A Survey of Techniques for Cache Partitioning in Multicore Processors

SPARSH MITTAL, Oak Ridge National Laboratory

As the number of on-chip cores and memory demands of applications increase, judicious management of cache resources has become not merely attractive but imperative. Cache partitioning, that is, dividing cache space between applications based on their memory demands, is a promising approach to provide capacity benefits of shared cache with performance isolation of private caches. However, naively partitioning the cache may lead to performance loss, unfairness, and lack of quality-of-service guarantees. It is clear that intelligent techniques are required for realizing the full potential of cache partitioning. In this article, we present a survey of techniques for partitioning shared caches in multicore processors. We categorize the techniques based on important characteristics and provide a bird's eye view of the field of cache partitioning.

Categories and Subject Descriptors: A.1 [**General Literature**]: Introductory and Survey; C.0 [**Computer Systems Organization**]: System architectures

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Review, classification, multicore processor, shared cache, partitioning, fairness, QoS

ACM Reference Format:
Sparsh Mittal. 2017. A survey of techniques for cache partitioning in multicore processors. ACM Comput. Surv. 50, 2, Article 27 (May 2017), 39 pages.
DOI: http://dx.doi.org/10.1145/3062394

1. INTRODUCTION

Recent trends in processor design have made the effective management of cache resources more important than ever before. As the core count and memory demands of applications rise, prudent cache architectures and management policies have become vital to meet the challenges of plateauing area and power budgets. Researchers have recently explored private and shared cache designs that offer distinct advantages and disadvantages. While private caches avoid interference, they cannot account for inter- and intra-application variation in cache requirements and by virtue of limited capacity, they cannot reduce miss rate effectively. By comparison, shared caches can provide higher total capacity for reducing the miss rate; however, with traditional management policies, they may show performance loss, unfairness, and lack of quality of service (QoS) due to interference between applications [Yun and Valsan 2015; Herdrich et al. 2016].

The author worked on this article both while working at ORNL and at IIT Hyderabad.
Author's address: S. Mittal, E-621, Department of Computer Science and Engineering, Indian Institute of Technology (IIT) Hyderabad, Sangareddy, Telangana 502285, India; email: sparsh0mittal@gmail.com.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2017 ACM 0360-0300/2017/05-ART27 $15.00
DOI: http://dx.doi.org/10.1145/3062394

Cache partitioning[1] holds the promise to solve this dilemma by providing performance isolation of private caches and capacity advantage of shared caches. It is well-known that not only different applications but even different threads of a multithreaded application may show disparate cache demand and performance sensitivity [Muralidhara et al. 2010]. In addition to the running-applications, performance differences between cores may also arise due to the processor design itself, such as differences in cache latencies due to NUCA design and differences in core frequencies due to PV [Kozhikkottu et al. 2014]. CP can effectively compensate for these performance differences between cores. In addition to throughput, CP can also optimize for fairness and QoS objectives, which are especially important in shared computing platforms such as consolidated servers and cloud computing. By avoiding interference, CP can provide higher effective cache capacity to each application and, hence, performance of a higher-sized (e.g. 2X [Jaleel et al. 2008] or 1.5X [Nikas et al. 2008]) cache. CP can reduce off-chip misses and bandwidth contention, which may benefit even those applications whose cache quota are reduced [Jin et al. 2009; Pan and Pai 2013]. Further, by reducing execution time and allowing unused cache to be power-gated, CP can improve energy efficiency. It is clear that CP is a versatile and powerful management approach for a broad range of usage scenarios.

Although promising, CP is not a panacea. With increasing core count, the number of possible partitions increase exponentially, which renders simple schemes (e.g., brute-force search) ineffective. In fact, the problem of finding the partitioning with minimum overall cache miss rate (i.e., optimal partitioning) is NP-hard [Yu and Petrov 2010; Stone et al. 1992] and yet, optimal partitioning may not be fair [Brock et al. 2015]. Naive CPTs may incur large profiling and reconfiguration overhead and the hardware support required for implementing CPTs (e.g., true-LRU) may be too costly or unavailable on most processors. It is clear that intelligent design approach and parameter choices are essential for harnessing the full potential of CP. Several recently proposed CPTs seek to address these challenges.

Contributions: In this article, we present a survey of techniques for partitioning shared caches in multicore processors. Figure 1 presents the overview of the article. We first present a background on CP and discuss the tradeoffs involved in it (Section 2). We also discuss use of CPTs in real processors. We then classify the research works from several perspectives to provide insights (Section 3). Then, we review CPTs in terms of their granularities (Section 4), the strategies used for partitioning (Section 5), and the objective they seek to optimize (Section 6). Further, we summarize works that use CPTs in various contexts (Section 7) and that integrate CP with other management approaches (Section 8). We conclude with an outlook on future challenges (Section 9).

Scope: For making a comprehensive yet focused presentation, we limit the scope of this article as follows. While a cache can be partitioned between blocks of different characteristics (e.g., read/write intensive [Khan et al. 2014], frequent/infrequent writeback, etc.), in this article, we only include techniques that partition the shared cache between different applications or threads in a multicore processor. We review CP in multicore CPUs and not in CPU-GPU systems [Lee and Kim 2012; Mekkat et al.

[1]Following acronyms are used frequently in this article: "auxiliary tag directory" (ATD), bandwidth (BW), "bimodal insertion policy" (BIP), "cache allocation ticks" (CAT), cache partitioning (CP) technique (CPT), "cycle per instruction" (CPI), "dynamic insertion policy" (DIP), "dynamic voltage/frequency scaling" (DVFS), fair speedup (FS), insertion point (IP), "instruction per cycle" (IPC), "last level cache" (LLC), "least recently used" (LRU), least significant bit (LSB), "memory level parallelism" (MLP), "miss status holding register" (MSHR), "most recently used" (MRU), multithreaded (MT), "non uniform cache architecture" (NUCA), "phase change memory" (PCM), process variation (PV), "proportional-integral-derivative" (PID), protecting distance (PD), replacement policy (RP), reuse distance (RD), scaling factor (SF), virtual memory (VM) monitor (VMM), weighted speedup (WS). Also W_C refers to cache associativity.

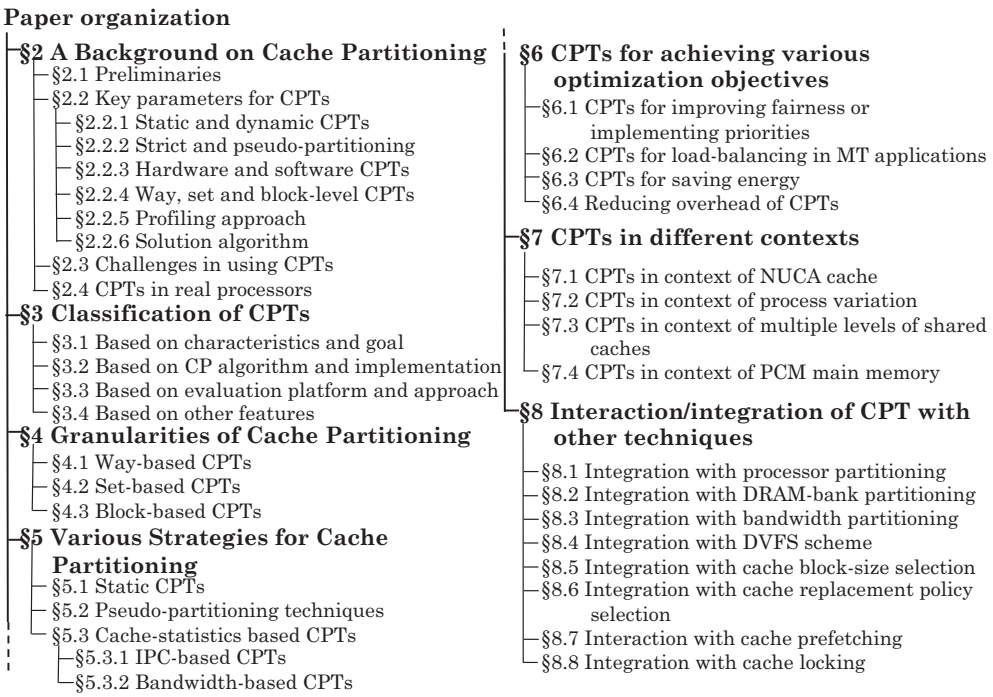

Fig. 1. Overall organization of the article.

2013]. Since different projects use different evaluation approaches, we focus on their qualitative insights and only include selected numeric results. This article is expected to be useful for both researchers and practitioners in the field of cache design and management.

2. A BACKGROUND ON CACHE PARTITIONING

2.1. Preliminaries

We now discuss some concepts/terms that will be useful throughout the article.

Quota allocation and enforcement policies: A CPT is composed of a "quota allocation" policy for determining the quota of each application and a "quota enforcement" policy, which actually imposes these quotas. For example, an allocation policy may determine quotas of two applications as 75% and 25% of the cache, respectively, based on their cache sensitivity. Now, for a cache with 64 colors and 16 ways, a coloring-based policy may enforce these quotas by giving 48 and 16 colors, respectively, whereas a way-based policy may enforce them by giving 12 and 4 ways, respectively, to those applications.

Victim selection, insertion, and promotion policies: An RP can be decomposed into the following three policies. When a new block needs to be inserted in cache, the "victim selection policy" decides the block to be evicted. The "insertion policy" decides the priority position at which the new block is inserted. For a block seeing cache hit, the "promotion policy" decides how its position is updated in the priority chain. For example, for LRU RP, the least-recent block is evicted, new block is inserted at most-recent position, and a block seeing hit is promoted to most-recent position.

Stack property: With RPs obeying stack property [Mattson et al. 1970], an access that hits in a W_C-way cache will also hit if the cache had more than W_C ways (for the same set-count).

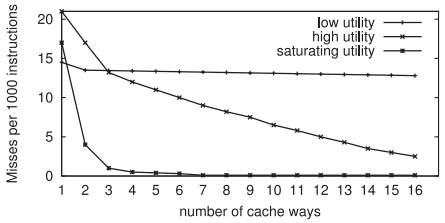

Fig. 2. Illustration of different utility curves (variation in misses with cache quota).

Set-sampling scheme: Several key characteristics (e.g., miss rate) of a set-associative cache can be estimated by sampling only a few of its sets and this refers to set-sampling. It allows lowering the profiling overhead.

Reuse distance: RD refers to the number of distinct accesses between two accesses to a cache block, for example, in access stream `M F G K E J H N E K J F G H`, RD of `G` is 6 [Mittal 2016b]. The distinct accesses can be counted for the whole cache or for each set [Pan and Pai 2013; Duong et al. 2012], which are termed as global or set-specific RDs, respectively.

