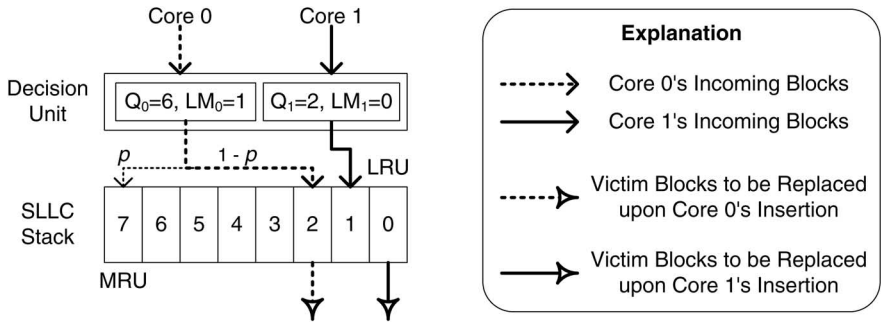


Fig. 8. An example of enforcing the management decisions.

best utility-based partitioning of the cache ways according to the composite hit curve, which is composed of the higher segments of the BIP and LRU hit curves. Other studies only examine the utility of an LRU hit curve, which has been shown to be ineffective in the case of poor locality in Section 2.

On each time interval boundary, the SLLC's space-partitioning result for $Core_i$ is kept in an m-bit *partition quota counter*, denoted as Q_i, where $0 \leq i \leq N-1$ and $m = log_2 A$. Assuming that A is greater than N, CLU also guarantees that at least one way is provisioned to every core, which means that initially $allocate\_capacity[i] = 1$ for $0 \leq i \leq n-1$. With each core $Core_i$ the *decision unit* in CLU also associates a (*locality management*) bit, LM_i, to indicate the LRU ($\mathrm{LM} = 0$) vs. BIP ($\mathrm{LM} = 1$) policy to be adopted for the core in its allocated SLLC space. LM_i can be determined by examining the difference between the LRU and BIP curves at the value k of the partition quota counter Q_i: the bit is set 0 if $LH(k) \geq BH(k)$ or 1 otherwise.

CLU enforces its space partitioning and replacement policy with specific promotion, insertion and victimization strategies. The *single-step promotion* policy [10] is adopted as CLU's cache-block promotion mechanism. As illustrated in Fig. 8, we assume that the SLLC's LRU stack is numbered $0, 1, \cdots, A-1$ from the LRU position to the MRU position. When a new block is brought in by $Core_i$, if its LM_i is 0, the LRU block of the target set is replaced and the incoming block is inserted at position $k-1$, where k is the value of Q_i. This part is similar to the promotion, insertion and victimization modules of PIPP [10]. On the other hand, if $Core_i$'s LM bit is 1, the block at position $A-k$ will be victimized, and the block brought in by the core will be inserted at position $A-k$ with a high probability and at the MRU position with the complementary low probability. Therefore, if a core's incoming block stream shows a poor locality (i.e. its LM bit is 1), part of its working set can be preserved well in its allocated space with the BIP-like victimization and insertion. Fig. 8 demonstrates a dual-core example with an 8-way SLLC managed by CLU: for $Core_1$, its incoming blocks are always placed at position 1 with the LRU blocks victimized, because it has good locality and gets a space quota of 2 SLLC ways; but for $Core_0$, since it exhibits inferior locality and is allocated with 6 ways, the blocks at position 2 ($= 8-6$) will be replaced upon insertion, and its incoming blocks will be inserted at the MRU position with a low probability (1/32 in our study and other BIP-related work [3], [11],) and at position 2 otherwise.

With respect to the time complexity, similar to existing cache partitioning approaches UCP and PIPP, the runtime performance overhead of CLU is negligible for the following reasons: (*i*) monitoring is in parallel and not intrusive with normal cache operations; (*ii*) every 5 million cycles, decision making is conducted in the background and not in the critical path of cache accesses; (*iii*) for A cache ways, curve fitting only involves addition, subtraction and shift operations (see Equation (2)), while deriving a composite curve just needs to compare LRU and BIP hit counts at each of the A way-points, and both of them can be accomplished in linear time; (*iv*) only several partition quota counters and locality management bits will be modified to embody the new management decisions, of which the time complexity is trivial; (*v*) decision enforcement only changes the timestamps of cache blocks in one set upon a cache hit, miss or fill, without moving or flushing a number of blocks.