Cache-behavior-based application classification: Based on their cache behavior, applications can be classified in multiple categories. Applications that do not or do benefit from cache are termed as cache "insensitive" and "friendly", respectively. "Streaming applications" have very large working set and they show thrashing with any RP due to inadequate cache reuse. The working set of "thrashing applications" is larger than the cache capacity and due to this, they thrash an LRU-managed cache, although they may benefit from cache on using thrash-resistant RPs. Similarly, based on rate of reduction in miss rate or execution time achieved with increasing cache quota, applications can also be characterized as having low, high or saturating utility, as illustrated in Figure 2.

2.2. Key Parameters for CPTs

CPTs can be characterized based on key parameters. We now discuss some of them.

2.2.1. Static and Dynamic CPTs. Static and dynamic CPTs make quota allocation decisions offline and at runtime, respectively (refer to Table I).

2.2.2. Strict and Pseudo-partitioning. Depending on whether the cache quota determined for an application is strictly enforced or not, CPTs can be classified as strict (hard) and pseudo (soft) partitioning, respectively (refer to Table I).

2.2.3. Hardware and Software CPTs. The CPTs can be implemented in HW or SW (refer to Table I).

2.2.4. Way, Set, and Block-level CPTs. Based on the granularity at which cache is allocated, CPTs may be classified as way, set (or color), and block-level (refer to Table II). In physical address, the overlapping bits between set-index and physical page number are termed as page color. Set-based CPTs work by controlling these bits to change the number of colors (and, hence, sets) allocated to a core [Cui et al. 2014; Lin et al. 2009].

Way, set and block-based CPTs provide increasingly fine-grained allocation, for example, a 16-way, 4MB cache with block size of 64B and system page size of 4KB has 16 ways, 64 colors and 65536 blocks. Also note that any CPT needs to provide a cache mapping for each cache block to avoid bypassing it since bypassing complicates cache coherence and requires special management techniques [Mittal 2016b]. Coarse-grain allocation may not enforce the exact partition size whereas fine-grain allocation may only be beneficial for large number of cores.

Table I. Properties/Challenges of Static/Dynamic, Strict/Pseudo, and HW/SW-based CPTs

Static	(+) For small core count, static CPTs may be useful for evaluating all possible partitions and finding the upper bound on gain from CP. (−) With increasing number of cores, the possible combinations of applications increase exponentially, which makes use of static CPTs infeasible. Static CPTs cannot adapt for temporal variation in cache behavior.
Dynamic	(+) Only dynamic CPTs are viable for large core counts. (−) Dynamic CPTs incur runtime overheads and they may be unnecessary if the application behavior is uniform over time.
Strict	(+) Strict partitioning is important to *guarantee* QoS and fairness. (−) It may lead to inefficient utilization of cache, especially when the allocation granularity is large (e.g., way-based CPTs). For example, if a core does not send accesses to a set, ways allocated to the core in this set remain unused. Also, dead blocks of one core cannot be evicted by other cores, even if these cores can derive additional hits by doing so.
Pseudo	(+) Pseudo partitioning may have simpler implementation and may allow core(s) to steal quotas of other core(s) for improving performance. (−) However, due to this, the instantaneous quota of an application can differ significantly from the target (e.g., by two out of eight ways [Halwe et al. 2013]). This problem becomes especially severe with increasing number of cores [Sanchez and Kozyrakis 2011].
HW-based	(+) HW management is important to reduce profiling and reconfiguration overhead. HW CPTs can be used at fine-granularity, for example, hundreds of thousands of cycles. (−) Adding the HW support required for their functioning may present challenges.
SW-based	(+) SW control is important to account for other processor components, management schemes, and system-level goals such as optimizing fairness (versus cache-level goals such as minimizing miss rate). (−) SW CPTs (e.g., those involving page-coloring) generally incur higher overhead and, therefore, can only be invoked at coarse granularity.

Table II. Properties and Challenges of Way/Set/Block-based CPTs

CPTs	Properties/Challenges
Way-based	(+) Way-based CP provides relatively simple implementation, flushing-free reconfiguration and ease of obtaining way-level profiling information. Due to these and the fact that most existing works have targeted small number of cores, way-based CP has received much more attention that set-based or block-based CP (refer to Table IV). (−) Way-based CPT is meaningful only if $W_C \geq 2N$ (N = number of cores) since at least one way needs to be allocated to each core. Hence, it requires caches of high associativity, which incur high access latency/power overheads [Mittal et al. 2014b]. It harms associativity [Wang and Chen 2014]. Also, it may require changes to RP and additional bits to identify owner core of each block
Set-based	(+) It provides higher granularity than way-based CP and is amenable to SW control. (−) It requires significant changes to OS functions and may complicate virtual memory management. Since reconfiguration in set-based CPTs changes the set-indices of many blocks, these blocks need to be flushed or migrated to new set-indices. To reduce this overhead, reconfiguration frequency can be lowered [Lin et al. 2009; Zhang et al. 2009], lazy reconfiguration [Lin et al. 2009] or address remapping strategies [Mittal et al. 2014a; Mittal and Zhang 2013; Lin et al. 2009] can be used and the number of recolored pages in each interval can be controlled [Zhang et al. 2009].
Block-based	(+) It provides highest granularity and it will become increasingly important with rising number of cores. (−) Obtaining profiling information for block-based (fine-grain) allocation is challenging and so, some block-based CPTs obtain this information by linearly interpolating the miss rate curve of way-level monitors [Sanchez and Kozyrakis 2011], which may not be fully accurate. Also, it may require changes to RP and additional bits to identify owner core of each block.

Table III. Properties/Challenges of Different Profiling and Solution Approaches

Collecting profiling data from ...	
Actual cache	(+) This approach does not incur storage and complexity overheads of additional units and, as such, it is especially suited for studies performed on real-systems. (−) It may require evaluating the performance/miss rate of each possible configuration one by one [Lin et al. 2008; Cook et al. 2013], which incurs cache reconfiguration and time overhead. Or it may require dedicating few cache sets to experiment with different policies [Jaleel et al. 2008] and, due to this, they continue to use sub-optimal policy.
Separate profiling unit	(+) Using separate per-core monitoring units can provide more accurate estimate about solo behavior of each program [Xie and Loh 2009; Qureshi and Patt 2006], and they allow experimenting with completely different policies or cache configurations without disturbing the main cache [Mittal et al. 2014a; Mittal and Zhang 2013]. (−) Separate units cannot model the impact of co-running programs [Jaleel et al. 2008], and their overhead rises rapidly with increasing number of cores.

Solution approach	
Heuristic-approaches	(+) They are generally simpler. (−) They require manual parameter-tuning and may not guarantee optimizing the objective.
Mathematical/ analytical models	(+) They are based on rigorous theoretical foundation and, as such, can provide strong guarantees and better scalability with rising number of cores. (−) They may require learning the parameters by offline simulation with training dataset or representative workloads and adjusting the parameters continuously to keep the model up to date. Also, their hardware implementation may incur large overhead.

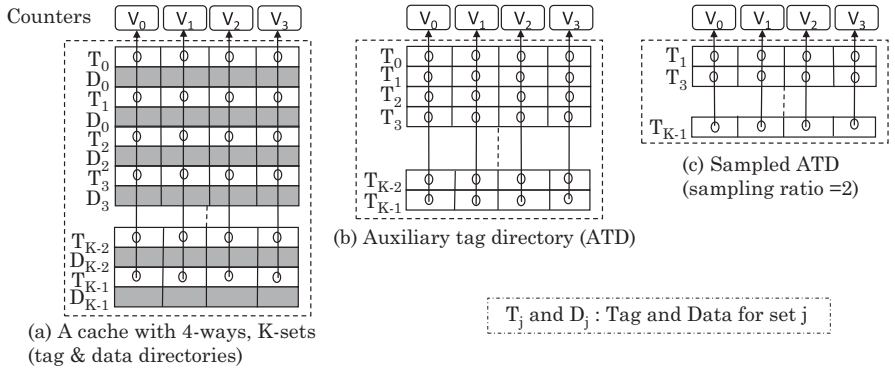

Fig. 3. Collecting profiling data for a core from (a) actual cache and (b, c) separate profiling unit (ATD). (Assuming that in each set, ways are ordered from MRU to LRU. Counter V_0 and V_3 are updated from MRU and LRU blocks of all the sets respectively, and so on.)

2.2.5. Profiling Approach. Profiling data for guiding CP algorithm can be collected from the actual cache itself or a separate monitoring unit (refer to Table III and Figure 3).

2.2.6. Solution Algorithm. While most CPTs work based on heuristics developed using insights into the problem, some CPTs use well-known solution algorithm or mathematical models (refer to Table III). Examples of these algorithms/approaches include dynamic programming, gradient-descent algorithm, feedback control theory, and so on (refer to Table V).

2.3. Challenges in Using CPTs

We now summarize the tradeoffs and roadblocks in using CPTs.

Unrealistic assumptions: Most way-based CPTs assume true-LRU replacement policy, although the commercial processors use only approximations of LRU [Kedzierski et al. 2010]. This is due to the large storage overhead of true-LRU ($W_C \log_2(W_C)$ bits for each set) and small difference between the performance of true-LRU and pseudo-LRU for large associativity caches. However, these pseudo-LRU RPs and other recent RPs (e.g., BIP [Qureshi et al. 2007]) do not follow stack property and, hence, some way-based CPTs may not work with these RPs. Also, the modeling frameworks for some CPTs are developed assuming fully associative caches [Petoumenos et al. 2006] or skew-associative caches [Sanchez and Kozyrakis 2011] and, consequently, these models may not be fully accurate when applied to set-associative caches.

Challenges in improving performance: Many CPTs treat different misses identically, though the latency of different misses can be different due to instantaneous MLP and NUCA effects (Section 5.3.1). Also, other factors, such as bandwidth congestion or load-imbalance, may create performance bottleneck and negate the advantage of CP. Due to these, a reduction in miss rate brought by a CPT may not translate into corresponding performance improvement. To avoid this issue, some CPTs account for latency impact of each miss; however, these models may not be straightforward, fully accurate, and portable.

Limited scope: CP is useful only for LLC-sensitive applications, which cause destructive interference in shared cache. Thus, CP becomes unnecessary and even harmful for small-footprint [Cook et al. 2013] or locality-friendly [Zhan et al. 2014; Chang and Sohi 2007] applications and large-sized caches, for example, stacked DRAM cache [Mittal and Vetter 2016].

Need of careful design choices: CPTs need to optimize multiple and generally conflicting objectives, such as throughput, fairness, QoS, energy efficiency, load-balancing, and so on. This necessitates a careful choice of CPT design parameters, for example, quota allocation and enforcement policies, partitioning interval [Moreto et al. 2009], and so on.

2.4. CPTs in Real Processors

CPTs have been integrated into several product systems. For example, some Intel processors provide support for way-based CP [Intel Corporation 2016]. Similarly, software-based shared cache partitioning approach using page coloring [Lin et al. 2008] has been integrated in Linux operating system [OSU-CSE News 2010].

Intel Xeon processor E5-2600 v3 family provides support for implementing shared cache QoS [Herdrich et al. 2016]. It provides a "cache monitoring technology" to track shared cache usage of various programs and a "cache allocation technology" for allocating cache quotas based on OS/VMM policies, for example, reducing contention, isolating thrashing programs, and avoiding cache starvation. Cache allocation technology can work with any underlying quota enforcement scheme, for example, way/block/set-based.

AMD Opteron processor uses a 6MB 16-way shared L3 cache [Conway et al. 2010]. To avoid cache pollution, the L3 controller detects applications with high cache access intensity but low hit rate. The lines of these applications are not promoted to the MRU position, instead, they are set to either LRU position or midway through the LRU stack [Conway et al. 2010]. The L3 cache is also partitioned to store normal data (5MB) and cache directory (probe filter) for the memory controller (1MB). This design saves chip space compared to keeping a separate on-chip probe filter, since the probe filter is enabled only in multiprocessors and is disabled in single-processor designs. Also, compared to keeping the directory in DRAM, keeping it in SRAM allows fast read/write accesses and reduces memory latency and bandwidth. The quotas of L3 cache (5MB) and directory (1MB) are chosen based on the tradeoff between minimizing L3 cache miss and minimizing downgrades [Conway et al. 2010].

3. CLASSIFICATION OF CPTS

3.1. Based on Characteristics and Goal

Table IV classifies the works based on their attributes and optimization targets. Evidently, dynamic CPTs and strict CPTs have been used more often than their counterparts due to their ability to dynamically adapt to workload behavior and enforce strict quotas for implementing system-level policies, respectively. From Table IV, it is also clear that CP has been employed for a diverse range of optimizations.

3.2. Based on CP Algorithm and Implementation

Table V highlights the solution algorithm/approach used by different works as well as their approach for obtaining profiling information (refer to Table III for a background). Schemes such as set-sampling, partial tag, and Bloom filter-based monitoring unit reduce the profiling overhead at the cost of small inaccuracy. Works that use these schemes are also highlighted in Table V.

3.3. Based on Evaluation Platform and Approach

Table VI summarizes the evaluation platform used by each work. Simulators allow flexibility to experiment with different policies/parameters that may possibly be infeasible to implement on actual hardware. By comparison, real processors enable rapid and realistic evaluation and, hence, allow using large execution length [Lin et al. 2008], which is especially important for evaluating large-sized caches. Some works use analytical models [Stone et al. 1992; Brock et al. 2015; Petoumenos et al. 2006; Planas et al. 2007; Oh et al. 2011; Liu et al. 2010; Sanchez and Kozyrakis 2011; Hsu et al. 2006] to develop/guide their technique and simulator for actual evaluation. Analytical models are useful for evaluating limit gains (e.g., optimal CP for miss rate reduction) independent of a particular processor architecture or workload. However, these models may study cache in isolation and, therefore, may not model the effect of cache optimization on system performance and its interaction with other components. Clearly, all three platforms/approaches are indispensable and offer complementary insights.

To get a rough estimate of scalability of CPTs, Table VI also categorizes them based on the highest number of cores for which they are evaluated. Most works compare their technique with unmanaged shared cache and other related techniques. Some works also provide comparison with statically partitioned (private) cache and partitioned fully associative cache and they are highlighted in Table VI.

3.4. Based on Other Features

To get more insights, we now classify the works based on additional characteristics.

(1) Some works classify the applications/threads based on their cache behavior to guide the CPTs or create representative workloads [Zhou et al. 2016; Chang and Sohi 2007; Kaseridis et al. 2014; Moreto et al. 2008; Jaleel et al. 2008; Nikas et al. 2008; Jin et al. 2009; Moreto et al. 2009; Herrero et al. 2010; Liu et al. 2014; Cook et al. 2013; Lin et al. 2008, 2009; Ye et al. 2014; Sundararajan et al. 2012; Planas et al. 2007; Dybdahl and Stenstrom 2007; Liu et al. 2010]. Some works perform classification of misses [Reddy and Petrov 2010; Sundararajan et al. 2012; Nikas et al. 2008].
(2) When a cache quota has been specified for an interval, some CPTs may purposely increase the quota for a sub-interval and reduce it for another sub-interval such that specified quota is enforced *on average over the entire interval* [Pan and Pai 2013; Chang and Sohi 2007]. By comparison, most other CPTs enforce the *exact specified quota for the entire interval*.

Table IV. A Classification Based on Characteristics and Goals of CPTs

Category	References
Static or dynamic CPT	
Static CPTs	[Stone et al. 1992; Yu and Petrov 2010; Cui et al. 2014; Jin et al. 2009; Reddy and Petrov 2010; Brock et al. 2015; Tam et al. 2007; Oh et al. 2011; Yun and Valsan 2015]
Both static and dynamic CPTs	[Rafique et al. 2006; Iyer 2004; Cook et al. 2013; Lin et al. 2008; Ye et al. 2014; Kandemir et al. 2011b; Kim et al. 2004]
Dynamic CPTs	Nearly all others
Pseudo or strict partitioning	
Pseudo-partitioning	[Reddy and Petrov 2010; Jaleel et al. 2008; Hasenplaugh et al. 2012; Xie and Loh 2009; Halwe et al. 2013; Kaseridis et al. 2014; Petoumenos et al. 2006; Duong et al. 2012]
Both pseudo and strict partitioning	[Rafique et al. 2006]
Strict-partitioning	Nearly all others
Allocation granularity	
Way-level	[Qureshi and Patt 2006; Zhou et al. 2016; Zhan et al. 2014; Sundararajan et al. 2012; Kedzierski et al. 2010; Moreto et al. 2009; Suh et al. 2004; Subramanian et al. 2015; Wang and Martínez 2015; Srikantaiah et al. 2009b; Kozhikkottu et al. 2014; Muralidhara et al. 2010; Srikantaiah et al. 2009a; Zhou et al. 2012; Moreto et al. 2008; Xie and Loh 2009; Pan and Pai 2013; Kotera et al. 2011; Nikas et al. 2008; Halwe et al. 2013; Reddy and Petrov 2010; Yeh and Reinman 2005; Gupta and Zhou 2015; Cook et al. 2013; Kandemir et al. 2011b; Kaseridis et al. 2014; Chang and Sohi 2007; Lin and Balasubramonian 2011; Kim et al. 2004; Lee et al. 2011; Chiou et al. 2000; Planas et al. 2007; Xie and Loh 2010; Bitirgen et al. 2008; Lo et al. 2016; Dybdahl and Stenstrom 2007; Rafique et al. 2006; Liu et al. 2010; Iyer 2004; Kandemir et al. 2011a; Intel Corporation 2016; Beckmann and Sanchez 2016]
Set/color-level	[Mittal et al. 2014a; Lin et al. 2008; Mittal and Zhang 2013; Awasthi et al. 2009; Zhang et al. 2009; Cui et al. 2014; Yu and Petrov 2010; Jin et al. 2009; Reddy and Petrov 2010; Liu et al. 2014; Suhendra and Mitra 2008; Ye et al. 2014; Tam et al. 2007; Lin et al. 2009; Yun and Valsan 2015]
Block-level	[Sanchez and Kozyrakis 2011; Iyer et al. 2007; Wang and Chen 2014; Hasenplaugh et al. 2012; Kasture and Sanchez 2014; Petoumenos et al. 2006; Duong et al. 2012]
Optimization target	
Performance	Nearly all
Fairness	[Kim et al. 2004; Zhan et al. 2014; Lin et al. 2009; Qureshi and Patt 2006; Lin et al. 2008; Chang and Sohi 2007; Kedzierski et al. 2010; Subramanian et al. 2015; Duong et al. 2012; Wang and Martínez 2015; Srikantaiah et al. 2009a, 2009b; Hasenplaugh et al. 2012; Moreto et al. 2008; Xie and Loh 2009; Pan and Pai 2013; Jaleel et al. 2008; Yeh and Reinman 2005; Moreto et al. 2009; Liu et al. 2014; Brock et al. 2015; Kaseridis et al. 2014; Zhang et al. 2009; Xie and Loh 2010; Oh et al. 2011; Bitirgen et al. 2008; Hsu et al. 2006; Rafique et al. 2006; Zhou et al. 2016]
QoS or priority	[Kasture and Sanchez 2014; Moreto et al. 2009; Mittal and Zhang 2013; Iyer et al. 2007; Srikantaiah et al. 2009a; Wang and Chen 2014; Hasenplaugh et al. 2012; Cook et al. 2013; Ye et al. 2014; Kasture and Sanchez 2014; Chang and Sohi 2007; Petoumenos et al. 2006; Lee et al. 2011; Lo et al. 2016; Rafique et al. 2006; Iyer 2004; Kandemir et al. 2011a; Herdrich et al. 2016]
Static energy	By power-gating unused cache [Sundararajan et al. 2012; Mittal and Zhang 2013; Mittal et al. 2014a; Kotera et al. 2011; Reddy and Petrov 2010; Wang et al. 2012]
Dynamic energy	By activating smaller cache portion [Sundararajan et al. 2012; Reddy and Petrov 2010; Varadarajan et al. 2006], due to reduced misses/writebacks leading to shorter execution time and fewer main-memory accesses [Herrero et al. 2010; Cook et al. 2013; Kedzierski et al. 2010; Zhou et al. 2012], by enabling co-location of applications [Cook et al. 2013; Lo et al. 2016]
Power capping	[Wang et al. 2012]