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 [14] with the configuration parameters listed in Table 1. For the memory hierarchy, we model two levels of on-chip caches. The L1 instruction and data caches adopt the conventional set-associative configuration,

TABLE 1
Major Configuration Parameters

Core (2/4/8)	Alpha ISA, in-order, IPC=1 except for memory accesses, 128/128 I/D TLBs
L1	2-way, 32KB, 64B/line, 1-cycle delay, 16 MSHRs, write back & 4 write buffer entries for L1D
L2	64B/line, 6/8-cycle tag/data store delay, totally 1024 MSHRs, write back & totally 256 write buffer entries, physically tagged and indexed (M5's built-in setting), 2MB & 16 ways/4MB & 16 ways/8MB & 32 ways for 2/4/8-core configurations
NoC	mesh topology ($1 \times 2, 2 \times 2, 2 \times 4$) with 1 cycle delay per hop
Mem	300-cycle delay
Oth-ers	#(sample sets/core) = 32, BIP's low probability = 1/32, hit counter length = 16 bits, single-step promotion probability = 3/4, stream promotion probability = 1/128, NUCACHE L2 setups are the same as in [8]

TABLE 2
Selected Benchmarks & Classification

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

the LRU replacement policy, and a coupled tag-data store organization. For the shared L2 cache, we model decoupled tag and data stores for each L2 slice, and account for the NoC latency when calculating the L2 access time. Using representative and specially-constructed composite workloads, we first evaluate and compare the performance of LRU (baseline), TADIP, NUCACHE, UCP, PIPP and the proposed CLU for the dual/quad/eight-core configurations. TADIP reevaluates its management decisions whenever any saturating counter of its monitor has its MSB altered, while NUCACHE, PIPP, UCP and CLU make management decisions periodically every 5M cycles. The 16-bit profiler hit counters in PIPP, UCP and CLU are reset upon each periodic decision boundary, and we have not found any overflow problems with the counters in our experiments. Particularly, the aforementioned schemes are all rooted in the true-LRU environment that timestamps each cache block using $\lceil \log_2 Associativity \rceil$ bits. In essence, for the true-LRU based schemes, the representative and specially-constructed composite workloads are generated to expose the performance gap between the locality-oriented and utility-oriented approaches and demonstrate CLU's ability of bridging the gap. Then, we also use the random workloads to evaluate the general overall performance for all of the true-LRU based schemes, as well as the pseudo-LRU based approach TA-DRRIP [5] that is also an alternative replacement policy but uses only 2 bits per cache block for timestamp. In particular, our implementation of both TADIP and TA-DRRIP is based on the open-source code provided by their original authors (available at [15]).

Performance Metrics: We adopt two standard metrics of *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.

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

$$\text{fair speedup} = \frac{N}{\sum_{i=1,2,\ldots,N} \text{Single } IPC_i/IPC_i}. \tag{4}$$

Composite Workload Construction: As listed in Table 2, we select 19 benchmarks from the SPEC CPU 2000 and 2006 benchmark suites and categorize them into four classes according to their locality and utility. Class I is a collection of benchmarks that exhibit poor locality and can be improved by 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, we can construct dual/quad/eight-core workload mixes/combinations ("workloads" for short in the remainder of the paper unless explicitly noted otherwise) in Table 3, which can be further divided into locality-favorable and utility-favorable categories, shown in the top and the bottom halves, respectively, of each workload mix. Every locality-favorable workload consists of at least one Class I benchmark and should enable either TADIP or NUCACHE to outperform both capacity-partitioning schemes. On the contrary, for every utility-favorable workload, constructed using benchmarks with diverse utility, PIPP and UCP should achieve a better performance than the alternative replacement policies.

Simulation Control: In the experiments, all threads under a workload are executed starting from a checkpoint that has already had the first 20 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.

4.2 Performance Comparison Under Representative and Specially-Constructed Composite Workloads

Fig. 9 shows the throughput performance of TADIP, NUCACHE, PIPP, UCP and CLU normalized to the baseline (LRU) on the simulated dual-core configuration. For 18 dual-core workloads, CLU provides a throughput improvement of 24.3% on average (and up to 95.5%), which is much higher than the improvements by locality-oriented (TADIP: 14.9%, NUCACHE: 9.6%) and utility-oriented (PIPP: 15.0%, UCP: 7.2%) approaches. If we look closer at the specific categories of workloads, we can find that the higher improvements of CLU come from its capability of bridging the performance gap between alternative replacement policies and capacity partitioning schemes. Specifically, for the locality-favorable workloads MIX2_1-MIX2_9, the better alternative replacement policy TADIP, can improve their throughputs by 24.6% on average, while the better capacity partitioning scheme (PIPP here) can only yield 11.7% higher performance over the baseline, in contrast to CLU's 30.1%. In terms of the utility-favorable workloads MIX2_10-MIX2_18, however, PIPP and UCP can improve their performance by 18.3% and 12.7% respectively, while TADIP and NUCACHE only improve by 5.9% and 7.5%, in contrast to CLU's 18.8%. Therefore, while CLU outperforms all of the existing approaches throughout both locality-favorable and utility-favorable workloads, none of the locality-oriented or utility-oriented approaches can perform consistently well.

We can further explain the performance implications of existing approaches and CLU by means of an example. For the locality-favorable workload MIX2_1, which is the

TABLE 3
Workload Construction

MIX2	Apps	MIX4	Apps	MIX8	Apps
1	mil+omn	1	two+omn+lbm+mcf	1	2 sph + 2 omn + 2 mcf + 2 two
2	omn+xal	2	lbm+amm+mcf+omn	2	2 two + 2 vpr + 2 omn + 2 sph
3	xal+mcf	3	mcf+two+omn+sph	3	2 swi + 2 bzi + 2 luc + 2 omn
4	omn+fma	4	omn+two+mcf+gal	4	lib+sph+mcf+amm+swi+two+fma+mil
5	lbm+omn	5	amm+omn+two+mcf	5	2 swi + 2 mcf + 2 sph + 2 omn
6	mcf+two	6	xal+two+omn+mcf	6	2 mil + 2 vpr + 2 omn + 2 mcf
7	sph+mcf	7	two+swi+mcf+omn	7	2 bzi + 2 luc + 2 sph + 2 omn
8	fma+xal	8	fma+omn+lbm+swi	8	2 mcf + 2 swi + 2 bzi + 2 luc
9	xal+vpr	9	cra+swi+lbm+omn	9	2 omn + 2 mcf + 2 swi + 2 bzi
10	mcf+lbm	10	omn+lbm+gal+mcf	10	2 sph + 2 omn + 2 luc + 2 swi
11	swi+two	11	amm+omn+vpr+lib	11	gcc+sph+mcf+amm+vpr+omn+two+mil
12	lbm+xal	12	gal+amm+omn+vpr	12	two+gal+omn+mcf+gcc+lib+xal+vpr
13	vpr+lbm	13	mcf+omn+vpr+sph	13	omn+sph+mil+gcc+lib+two+swi+lbm
14	Gem+two	14	lbm+xal+fma+omn	14	vpr+swi+two+lib+fma+mcf+omn+xal
15	two+mil	15	mcf+lib+omn+amm	15	lib+sph+omn+gcc+two+xal+gal+lbm
16	lib+xal	16	omn+lib+xal+cra	16	2 omn + 2 swi + 2 two + 2 bzi
17	two+sph	17	two+xal+lib+omn	17	2 xal + 2 omn + 2 luc + 2 swi
18	xal+swi	18	mil+cra+omn+xal	18	2 sph + 2 xal + 2 omn + 2 mcf

combination *omnetpp+milc*, TADIP can provide a much better performance than both of the capacity partitioning schemes. Specifically, *milc* is a streaming application, and *omnetpp*, belonging to benchmark Class I, can be improved by judicious replacement policies. In this scenario, TADIP will adopt its BIP module to manage both threads so as to prevent thrashing for *milc* and significantly promote the performance of *omnetpp* by preserving a large part of its working set in the SLLC. PIPP and UCP cannot do as well as TADIP through SLLC partitioning alone (e.g., giving at least 15 ways to *omnetpp* in a 16-way 2 MB SLLC), because they cannot detect *omnetpp*'s being in Class I. So, PIPP and UCP will insert the incoming blocks of *omnetpp* at the high position of the SLLC stack (position 14 for PIPP and position 15/MRU for UCP), still thrashing its working set. However, with its locality & utility monitor and BIP-like insertion, CLU can match TADIP's performance. On the other hand, for a utility-favorable workload MIX2_11, *swim+twolf*, that exhibits diverse utility, no opportunities are present for locality-oriented improvement. Thus, PIPP and UCP can improve the performance by 9.7% and 10.0% respectively, better than TADIP (−0.2%) and NUCACHE (2.6%). Since CLU also has a utility-management module, it can improve the performance of this benchmark combination by 9.8%.