Table V. A Classification Based on Algorithm and Implementation of CPTs

Solution approach/heuristic	
Regression and curve fitting	[Srikantaiah et al. 2009a; Kandemir et al. 2011b; Muralidhara et al. 2010; Zhan et al. 2014; Wang et al. 2012; Kandemir et al. 2011a]
Machine learning	[Bitirgen et al. 2008; Liu et al. 2014]
Dynamic programming	[Kozhikkottu et al. 2014; Suhendra and Mitra 2008; Brock et al. 2015]
Feedback control theory	[Srikantaiah et al. 2009b; Wang et al. 2012]
Gradient-descent algorithm	[Hasenplaugh et al. 2012; Lo et al. 2016]
Market-based approach	[Wang and Martínez 2015]
Graph coloring	[Jung et al. 2010]
Profiling information obtained from	
A separate profiling unit	[Qureshi and Patt 2006; Mittal et al. 2014a; Kedzierski et al. 2010; Mittal and Zhang 2013; Zhou et al. 2012; Moreto et al. 2008; Xie and Loh 2009; Reddy and Petrov 2010; Subramanian et al. 2015; Yeh and Reinman 2005; Moreto et al. 2009; Gupta and Zhou 2015; Wang and Martínez 2015; Kaseridis et al. 2014; Kasture and Sanchez 2014; Lin et al. 2009; Zhan et al. 2014; Lee et al. 2011; Xie and Loh 2010; Sanchez and Kozyrakis 2011]
Actual cache itself	Nearly all others
Reducing profiling overhead	
Set sampling	[Qureshi and Patt 2006; Mittal et al. 2014a; Zhan et al. 2014; Kedzierski et al. 2010; Mittal and Zhang 2013; Hasenplaugh et al. 2012; Zhou et al. 2012; Moreto et al. 2008; Xie and Loh 2009; Jaleel et al. 2008; Kaseridis et al. 2009; Subramanian et al. 2015; Moreto et al. 2009; Gupta and Zhou 2015; Wang and Martínez 2015; Kaseridis et al. 2014; Kasture and Sanchez 2014; Lin et al. 2009; Xie and Loh 2010; Jaleel et al. 2008; Sanchez and Kozyrakis 2011]
Partial tag	[Kaseridis et al. 2009, 2014; Nikas et al. 2008]
Bloom Filter	[Nikas et al. 2008]

(3) In some works, all shared caches (e.g., L2 and L3) are partitioned [Kandemir et al. 2011b, 2011a], whereas in other works, only a single last-level shared cache (which may be L2 or L3) is partitioned.
(4) In some CPTs, most partitions are private and the remaining cache is shared between cores [Herrero et al. 2010; Brock et al. 2015; Liu et al. 2014; Sanchez and Kozyrakis 2011; Cook et al. 2013; Lin et al. 2009; Xie and Loh 2010; Dybdahl and Stenstrom 2007; Settle et al. 2006]. While this approach precludes partitioning the whole cache, it may provide the benefit of both private and shared caches. By comparison, in most other CPTs, all the partitions are private to assigned cores.
(5) Based on stack property, a single W_C-way ATD can provide hit/miss information for all caches with ways 1 to W_C [Qureshi and Patt 2006]. Hence, CPTs utilizing ATD can change the number of ways of a core by *more than one way* in each repartitioning event. By comparison, some techniques collect profiling information for one more or one less way than the current number of ways [Nikas et al. 2008; Herrero et al. 2010] and, as such, each time these techniques can change the quota by *one way only*.
(6) Some techniques require the miss-ratio curve of individual curves to be convex [Stone et al. 1992], whereas other techniques do not have such requirement [Brock et al. 2015; Wang and Martínez 2015; Qureshi and Patt 2006].
(7) Some CPTs enforce partitioning decisions at the time of replacement (e.g., Wang and Chen [2014], Sanchez and Kozyrakis [2011], and Rafique et al. [2006]), whereas others enforce them through cache allocation (e.g., Lin et al. [2008] and Mittal et al. [2014a]).

Table VI. A Classification Based on Evaluation Platform and Approach Used by Different Works

Category	References
Evaluation platform	
Real-system	[Cook et al. 2013; Lin et al. 2008; Tam et al. 2007; Liu et al. 2014; Zhang et al. 2009; Jin et al. 2009; Ye et al. 2014; Lo et al. 2016; Yun and Valsan 2015; Herdrich et al. 2016]
Simulator	most others
Number of cores evaluated	
2 cores	[Settle et al. 2006; Mittal and Zhang 2013; Lin et al. 2008; Iyer et al. 2007; Kotera et al. 2011; Halwe et al. 2013; Jin et al. 2009; Suhendra and Mitra 2008; Cook et al. 2013; Zhang et al. 2009; Tam et al. 2007; Kim et al. 2004; Petoumenos et al. 2006; Suh et al. 2004; Planas et al. 2007; Lo et al. 2016; Oh et al. 2011; Iyer 2004]
$4 \leq$ cores ≤ 8	[Suh et al. 2001; Yun and Valsan 2015; Mittal et al. 2014a; Lin and Balasubramonian 2011; Qureshi and Patt 2006; Chang and Sohi 2007; Zhan et al. 2014; Kedzierski et al. 2010; Awasthi et al. 2009; Kaseridis et al. 2014, 2010; Nikas et al. 2008; Jung et al. 2010; Srikantaiah et al. 2009b; Kozhikkottu et al. 2014; Muralidhara et al. 2010; Hasenplaugh et al. 2012; Moreto et al. 2008; Pan and Pai 2013; Yu and Petrov 2010; Nikas et al. 2008; Cui et al. 2014; Kaseridis et al. 2009; Yeh and Reinman 2005; Moreto et al. 2009; Liu et al. 2014; Gupta and Zhou 2015; Brock et al. 2015; Ye et al. 2014; Kandemir et al. 2011b; Sundararajan et al. 2012; Kasture and Sanchez 2014; Lin et al. 2009; Wang et al. 2012; Xie and Loh 2010; Liu et al. 2010; Bitirgen et al. 2008; Hsu et al. 2006; Dybdahl and Stenstrom 2007; Rafique et al. 2006]
$16 \leq$ cores ≤ 32	[Sanchez and Kozyrakis 2011; Jaleel et al. 2008; Wang and Chen 2014; Subramanian et al. 2015; Duong et al. 2012; Srikantaiah et al. 2009a; Zhou et al. 2012; Herrero et al. 2010; Herdrich et al. 2016; Zhou et al. 2016]
$64 \leq$ cores ≤ 256	[Kaseridis et al. 2010; Wang and Martínez 2015; Lee et al. 2011]
Comparison with ...	
Statically partitioned cache	[Qureshi and Patt 2006; Brock et al. 2015; Wang and Martínez 2015; Lin et al. 2008; Muralidhara et al. 2010; Srikantaiah et al. 2009a; Mittal et al. 2014a; Hasenplaugh et al. 2012; Pan and Pai 2013; Kozhikkottu et al. 2014; Kotera et al. 2011; Kaseridis et al. 2009; Yeh and Reinman 2005; Herrero et al. 2010; Jung et al. 2010; Cook et al. 2013; Kandemir et al. 2011b; Sundararajan et al. 2012; Kasture and Sanchez 2014; Chang and Sohi 2007; Lin et al. 2009; Lee et al. 2011; Jaleel et al. 2008; Bitirgen et al. 2008; Hsu et al. 2006; Liu et al. 2010; Kandemir et al. 2011a]
Partitioned fully associative cache	[Wang and Chen 2014]

4. GRANULARITIES OF CACHE PARTITIONING

In this section, we discuss CPTs in terms of their granularity of cache allocation.

4.1. Way-Based CPTs

Suh et al. [2004] present a way-based CPT that estimates the cache utility of an application by using per-way hit counters. From utility values, marginal gain is computed and then, iteratively, one way is transferred from an application with least marginal gain to that with largest marginal gain. Finally, the new partition is compared with the previous one and the better partition is selected. To enforce cache quota, at the time of replacement, if the actual quota of an application is larger or smaller than its target quota, a block owned by same or other (respectively) application can be evicted.

Qureshi and Patt [2006] present a CPT that allocates cache quotas to application based on their cache utility (i.e., reduction in miss rate with increasing allocation) and not cache demand (i.e., the number of unique blocks referenced in an interval). To obtain utility values, they use ATD for each core (application) and based on set-sampling idea, only few sets are monitored. ATD stores only tags and not data and it records LRU position of each hit. Since LRU follows stack property, ATD allows estimating cache

hit/miss rate for cases where different number of ways were allocated to the application. From hit/miss rate information, utility of each cache way is obtained and then their CP algorithm periodically determines cache quotas that maximize the total utility, which is same as minimizing the total number of misses. However, with increasing number of cores (applications), the number of possible partitions increase exponentially. They propose a greedy CP algorithm that iteratively assigns a way to an application with highest utility for that way. This algorithm is optimal if the utility curves of all applications are convex [Stone et al. 1992]. For the general case of applications with non-convex curve, they propose a "lookahead algorithm." In every iteration, this algorithm computes the "maximum marginal utility" (MMU) and least number of ways at which MMU occurs for all the applications. The application with largest MMU value is allocated the number of ways it requires to achieve MMU. The iterative algorithm stops when all ways have been allocated. If all applications show convex utility curves, then lookahead algorithm reduces to the greedy algorithm discussed above. They show that their CPT improves performance and fairness compared to shared cache and equally partitioned cache.

Planas et al. [2007] present two metrics for measuring cache sensitivity of applications. The first metric ($w_{K\%}$) shows the number of cache ways required for an application to achieve $K\%$ (e.g., 90%) of its IPC with all the ways. As an example, in a 16-way LLC, applications with $w_{90\%}$ greater than or less than 8 can be classified as high or low utility, respectively. The second metric (w_{LRU}) estimates the number of ways that would have been given by LRU to an application when executed with another application. This metric is estimated by scaling the cache associativity in proportion to the cache accesses of each application when executed alone. Based on the insights provided by these metrics, the performance with LRU and a CPT seeking to minimize misses, termed MinMisses (e.g., Qureshi and Patt [2006]), can be predicted in a two-core system. For example, assume case 1 where the sum of $w_{90\%}$ of both applications is less than associativity. Now, if $w_{90\%} < w_{LRU}$ for both applications (case 1A), then LRU achieves similar performance as MinMisses. If $w_{90\%} < w_{LRU}$ for one application and $w_{90\%} > w_{LRU}$ for another application (case 1B), then LRU harms performance of second application. MinMisses can improve performance in such cases. In case 2, the sum of $w_{90\%}$ of both applications exceeds the associativity, and for such cases, LRU and MinMisses are suited for different applications. The limitation of their approach is that it works only with true-LRU and becomes infeasible for more than two cores.