Figs. 10 and 11 present the performance comparison among the schemes for the quad-core and eight-core configurations. For both configurations, the gap between alternative replacement policies and cache partitioning schemes is similarly manifested by a significant impact on the performance. Again, CLU exploits the opportunities for performance improvement in both locality and utility dimensions interactively and provides 45.3% and 43.3% higher throughputs over the baseline for quad-core and eight-core systems respectively, significantly outperforming other approaches. It must be noted that CLU slightly underperforms some of the locality-oriented and utility-oriented approaches under a few workload combinations, such as the cases MIX4_1 and MIX8_17. This is because CLU is designed to strike a good balance/compromise between the locality and the utility optimizations. Therefore, it may not work as aggressively as a single-dimension management approach when the opportunities for performance optimization dominate in exactly one dimension for a workload. However, CLU can still win out robustly in a much broader range as a result of its adaptive

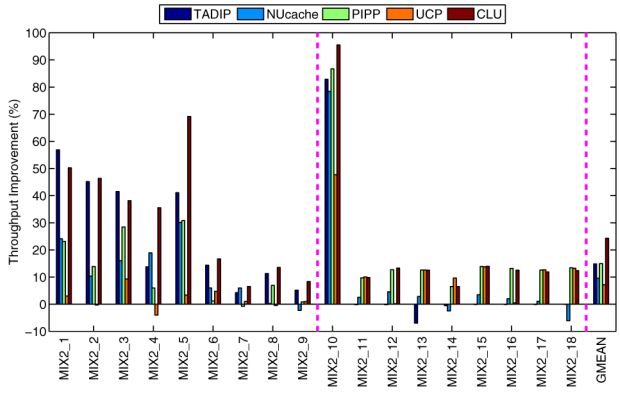


Fig. 9. Dual-core workloads: for the locality-favorable workloads (to the left of the first dotted line) CLU improves the baseline performance by 30.1%; the second-best is TADIP with a 24.6% improvement. For the utility-favorable workloads, between the two dotted lines, CLU, again, achieves the best overall performance, a 24.3% improvement vs. the second-best PIPP's 15.0%. The overall improvement for all workloads is indicated by the GMEAN bars in the figure.

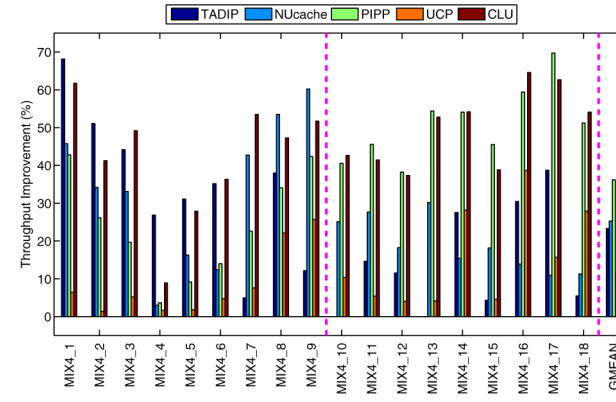


Fig. 10. Quad-core workloads: for the locality-favorable workloads, CLU achieves a 41.1% improvement, while the second best, TADIP, achieves a 33.4% improvement over the baseline. For the utility-favorable workloads, CLU's improvement is 45.3% vs. 36.2% for PIPP, which is the second-best. The overall improvement for all workloads is shown by the GMEAN bars in the figure.

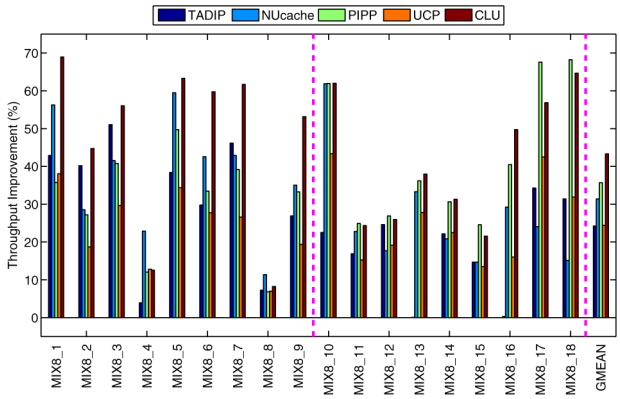


Fig. 11. Eight-core workloads: For the locality-favorable workloads, CLU, with 45.9% improvement, has the best overall performance vs. 37.1% for the next-best, NUCACHE. For the utility-favorable workloads, PIPP, with 41.4% improvement, narrowly outperforms CLU, with 40.7% improvement. Overall, CLU still provides a 7.6% improvement over the next best (PIPP) as indicated by the GMEAN bars in the figure.