Bitirgen et al. [2008] present a machine-learning approach to synergistically manage multiple shared resources (specifically L2 cache quota, power budget, and off-chip bandwidth) for enforcing higher-level performance objectives. Their technique periodically redistributes resources between applications to respond to dynamic changes in program behavior. They use an ensemble of "artificial neural networks" (ANNs) for learning an approximate model of program performance as function of the allocated resources and its recent behavior. Attributes, which characterize current L2 cache state and recent behavior, include read/write misses and hits in L1D cache over last 20K and 1.5M instructions separately and the fraction of cache ways assigned to a program that are dirty. These attributes model program behavior over both recent and distant past and also estimate the number of writebacks that may be generated by an application. By averaging the estimates from all ANNs, the final performance estimate is obtained. This approach allows assigning confidence levels to ANN estimates and also improves accuracy. They also discuss strategies for training the ANNs and improving their accuracy, for example, avoiding over-fitting and reducing estimation errors.

For optimizing a system-level performance metric, different possible combinations of resource allocations need to be searched. Since exhaustive search is infeasible, they use a stochastic hill-climbing-based search heuristic. The search algorithm focuses on

specific regions of the global allocation space where each region restricts resource allocation optimization to a subset of program and systematically allocates fair quotas to other programs. They select the samples for training the models by distributing resources uniformly randomly across the allocation space, which allows selecting representative samples. Also, with each addition of a new sample, an existing sample is discarded to keep samples from a long time duration and still give higher priority to recent samples. Finally, the resource allocation configuration expected to optimize the desired metric is chosen and implemented. They experiment with a way-based CPT, a per-core DVFS-based technique to manage chip-power distribution, and a strategy for distributing bandwidth between applications. They show that compared to isolated management of individual resources or uncoordinated management of multiple resources, their ANN-based synergistic management approach provides higher throughput and fairness.

Rafique et al. [2006] present a technique that allows the OS to specify cache quota based on system-level objectives and the quotas are enforced by the hardware. The OS specifies the quotas in terms of cache ways and the quotas are enforced at the time of replacement. If the actual share of a thread becomes larger or smaller than its target quota, then a block owned by the same or other (respectively) thread is replaced. The granularity of enforcing quota can be either set-level or overall cache-level. The advantage of cache-level quota scheme is that it allocates larger cache space to a thread in some portions if it is occupying less space in other cache portions. By comparison, set-level quota scheme enforces quotas at set-level regardless of the cache utilization of a thread in remaining cache. Due to this, set-level scheme also provides stronger guarantees of enforcing quotas than the cache-level scheme.

A limitation of both these schemes is that due to the strict partitioning used by them, if the cache requirement of a thread reduces, the blocks allocated to it may not be reclaimed. To avoid this issue, they employ a mechanism that uses a counter in each set. Every time, a replacement candidate selected by the RP in a set is spared due to its owner's share being less than the target quota, the counter of this set is incremented. When the counter reaches a threshold, the replacement candidate is evicted even if it leads to quota violation. As for OS-level policies to determine quotas, they study statically allocating quotas, dynamically deciding quotas to improve fairness (e.g., miss rate equalization [Kim et al. 2004]) or achieve performance differentiation. They show that their approach avoids the overhead of frequent OS intervention, while still providing flexibility to OS to implement various cache management strategies.

4.2. Set-Based CPTs

Lin et al. [2008] propose page-coloring-based dynamic and static CPTs for optimizing performance and QoS in a two-core system. Their dynamic CPT for performance runs one interval with current partition and one interval each with neighboring partitions (i.e., increasing/decreasing quotas of each core) and selects the partition with the least number of total misses. Their dynamic CPT for QoS seeks to ensure that performance of first application is not degraded by more than a threshold (compared to that with baseline execution of homogeneous workload with equal partitioning) and performance of second application is maximized. This CPT periodically compares the IPC of first application with baseline IPC and if the IPC is lower than baseline IPC by a threshold, then cache quota of first application is increased and if it is already maximum, the second application is stalled. If the IPC is higher than baseline, then a second application is resumed (if it was stalled) or its cache quota is increased. To enforce CP, they divide each list of free memory pages into multiple lists having free pages of the same color. On a page fault, pages are searched from these free lists with allocated colors in round-robin manner. On a change in cache allocation, contents of recolored pages are

moved only when they are accessed. Their static CPT finds the best partitioning by evaluating all possible candidates. Their experiments confirm the effectiveness of their CPTs.

Lin et al. [2009] present a coloring-based CPT that uses a low-cost hardware structure for reducing the page-relocation overhead. For mapping OS pages to cache colors and remembering the mapping, they use a "region mapping table," which groups memory pages into multiple memory regions. Thus, remapping is performed at granularity of memory region since keeping an entry in the table for each page is infeasible. The region mapping table allows mapping a page to any cache color. They use a sampled ATD for obtaining profiling information, which is used to classify applications into different categories based on access frequency, spatial locality of L2 cache accesses of their different memory regions, and the increase in cache misses on sharing a cache color between multiple memory regions. Based on this classification, cache quotas are decided, for example, several regions showing streaming behavior are assigned to a cache color, regions whose working set fit in the cache are given large number of colors, and those showing weak cache sensitivity are given small number of colors. The number of colors allocated to an application depends on the characteristics of its memory regions. These quotas are enforced by reconfiguring the region mapping table. Their CPT improves throughput and fairness compared to shared and statically partitioned cache.

Zhang et al. [2009] present a page-coloring-based CPT that seeks to reduce the repartitioning overhead. Their CPT uses miss-ratio curves to determine cache quota of each application in terms of number of colors. Since page coloring further constraints memory space allocation and recoloring incurs large overhead, their CPT recolors at most a fixed number (Z) of hot pages. Z is computed as the ratio of cost of every recoloring and the time duration between recoloring events. Hot (most frequently accessed) pages of each process are identified by scanning the page table and its overhead is lowered by exploiting the spatial locality of access patterns. They show that by virtue of recoloring only selected hot pages, their technique lowers the overhead of coloring-based CP.

Tam et al. [2007] present a software-based static CPT that uses page-coloring to allocate cache quotas. For each application, they generate L2 miss rate curve and "instruction retirement stall rate curves", which shows the stall cycles due to memory latencies for different L2 sizes. Since a change in miss rate may not directly relate to performance, they note that stall-rate curve is more suitable for guiding CP than miss rate curve. Stall rate curve accounts for factors such as L2 miss rate, sensitivity of instruction retirement stall to L2 misses, memory bus contention and variable latencies of lower levels of memory hierarchy (e.g., L3 and main memory). Also, stall rate curve can be obtained from hardware counters available on a real-system. Their CPT improves performance over shared cache and is useful for determining the pair of applications that can be co-scheduled.

Jin et al. [2009] present a static color-based CPT for providing performance isolation between applications executing in different VMs (possibly running different OSes), which is different from other works (e.g. Lin et al. [2008]), where CP achieves performance isolation between applications of a single OS. They perform CP in VMM by using page coloring approach such that data from different VMs are mapped to different colors and, hence, different sets. They implement their CPT in Xen hypervisor, which is transparent to guest OSes. They show that their CPT improves performance for workloads with a mix of cache-sensitive and cache-polluting applications.

4.3. Block-Based CPTs

Sanchez and Kozyrakis [2011] present a CPT for caches with good hashing and high associativity, for example, zcache [Sanchez and Kozyrakis 2010], which can be modeled

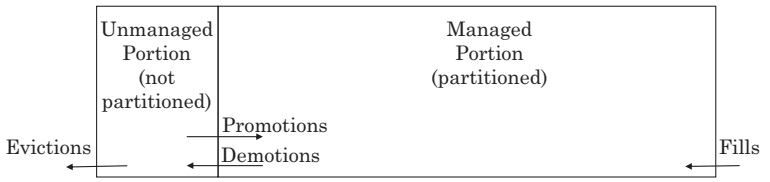

Fig. 4. Division of cache in managed/unmanaged portions and flows between portions in technique of Sanchez and Kozyrakis [2011].

using application-independent analytical models. These caches provide high probability of avoiding evictions from a major fraction of blocks simply by adapting the RP. In their CPT, cache is shared by all the blocks and partition sizes are enforced at replacement time. Since replacing a block from another partition causes interference (even if the block is dead) and replacing from the same partition does not scale with number of partitions, their CPT only requires that eviction/insertion rates of every partition should match on average. However, this still causes interference, and to avoid it, their CPT partitions most (e.g., 85%) and not all of the cache. Their CPT divides the cache into a managed and an unmanaged portion and partitions the managed portion only. Partitions may slightly exceed their quota by acquiring space from the unmanaged portion and not other partitions, and thus, their CPT maintains associativity of each partition irrespective of the number of partitions. On an eviction, blocks in unmanaged portion are preferentially evicted. By using a suitable size of unmanaged portion, evictions from managed portion can be almost completely avoided. Blocks are initially inserted in managed portion, get demoted to unmanaged portion (which only requires changing the tag and not actual movement), and get evicted or promoted again on a hit. This is illustrated in Figure 4.

Their CPT seeks to match the actual quota of a partition with the target (found by an allocation scheme, e.g., "lookahead algorithm" [Qureshi and Patt 2006]) by matching the demotion and insertion rates. On any eviction, all candidates with eviction priorities greater than a partition-specific threshold are demoted. The threshold value depends on insertion rates and sizes of all the partitions and is determined by letting partitions exceed their quotas and then adjusting threshold based on how much they exceeded the quotas. Further, they use time-log-based LRU where each partition has a counter to record its time-log, which is incremented after P accesses (=1/16 of partition's size). Every incoming block gets current time-log of its partition. Using these, they avoid the need of tracking eviction priorities by demoting all candidates below a fixed-point time-log whenever a partition exceeds its quota. The fixed point is adjusted based on number of candidates seen and evicted for every partition. They show that compared to way-based CPT and pseudo-CPT, their technique provides higher performance and scales better to large number of cores (e.g., 32). A limitation of their CPT is that its inability to partition the entire cache may lead to performance loss for some workloads.

Wang and Chen [2014] present a replacement-based CPT for maintaining high associativity even with large number of partitions. They define "futility" as the uselessness of a block to application performance. Different RPs rank futility values differently, for example, OPT (optimal), "least frequently used," and LRU RPs rank the blocks by time of next access, access frequencies, and time of last access, respectively. They show that replacement-based CPTs have tradeoffs between improving associativity and achieving partition (cache quota) targets, since the former requires evicting most-useless eviction candidates, whereas the latter requires preferentially replacing candidates from over-sized partitions. They discuss a "partitioning-first" policy that prefers enforcing quotas over improving associativity. This policy first selects partitions exceeding their

target sizes the most and then replaces the block with highest futility from candidates in this partition. However, due to degrading associativity, this policy does not scale to a large number of partitions.

Their proposed CPT works by controlling the partition sizes by scaling the futility of its blocks. It assigns a "scaling factor" (SF) for each partition and on every eviction, it scales the futility of replacement candidates from a partition by SF. It then evicts the candidate with highest scaled futility. Thus, the futility of blocks of a partition is judged from perspective of entire cache and by adapting SF, the size of a partition can be controlled. Using analytical modeling, they show that under some assumptions, their CPT maintains the partition sizes statistically close to their target sizes and the associativity of a partition does not depend on total number of partitions. They further propose a practical implementation of their CPT, which uses time-log based LRU [Sanchez and Kozyrakis 2011]. The larger the difference between time-log of a block and current time-log, higher is the futility of the block. They also use a feedback-based approach to adapt SF values. For a partition, after a fixed insertions or evictions, if the actual size is larger than the target size and the partition is likely to grow (i.e., insertions outnumber evictions), its SF is increased. Their CPT improves performance significantly and the practical implementation achieves performance close to that expected from analytical model.

5. VARIOUS STRATEGIES FOR CACHE PARTITIONING

In this section, we review several CP strategies, such as static CP, pseudo-partitioning, cache-statistics-based CP, and so on.

5.1. Static CPTs

Stone et al. [1992] present an approach for optimal partitioning of cache for minimizing overall cache miss rate. Under the assumption that the miss ratio curve of each application is convex, they show that optimal quotas are the points where the derivatives of miss rate of the applications become equal. Their algorithm uses greedy-allocation approach such that next block is allocated to the application with largest miss rate derivative until all the blocks have been allocated.

For processors with more than two cores, Brock et al. [2015] present theoretic background and a technique for "partitioning-sharing" (PS) allocation where some cores may share the cache, whereas others may have their private partition. They define "natural partition" such that each application has the same miss ratio in its natural partition as in the shared cache and, thus, the performance of naturally partitioned cache is equivalent to that of shared cache. This formulation allows reducing PS problem to simply partitioning and, thus, optimal solution for partitioning is at least as good as the optimal solution for PS. In other words, although a large number of possible combinations of partitioning-sharing exist, partitioning is usually the best option. They further propose a dynamic programming algorithm for finding optimal partitioning. This algorithm finds the partitions application by application. When a new application A_i is added, it gets c_i amount of cache, which minimizes the sum of its misses and misses of optimal partitioning of first $i-1$ applications with cache size of $C-c_i$, where C is total cache size. Unlike the technique of Stone et al. [1992], their algorithm can optimize for fairness and QoS also, in addition to throughput. Their optimal PS algorithm performs significantly better than shared cache and equal partitioning.

5.2. Pseudo-Partitioning Techniques

Pseudo-partitioning techniques work by controlling the insertion and promotion policies [Xie and Loh 2009; Halwe et al. 2013; Jaleel et al. 2008; Hasenplaugh et al. 2012; Kaseridis et al. 2014] or use decay/reuse-distance-based management to control the

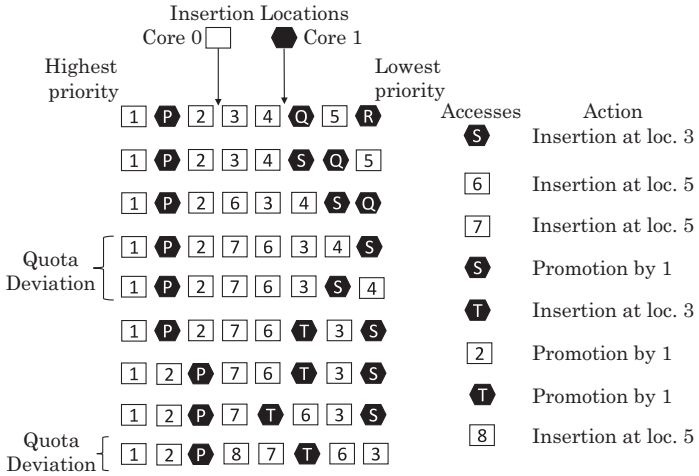

Fig. 5. Illustration of working of pseudo-partitioning scheme of Xie and Loh [2009].

lifetime of blocks in cache [Petoumenos et al. 2006; Duong et al. 2012]. We now discuss some of them.

Xie and Loh [2009] present a pseudo-partitioning technique that works by controlling cache insertion and promotion policies. They first find the cache quota targets of each core (say, Q_i) by using utility monitors [Qureshi and Patt 2006]. Then, new blocks from a core i are inserted at priority location Q_i, and, thus, the quota of a core decides its insertion location. A cache hit promotes a block by one priority step with probability P (with typical values of 3/4, 1, etc.) and, thus, priority remains the same with probability $1 - P$. The "victim selection policy" is the same as that in LRU, that is, the least-priority block is evicted. Blocks of cores with smaller cache quota get inserted at lower position and, hence, they experience more competition for promotion and are likely to get evicted sooner. The opposite is true for cores with larger quota. To avoid cache thrashing from streaming applications, they first detect them by seeing whether their misses or miss rates exceed certain thresholds. For such applications, insertion is made at priority S, where S shows the total number of streaming applications. Thus, irrespective of its target quota, each streaming application is only given one way each. Also, promotion probability for such cores is reduced, which ensures that only blocks showing significant reuse get promotion to higher priority locations. Their CPT is illustrated in Figure 5. They show that by virtue of handling different memory access patterns, their technique provides higher throughput and fairness than CPTs aiming at adaptive insertion or capacity partitioning only. A limitation of their approach of deciding insertion position based on the number of assigned ways is that many partitions may have low insertion positions, which causes severe contention at near-LRU position and difficult-to-evict blocks at near-MRU position.

Jaleel et al. [2008] note that using DIP in shared cache provides better performance than LRU-managed cache, yet it fails to account for behavior of individual threads. Note that DIP policy works by finding the best policy out of LRU policy and BIP, where BIP is a policy that inserts blocks in MRU position with a small probability and in LRU position with a large probability [Qureshi et al. 2007]. They propose a thread-aware DIP that uses heuristics for reducing the overhead of set-dueling monitors in processors with a large number of cores. They show that their technique outperforms thread-unaware DIP and a way-based CPT.

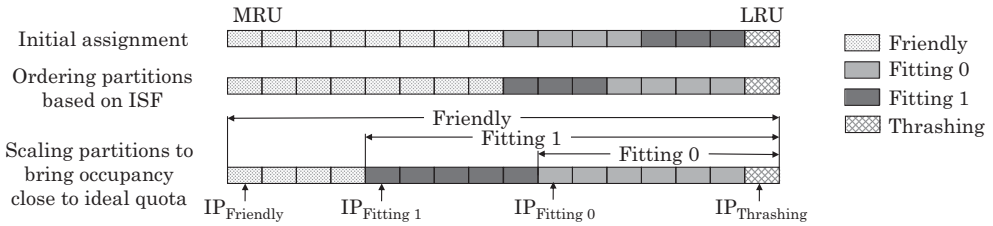

Fig. 6. Working of pseudo-partitioning scheme of Kaseridis et al. [2014] for a 16-way cache with 4 programs.

Halwe et al. [2013] note that in pseudo-partitioning techniques, one core may steal ways of other cores, whereas in strict-partitioning techniques, the ways assigned to some cores may remain unused. They extend the technique of Xie and Loh [2009] and provision that a core with a target allocation of W-ways can donate/steal up to $\lceil W/2 \rceil$ ways to/from other cores. To enforce this rule, they adapt the victim selection policy such that if the actual quota of a core is less than $W + \lceil W/2 \rceil$, then on a cache miss, a block is replaced from other cores who can donate a block, but if the condition is not true, then a block from the same core is replaced.

Kaseridis et al. [2014] present a technique that performs soft-partitioning of cache based on cache-friendliness and MLP behavior of programs. They classify programs into three types: (1) cache fitting (their working set fits in cache but they benefit from cache); (2) cache-friendly (their working set is larger than cache size and they benefit from cache); and (3) thrashing/streaming (those with very large working set size or no cache reuse). Using a sampled ATD for each core, they find the change in miss rate of each program with its cache allocation. From this information, thrashing programs are those where miss rate difference between allocation of single and all the cache ways is smaller than a threshold, fitting programs are those where miss rate falls below a threshold at some cache quota, and remaining are friendly programs. Also, the MLP of program is taken as the number of entries in the MSHR, since it shows the number of outstanding long-latency memory requests at any time from core to memory hierarchy.

Their CPT works in two steps. In the first step, ideal cache quotas of each program are estimated. They first compute "utility rate" for a program defined as the ratio of fraction of cumulative hits for each cache way and MLP. Then, "marginal utility" is defined as change in utility rate for allocation of k additional ways divided by k. Using this metric, cache ways are iteratively assigned to programs with highest marginal utility. In the second step, pseudo-partitioning is performed by modifying insertion and promotion policies with the goal that average cache occupancy of every core remains close to the ideal capacity predicted in the first step. Note that "insertion point" (IP) refers to the position in LRU stack where a line is inserted. Their CPT uses different IPs for different programs, such that thrashing program only gets the last way in LRU stack, and friendly programs get higher IPs than fitting programs. Also, on a hit, a block is promoted to its IP and, thus, it cannot advance more than that. IPs for programs in the same category are decided based on their "interference sensitivity factor" (ISF), whereby programs showing larger number of hits at close-to-LRU positions are considered more sensitive to cache contention. Programs with higher sensitivity get higher IPs and vice versa. Their CPT also monitors the average cache occupancy of a program and if it falls significantly below the ideal quota, the IP of the program is increased in the next epoch to reduce the gap between ideal and actual occupancies. Their technique is illustrated in Figure 6. They show that their CPT improves both throughput and fairness.

Petoumenos et al. [2006] present a technique that controls cache quota of programs by using decay-based management. Their technique is guided by statistical model of

reuse distance, defined as the number of events between two consecutive accesses to an address. For measuring RD, they quantify time in terms of cache replacements, referred to as "cache allocation ticks" (CAT). CAT allows relating time (events) with cache space. This model accounts for both program characteristics (e.g., its RD profile) and cache characteristics (e.g., capacity, RP) to estimate cache hits and misses. They define "spacetime" as the product of lifetime (in CAT units) of a block in cache and the space occupied by the block. Spacetime corresponding to hits and misses are useful and useless space, respectively. The ratio of useful to useless space, called "useful ratio," shows the ability of a program to utilize its cache quota and potential for improvement by any CPT. They define "decay interval" as the time period measured in CAT such that a block not accessed for a decay interval becomes a candidate for replacement irrespective of its LRU status. Based on this information, the cache quotas of programs can be controlled by tuning their DIs based on their cache utility and priority. Thus, the blocks of programs with higher priority and temporal locality can stay in cache for longer duration than those of low-priority and poor-locality. Their technique estimates the misses with different DIs and then chooses DIs that minimize total misses and maximize the total useful spacetime. To achieve QoS, DIs can be selected to ensure that a certain cache quota is allocated to each program. They show that their decay-based approach is effective in managing cache and improving useful ratio. Also, their technique reduces misses more than a way-based CPT.