management capabilities. More interestingly, when the improvement opportunities are significant in interactively exploiting both locality and utility for a workload, CLU is able to co-optimize the management decisions, leading to its superior performance to the existing approaches in these cases (e.g., MIX2_10, MIX4_16 and MIX8_1).

Fig. 12 illustrates the performance impact of different SLLC management schemes for the *fair speedup* metric. For the dual-, quad- and eight-core configurations, CLU outperforms the baseline by 23.1%, 43.0% and 36.5% on average (geometric mean) respectively, which are much better than any of the locality-oriented or the utility-oriented approaches (TADIP: 16.1%/25.0%/22.4%, NUCACHE: 11.4%/30.5%/32.6%, PIPP: 16.1%/34.7%/32.1%, UCP: 8.3%/13.0%/26.2%). This set of results reveal that not only can CLU provide higher absolute throughput but also it is able to improve on the fairness over the existing schemes.

4.3 Performance Comparison Under Randomly-Selected Workloads

So far, we have demonstrated CLU's capabilities of locality and utility co-optimization with representative and specially-constructed locality-favorable and utility-favorable workloads. Next, we show that CLU also performs well in

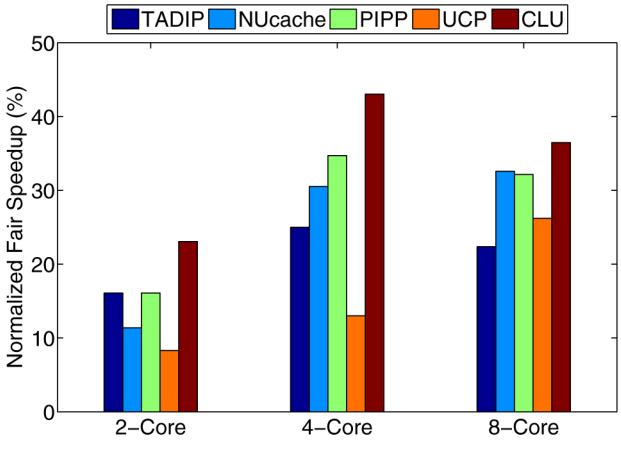


Fig. 12. Fair speedup improvement.

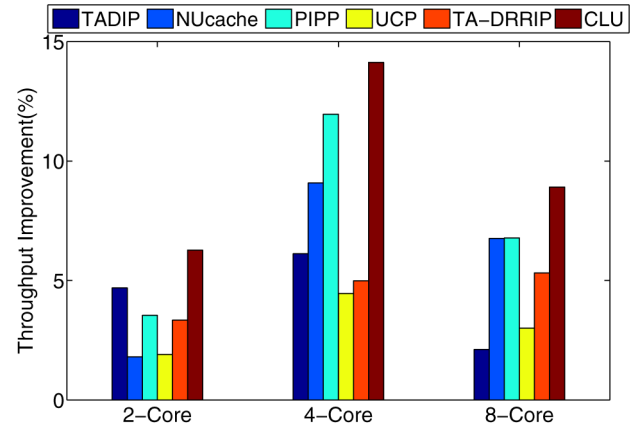


Fig. 13. Fifty random 2/4/8-core workloads.

randomly selected workload combinations. In particular, the pseudo-LRU based scheme TA-DRRIP that uses 2 bits ($2 < \lceil \log_2 Associativity \rceil$) per block for timestamp (see Section 5) is also included in the comparison. Specifically, for the dual-core, quad-core and eight-core configurations, we generate 50 random workload combinations from the pool of 19 SPEC CPU benchmarks in Table 2. Fig. 13 illustrates the five schemes' average throughputs on the 50 randomly selected dual-core, quad-core and eight-core workloads respectively, where CLU is shown to outperform all other schemes with a throughput improvement of dual cores by 6.3%, quad cores by 14.1% and eight cores by 8.9% over the baseline (true-)LRU scheme.

We find that the normalized performance of TADIP under random workloads in our study (6.1% / 2.1% for quad/eight-core systems respectively) is different from those reported in [3] (18% and 15% respectively). We speculate that several factors may have contributed to this discrepancy. For instance, [3] uses a CMP simulator based on the X86 ISA that has much more sophisticated memory addressing modes than a RISC ISA (e.g., the Alpha ISA in the M5 simulator), such as the register/immediate/direct/indirect/indexed addressing modes, which will make the memory accesses more intensive in X86. In addition, our benchmark pool differs from the one adopted in [3], which can also contribute to the discrepancy. We notice, however, that the mutual/relative performance trend among TADIP, NUCACHE, PIPP and UCP remains consistent with the conclusions in their respective studies [3], [8]–[10].

Another interesting observation is that the performance gap between CLU and PIPP shrinks under the random workloads, compared to the gap under the specially-constructed ones. The underlying reason might be that, as analyzed in Section 2.1, the capacity partitioning scheme PIPP has an *ad hoc* ability for locality-oriented improvement via such mechanisms as the stream handler or the single-step promotion [10]. We speculate this ability helps PIPP perform better in the locality dimension under the random workloads than under the specially-constructed workloads. However, CLU's ability to co-optimize locality and utility is systematic and enables it to consistently outperform PIPP in both kinds of workloads.

Furthermore, TA-DRRIP underperforms TADIP by 1.4% and 1.1% for the dual-core and quad-core random workloads

TABLE 4
Hardware Overhead Details

address length	44-bit physical address
cache line size	64 bytes
associativity	16 ways for 2MB dual-core and 4MB quad-core, 32 ways for 8MB eight-core
#(sample sets)/core	32
tag bits	27 bits for 2-core, 26 bits for 4/8-core
hashed monitor tag	10 bits for 2/4/8-core configurations
valid bit	1 bit
stack position bits	4 bits for 2/4-core, 5 bits for 8-core
hit counter	16 bits each

respectively. This is mainly because the pseudo-LRU based TA-DRRIP uses only 2 bits for each block, while the true-LRU based TADIP uses $\lceil \log_2 Associativity \rceil$ bits (>2). Besides, it was experimentally shown in [5] that RRIP's design choice of using 2 bits per block achieves almost the same performance as using a higher number of bits (e.g., $\lceil \log_2 Associativity \rceil$). That is why the 2-bit TA-DRRIP is chosen for evaluation and comparison in our experiments. We also notice that TA-DRRIP outperforms TADIP by 3.2% under the eight-core workloads. Our speculation is that, in this case, TA-DRRIP's capability of being both thrashing-resistant and scan-resistant (see Section 5) enables it to outperform TADIP that is only thrashing-resistant. Nevertheless, TA-DRRIP underperforms CLU under all of the dual-core, quad-core and eight-core random workloads, although CLU is not equipped with a scan-resistant module. We speculate that it is CLU's capability of locality and utility co-optimization that enables it to consistently outperform TA-DRRIP's locality-oriented management of being thrashing-resistant and scan-resistant.

4.4 Overhead Estimation

Since CLU requires an LRU profiler and a BIP profiler for per-core locality & utility monitoring, the shadow sets and hit counters will dominate hardware overhead in its design. Specifically, each shadow set entry consists of a tag field, a valid bit and stack position bits, as listed in Table 4. We also found in experiments that 16 bits are sufficient for a hit counter. Therefore, we can estimate that the storage overhead of CLU (with the practical BIP profiler) in dual-core, quad-core and eight-core systems are 5.79KB, 5.61KB and 11.46KB for the locality & utility monitor of each core, amounting to 0.56%, 0.55% and 1.12% of the overall SLLC capacity respectively. Moreover, we found that if we apply the 10-bit hash function of reference [16] to the tag field in the locality & utility monitoring logic, we can further reduce the overhead to 0.26%, 0.27% and 0.53% for dual-core, quad-core and eight-core configurations with negligible performance change compared to using full tag bits.

5 Related Work

As our paper focuses on the locality and the utility co-optimization of SLLCs, in what follows, we briefly review the representative work related to the shared organization and the co-optimization.