Duong et al. [2012] note that to avoid cache pollution, a block can be kept in cache only until its expected reuse happens. This reuse distance is called "protecting distance," and it strikes a balance between maximal reuse and timely eviction. At the time of insertion or promotion, the reuse distance of a block is set to PD. On each access to the set, PD values for all blocks in the set are decreased by one, and a block with 0 value becomes a replacement candidate. They note that in a multicore processor, increasing the PD of an application increases its cache quota by slowing down replacement of its blocks. Based on this insight, they propose a CPT that finds PDs for each application such that overall cache hit rate is maximized. Their heuristic-based CPT works on three insights: (1) an application with large single-core hit rate also makes high contribution to multi-core hit rate, (2) multi-core PD of an application is expected to be close to one of the peaks in its single-core hit rate, and (3) only few (e.g., 3) important peaks need to be examined for each application. Using these insights, their CPT sorts the applications based on their hit rate and adds the application with highest hit rate to a list. Then, in an iterative manner, the application with the next-highest hit rate is considered and each of its peaks (found from single-core hit rate) are evaluated in conjunction with peaks of existing applications in the list. Using a search algorithm, peak of the combination maximizing multi-core hit rate is found. Their CPT improves throughput and fairness and scales well with number of cores.

5.3. Cache-Statistics-Based CPTs

While most CPTs are guided by misses or miss rate, some CPTs primarily work on estimated latency impact of misses, while others account for bandwidth and writeback impact of CP. These works are highlighted in Table VII. We now discuss some of these works (also see Section 7.4).

5.3.1. IPC-Based CPTs. Subramanian et al. [2015] present an "application slowdown model" (ASM) to estimate application slowdowns due to interference at shared cache and main memory. Slowdown is defined as the ratio of execution time when running with other applications and that when running alone on the system. They note that accurate estimation of interference-effect on individual requests is challenging since multiple requests are simultaneously served in the memory system. Hence, ASM works

Table VII. Parameters Which Guide the CPTs

IPC or memory latency estimated from misses	[Moreto et al. 2008; Muralidhara et al. 2010; Hasenplaugh et al. 2012; Yeh and Reinman 2005; Jung et al. 2010; Lin et al. 2008; Kaseridis et al. 2014; Tam et al. 2007; Srikantaiah et al. 2009b; Mittal et al. 2014a; Mittal and Zhang 2013; Wang and Martínez 2015; Kasture and Sanchez 2014; Lin and Balasubramonian 2011; Oh et al. 2011; Moreto et al. 2009]
Bandwidth	[Kaseridis et al. 2010; Yu and Petrov 2010; Yeh and Reinman 2005]
Writebacks and misses	[Zhou et al. 2012]
miss rate or misses	Nearly all others

based on overall (and not individual) request service behavior. Based on the correlation between application-performance and its rate of accessing shared cache, ASM estimates slowdown as C_{alone}/C_{shared}. C_{shared} can be easily measured and to measure C_{alone}, they perform two steps at regular intervals. First, to remove the impact of memory BW contention, they give highest priority to application's requests at memory controller for a short duration. This also helps in finding cache miss penalty in absence of main memory interference. In absence of a request from prioritized application, another application's request may get scheduled, which may delay the request of prioritized application and therefore, ASM also accounts for and subtracts the effect of such queuing delay. Second, to estimate the effect of shared cache interference, they find extra time consumed in serving contention misses (i.e., those that would hit in alone execution). This is estimated based on number of contention misses (obtained from a sampled ATD) and average time spent in serving cache misses (computed in first step) and hits. To estimate C_{alone}, these extra cycles are subtracted from the time for which application was given highest priority at memory controller. They show that ASM is reasonably accurate in estimating slowdowns.

Based on ASM, they further propose a CPT, which seeks to minimize slowdown. Their CPT first estimates the slowdown of every application for different number of ways. If C_k shows cache access rate of an application with k ways, then the slowdown with k ways is C_{alone}/C_k. C_k is computed from the number of hits and misses in shared cache during an epoch and the estimated number of cycles required to serve these accesses with k ways. After estimating slowdowns for different ways, they compute marginal slowdown utility as $(slowdown_{n+k} - slowdown_n)/k$. Then, cache is partitioned using "lookahead algorithm" [Qureshi and Patt 2006] except that it uses "marginal *slowdown* utility" instead of "marginal *miss* utility" as in Qureshi and Patt [2006]. They show that their CPT provides higher performance and fairness compared to shared cache and miss rate minimization based CPTs.

Moreto et al. [2008] note that performance impact of a cache miss depends on MLP of L2 misses of the application. Clustered L2 misses fitting in ROB can be simultaneously served and, thus, they incur nearly the same miss latency as an isolated L2 miss, as illustrated in Figure 7. Hence, their CPT assigns higher "MLP penalty" to isolated misses than clustered misses. Using Mattson's stack algorithm [Mattson et al. 1970], they find cache misses for different number of cache ways. Then, they compute the performance impact of a cache miss converted into hit and vice versa on an increase or decrease in cache size, respectively. If the number of cache ways of a core are increased, then some misses will become hits. For finding their MLP penalty, they store the LRU stack distance (D_i) and owner-core of this miss in the corresponding MSHR entry. In each cycle, they find for every possible value of D_i, the number of L2 accesses with stack distance no less than D_i. Thus, if J misses are simultaneously served, the cost of each miss is assigned as $1/J$. Similarly, to find the MLP penalty of a hit that would turn into a miss, they find a number of accesses with stack distance no less than D_i (including L2

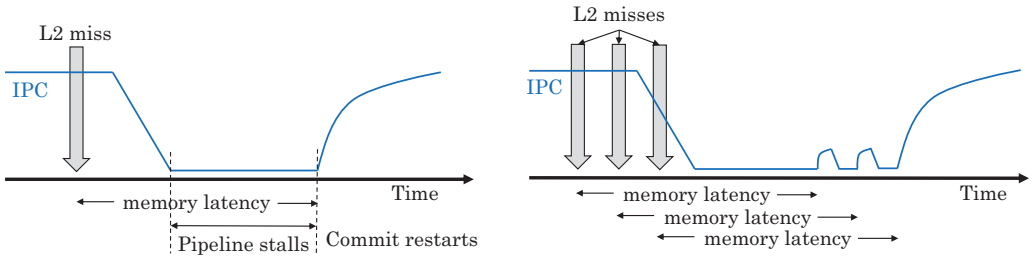

Fig. 7. Clustered misses incur nearly same miss penalty as an isolated miss [Moreto et al. 2008].

misses and hits). Based on this information, length of the miss-cluster is estimated. L2 instruction misses stall the fetch and, hence, they have a fixed miss latency and MLP penalty. Based on the above ideas, they evaluate different partitions and find one with minimum total "MLP penalty" and use it in the next epoch. They also study a variant of their technique, where MLP costs are multiplied with IPC of the running application to give larger weight to higher-IPC applications for optimizing throughput at the cost of fairness. They show that their techniques provide higher throughput compared to LRU-managed cache and an MLP-unaware CPT.

Moreto et al. [2009] present a CPT that allows optimizing for throughput, fairness, or per-application QoS metric. They use a sampled ATD for estimating L2 misses for different number of allocated ways and obtain corresponding IPC values from the CPI stack model of application performance. Based on this, their CPT allocates cache ways to threads to meet their QoS requirements. Remaining ways are distributed between threads to meet overall requirement (e.g., fairness or throughput). Thus, their technique can translate QoS targets into cache quota requirements. Their CPT is effective in optimizing a desired objective (fairness or throughput) and, in most cases, can also allow running an application at a desired percentage of their maximum IPC (achieved when running alone with entire cache), irrespective of co-running applications.

5.3.2. Bandwidth-Based CPTs. Yu and Petrov [2010] present a CPT that focuses on reducing off-chip bandwidth requirement instead of primarily reducing misses since the former has direct impact on performance. They note that an application with a large number of misses may not consume largest off-chip bandwidth (amount of data transferred in unit time). Applications with a smaller number of misses may require larger bandwidth if their memory accesses are clustered together. Based on this, cache quota of applications with lower bandwidth can be reduced. They use static set-level CP found through an offline heuristic algorithm whose input is application bandwidth for different cache sizes. The algorithm iteratively evaluates different partitions to find the one with least overall bandwidth requirement. In every step, the algorithm searches the task showing largest reduction in bandwidth demand on increasing the cache quota. This larger cache quota is confirmed for that task. By iterating in this manner, quota for remaining tasks is searched within the remaining cache space. They show that their technique is effective in reducing BW demand.

6. CPTS FOR ACHIEVING VARIOUS OPTIMIZATION OBJECTIVES

In this section, we discuss CPTs in terms of their optimization targets.

6.1. CPTs for Improving Fairness or Implementing Priorities

Kim et al. [2004] define five metrics for measuring fairness of co-scheduled applications, based on cache *misses* and *miss rate*. They show that two of these metrics relate strongly to *execution-time* fairness. They further propose a static and a dynamic CPT that use these metrics to optimize fairness. The static CPT requires a profiling run, during which cache misses for different numbers of ways are estimated assuming LRU RP. Based on it, the total number of misses under different possible partitions is estimated and a partition is chosen, which provides the optimum value for a selected fairness metric. The dynamic CPT repeatedly performs two steps: "quota allocation" and "adjustment." In quota allocation step, the chosen fairness metric is evaluated for all applications. Based on this metric, two applications showing least unfair and most unfair impact of partitioning are selected and if the unfairness difference between them exceeds a threshold, then a fixed cache space is transferred from an application showing smaller unfairness to that showing larger unfairness. Then, after excluding these two applications, the process is repeated for remaining applications. In adjustment step, the decision made in quota allocation step is committed or reverted if the reduction in miss rate of the application receiving increased quota is more or less (respectively) than a threshold. They show that maximizing fairness generally boosts performance, whereas optimizing throughput may not optimize fairness, since throughput improvement may come from unfairly increasing the quota of some applications.