5.1 SLLC Capacity Management

Locality-Oriented Alternative Replacement Policies: As LRU is ineffective in handling workloads with poor locality, alternative replacement policies have been proposed to adapt locality-based capacity management decisions to workloads' specific locality characteristics, by means of sophisticated block insertion, promotion or victimization. Besides TADIP [3] and NUCACHE [8] introduced in Section 2.1, in [5], the *re-reference interval prediction* (RRIP) model generalizes three different categories of re-reference intervals for blocks—near, long and distant re-reference intervals. RRIP always predicts a long re-reference interval for incoming blocks in an attempt to prevent both thrashing and, more critically, scans that are referred to as the cache pollution due to a subset of incoming blocks being dead-on-fill. BRRIP (*bimodal RRIP*) predicts a distant re-reference interval with a high probability so as to prevent thrashing. TA-DRRIP is a thread-aware and feedback-based extension of RRIP to CMPs by dynamically learning whether RRIP or BRRIP is better with a dueling mechanism. But unlike the aforementioned schemes that adopt the true LRU policy in their underlying modules, TA-DRRIP puts more focuses on the practical implementation in a pseudo-LRU based environment by using only 2 bits per block for timestamp. Based on set sampling and a skewed dead-block prediction table, SDBP [7] bypasses dead-on-fill blocks.

Utility-Oriented Capacity Partitioning Schemes: The commonly used LRU policy implicitly divides the SLLC capacity among competing threads on a miss-driven basis, which can be ineffective in that threads may bring into the cache a lot of missed blocks without referencing them again. SLLC capacity partitioning is targeted at allocating LLC resources to threads on a utility or QoS basis. UCP [9] and PIPP [10] are both utility-oriented capacity partitioning schemes aiming to maximize the throughput performance. Among the QoS-directed schemes, Kim et al. [17] propose a fairness-based cache partitioning so that all threads are slowed down equally from where each thread monopolizes the SLLC. Nesbits et al. [18] introduce the notion of *virtual private caches* to satisfy certain QoS requirements.

6 Conclusion & Future Work

In this paper, we demonstrate that the existing alternative replacement policies or cache partitioning schemes cannot adapt to a wide spectrum of workloads with diverse locality and utility since they are oriented towards either of the two optimization goals only. Therefore, we propose CLU, a novel SLLC capacity management scheme that is capable of interactive locality and utility co-optimization. By employing light-weight monitors that profile both LRU and BIP hit curves, CLU can exploit the co-optimized locality and utility of concurrent threads and effectively manage the SLLC capacity for CMP workloads. Our execution-driven simulation shows that CLU can improve the throughput by 24.3%, 45.3% and 43.3% for our simulated dual-core, quad-core and eight-core systems (with 0.26%, 0.27% and 0.53% storage overhead) respectively, outperforming the existing replacement policies and partitioning schemes.

In this work, the design and implementation of CLU have been discussed based on the implicit assumption of more SLLC ways than CMP cores in capacity partitioning. With the

technology scaling, however, the core count tends to exceed the number of SLLC ways [19]. In our future exploration of CLUs design space, we plan to leverage *cuckoo hashing* [20] to decouple the logical cache associativity from physical cache ways (e.g., generating a 1024-associative cache with just 64 physical ways) and partition the high logical associativity rather than limited physical ways among many cores. Additionally, our current CLU design is rooted in the true-LRU environment, which simplifies our conceptualizing and presenting the locality and utility co-optimization idea that is the current major focus. Nevertheless, to make CLU more practical for CMP manufacturing, we are also investigating how to adapt CLU to a pseudo-LRU environment with much fewer timestamp bits (e.g., 1 bit in NRU or 2 bits in TA-DRRIP [5]) per block in our on-going work.

References

[1] Available: http://www.tilera.com/abouttilera/press-releases/tilera-announces-worlds-first-100-core-processor, accessed on 2014.

[2] I. Goiri et al., "Multifaceted resource management for dealing with heterogeneous workloads in virtualized data centers," in *Proc 2010 11th IEEE/ACM Int. Conf. GRID Comput. (GRID)*, 2010, pp. 25–32.

[3] A. Jaleel et al., "Adaptive insertion policies for managing shared caches," in *Proc. 17th Int. Conf. Parallel Archit. Compil. Tech. (PACT)*, 2008, pp. 208–219.

[4] W. Liu et al., "Using aggressor thread information to improve shared cache management for CMPs," in *Proc. 18th Int. Conf. Parallel Archit. Compil. Tech. (PACT)*, 2009, pp. 372–383.

[5] A. Jaleel et al., "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proc. 37th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2010, pp. 60–71.

[6] S. Khan et al., "Using dead blocks as a virtual victim cache," in *Proc. 19th Int. Conf. Parallel Archit. Compil. Tech. (PACT)*, 2010, pp. 489–500

[7] S. Khan et al., "Sampling dead block prediction for last-level caches," in *Proc. 2010 43rd Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2010, pp. 175–186.

[8] R. Manikantan et al., "NUcache: An efficient multicore cache organization based on next-use distance," in *Proc. 2011 IEEE 17th Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2011, pp. 243–253.

[9] M. Qureshi et al., "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2006, pp. 423–432.

[10] Y. Xie et al., "PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proc. 36th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2009, pp. 174–183.

[11] M. Qureshi et al., "Adaptive insertion policies for high performance caching," in *Proc. 34th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2007, pp. 381–391.

[12] G. E. Suh et al., "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *Proc. 8th Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2002, p. 117.

[13] R. L. Mattson et al., "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, 1970.

[14] N. L. Binkert et al., "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, Jul./Aug. 2006.

[15] Available: http://www.jaleels.org/ajaleel/htmls/RRIP.tgz, accessed on 2014.

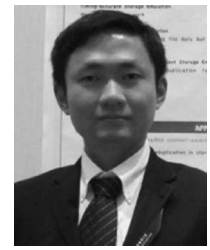
[16] M. V. Ramakrishna et al., "Efficient hardware hashing functions for high performance computers," *IEEE Trans. Comput.*, vol. 46, no. 12, pp. 1378–1381, Dec. 1997.

[17] S. Kim et al., "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proc. 13th Int. Conf. Parallel Archit. Compil. Tech. (PACT)*, 2004, pp. 111–122.

[18] K. J. Nesbit et al., "Virtual private caches," in *Proc. 34th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2007, pp. 57–68.

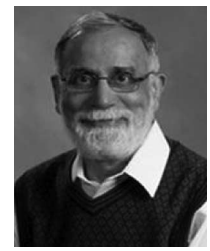
[19] D. Sanchez et al., "Vantage: Scalable and efficient fine-grain cache partitioning," in *Proc. 38th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2011, pp. 57–68.

[20] D. Sanchez et al., "The ZCache: Decoupling ways and associativity," in *Proc. 2010 43rd Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2010, pp. 187–198.



Dongyuan Zhan received the Bachelor's degree in computer engineering from Wuhan University, China, in 2006, and the PhD degree in computer engineering from the University of Nebraska-Lincoln, Lincoln, Nebraska, in 2012, with the dissertation on improving cache performance for chip multiprocessors at the architecture level. He is now an engineer at the Advanced Micro Devices, Inc.

Hong Jiang received the BSc degree in computer engineering from Huazhong University of Science and Technology, Wuhan, China, in 1982; the MASc degree in computer engineering from the University of Toronto, Canada, in 1987; and the PhD degree in computer science from the Texas A&M University, College Station, Texas, in 1991. Since August 1991, he has been with the University of Nebraska–Lincoln (UNL), where he served as a vice chair, from 2001 to 2007, and a professor with the Department of Computer Science and Engineering. At UNL, he has graduated more than 10 PhD students who upon their graduations either landed academic tenure-track positions in PhD-granting US institutions or were employed by major US IT corporations. His current research interests include computer architecture, computer storage systems and parallel I/O, high-performance computing, cloud computing, and performance evaluation. He serves as an associate editor of the *IEEE Transactions on Parallel and Distributed Systems*. He has over 180 publications in major journals and international conferences in these areas, including IEEE-TPDS, IEEE-TC, JPDC, ISCA, MICRO, USENIX ATC, FAST, LISA, ICDCS, IPDPS, OOPLAS, ECOOP, SC, ICS, HPDC, ICPP, etc., and his research has been supported by NSF, DOD, and the State of Nebraska. Dr. Jiang is a Senior Member of IEEE and Member of ACM.



Sharad C. Seth after receiving his PhD degree in electrical engineering from the University of Illinois, Urbana-Champaign, in 1970, was with the faculty of the Computer Science and Engineering Department, University of Nebraska–Lincoln, Nebraska until 2012, and is now an emeritus professor. Seth's long-term research has spanned the areas of VLSI testing and document analysis. In recent years, he has been working with colleagues at UNL on issues related to cache management for chip multiprocessors. From 1996 to 2003, he served as the Director of the UNLs Center for Communication and Information Science. Seth is a Member of ACM.