Srikantaiah et al. [2009b] use feedback control theory to partition cache for improving fair speedup and provide service differentiation. Assuming that performance targets (e.g., IPC) are provided for each application, their "partitioning controller" determines new targets to maximize cache utilization. Each application also has an "application controller," which independently ascertains the cache share (in terms of cache ways) required to track those new targets. In case the sum of cache quotas determined by these controllers exceeds the cache size, a negotiation module is invoked that negotiates between requested quotas for one of the specified objectives: for service differentiation, it reduces quotas of low-priority applications first, and for fair-speedup improvement, it reduces quotas of applications in fair manner (i.e., in proportion to their current quotas). Similarly, if total cache quotas are less than the cache size, their technique increases cache quotas to maximize cache utilization using an approach opposite to that mentioned above. Thus, their technique determines feasible cache quotas to produce output-performance targets. These output targets are used again by application and partitioning controllers in the next epoch. Each application controller is designed as a PID controller. Based on the performance achieved in previous epoch with its quota, it finds the quota required for upcoming epoch to meet the performance target. However, PID controller does not account for system history and domain-specific knowledge and, hence, it may introduce oscillations in cache allocation. To address this, they use a model to estimate performance (IPC) as a function of cache quota (number of ways). The insights provided by the controller help in avoiding oscillations by tracking the history of control decisions. They show that their fair-speedup and service differentiation mechanisms are highly effective in optimizing respective targets.

Wang et al. [2012] present a CP and capacity-scaling technique for limiting the maximum power of LLC in a power-constrained multicore and achieving fair or differentiated cache access latencies between different programs. Based on feedback control theory, they use a two-level synergistic controllers design. First, the "LLC power controller" runs at coarse-granularity (e.g., 10M cycles) and it seeks to limit the maximum LLC power for a given budget (e.g., 80% of the maximum power) by controlling the number of operational LLC banks. Remaining banks are power-gated. Second, the latency controller runs at finer time-granularity (e.g., 1M cycles) and it controls the ratio of latencies between two programs on every pair of neighboring cores. For achieving

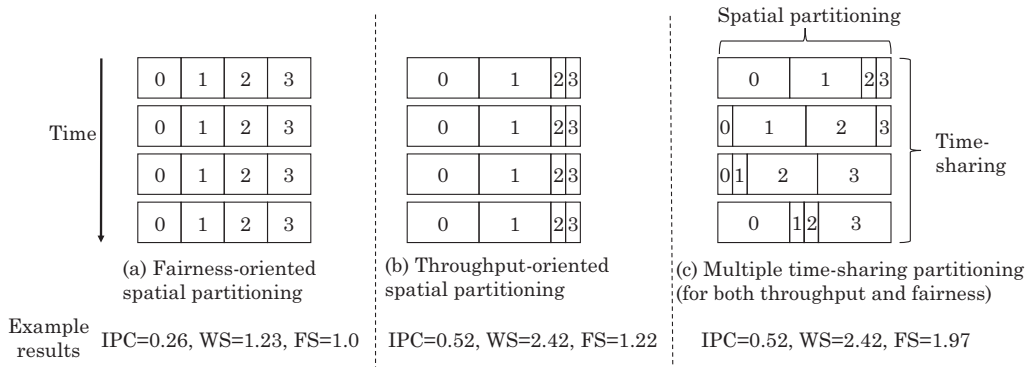

Fig. 8. Spatial partitioning ((a) and (b)) and multiple time-sharing partitioning (c) [Chang and Sohi 2007] schemes. Example results for a 4-core workload (art-art-art-art) highlight that scheme (c) can improve both fairness and throughput.

fairness or differentiation, it seeks to achieve the same cache access latencies for all programs or shorter latencies for higher-priority programs (respectively), even when the total cache capacity is changing. Based on control theory, it computes the relative number of banks to be allocated to each program, and then an actual number of banks is computed based on total available banks. Both controllers are designed as "proportional-integral" (PI) controller and their parameters are determined by experimenting with a few representative programs and then using curve-fitting. Their design provides theoretical guarantee of accurate control and system stability. Their experiments show that their technique can control LLC power consumption and cache latencies of programs very accurately and also allows exercising tradeoff between performance and power consumption. Also note that Srikantaiah et al. [2009b] and Wang et al. [2012] use PID and PI controllers, respectively.

Chang and Sohi [2007] present a CPT for improving throughput and fairness while also providing QoS. They note that there exists a tradeoff between different optimization targets, such that optimizing throughput harms fairness and vice versa (refer to Figures 8(a) and (b)). Consequently, instead of using a single unfair or low-throughput partition, their CPT allocates different-sized partitions to different applications in different time intervals. Thus, the quotas of applications are expanded and contracted in different epochs, as illustrated in Figure 8(c). Since a thrashing application (those with a working set larger than equal partition, which benefit greatly from increased cache capacity) already has low throughput, reducing its quota in contraction epoch does not reduce its performance significantly, but increasing its quota in expansion epoch boosts its performance greatly, which compensates the slowdown in contraction epochs. Expansion opportunity can be given to different applications in equal manner to achieve fairness and in differentiated manner to implement priority. Also, QoS can be guaranteed over long term by allocating quotas that *on average* bound application's slowdown compared to equal-partitioning case. They further note that when applications do not show destructive interference, LRU-based cache sharing may outperform CP. Therefore, they propose combining their CPT with an LRU-based cache management strategy [Chang and Sohi 2006]. The execution period is divided in those controlled by CPT or sharing technique, depending on how many applications benefit from each of them. They show that this technique improves performance of applications with and without interference.

Cook et al. [2013] present static and dynamic CPTs for dual-core system, where one core is prioritized over another. Their CPTs aim that the performance of the prioritized

core is not harmed while the unprioritized core can make maximum progress. Their static CPT evaluates all combinations of way-assignments to two applications and of those with least degradation in performance of prioritized core, finds one with highest performance of unprioritized core. Compared to the shared cache and equal partitioning, this CPT reduces performance degradation of prioritized core. They also present a dynamic CPT that executes when application phase changes, that is, when change in LLC miss rate for a fixed time period exceeds a threshold. The CPT begins with giving all except one way to prioritized core and starts reducing its way allocation until the change in "miss-per-kilo-instruction" exceeds a threshold. Remaining ways are given to unprioritized application. Compared with static CPT, dynamic CPT achieves comparable performance of prioritized core, while significantly improving performance of unprioritized core.

6.2. CPTs for Load-Balancing in MT Applications

Pan and Pai [2013] note that set-specific RD (SSRD) histograms of threads in an MT application have knees at specific RDs and flat regions in-between. Thus, flat regions do not contribute significantly to miss rate reduction. In several cases, knees occur at RDs such that equally partitioning the cache among threads does not provide highest performance, even if threads show similar IPC and data reuse. Their CPT works by temporarily using unequal partitions to improve cache utilization based on symmetry of threads and the shapes of their SSRD curves. If a thread is presently operating away from a knee, then its cache quota is stolen and given to a single "favored" thread to bring it to the next knee. The overall advantage to favored thread is more than the loss to other threads if their quota lies in flat portion of the curve. Further, for some cache sizes, augmenting the quota of favored thread more than that inferred from RD profile boosts the performance of other threads, even though their quota is reduced. This happens due to two factors. First, due to continuous variation in quotas during execution, threads generally have data left in partitions of other threads. Second, overallocation to favored thread lowers its evictions and, thus, left-over data of unfavored threads in partition of favored thread also see reduced eviction. Thus, a large number of accesses to unfavored threads hit in the partition of favored thread, which offset the reduction in hits in their own partitions. Since favoring a single thread leads to unfairness, their CPT selects favored thread in a "round-robin" manner, based on symmetric nature of application. Thus, by temporarily unbalancing the cache quota of all threads, their CPT improves cache utilization and throughput while ensuring load-balancing.

Muralidhara et al. [2010] present two CPTs for boosting slowest thread in MT applications. The first CPT records CPIs of all threads and then, partitions cache ways such that a larger number of ways are given to threads with higher CPI and vice versa. This is illustrated in Figure 9. Since this CPT does not account for the thread's cache sensitivity, their second CPT builds a model of how CPI varies with cache quota. For accomplishing this, first the CPT is used for two intervals. From the values of CPI for different cache ways recorded in these intervals, a performance model is built for each thread by curve fitting using "cubic spline interpolation." Using these models, the CPT repeatedly transfers one way from the fastest thread to the slowest thread until some other thread becomes the slowest (found from performance models). At this point, cache allocation is reverted by one-step and this partitioning is accepted. The second CPT outperforms the first CPT, shared cache, and equally partitioned cache.

6.3. CPTs for Saving Energy

CPTs save dynamic energy by reducing off-chip misses and allowing early completion. CPTs can ascertain the total cache demand of all the applications, and when this is

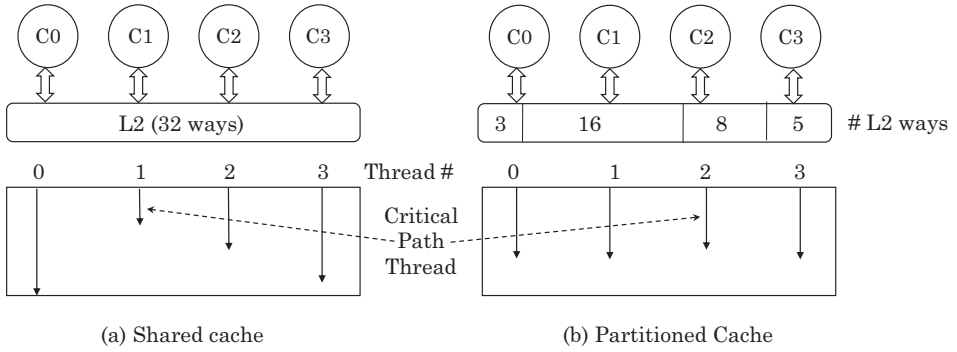

Fig. 9. Use of CP for accelerating critical path thread in MT applications [Muralidhara et al. 2010].

smaller than the overall capacity, the remaining cache can be power-gated and, thus, CPTs also can help in saving leakage energy. We now discuss CPTs proposed for saving energy.

Mittal et al. [2014a] present a CPT that uses dynamic profiling to estimate energy consumption of running applications at different LLC sizes. Assuming the number of sets in LLC as X, their profiling unit estimates miss rate for caches of *different number of sets*, viz., X, $X/2$, $X/4$, $X/8$, and so on, whereas the profiling unit in Qureshi and Patt [2006] estimates miss rate for *different number of ways* of a cache with X sets only. From these miss-rate estimates, miss rate at other set counts is estimated by using linear interpolation. Using this, energy values for different cache partitioning configuration are estimated, such that quota of an application with small utility is lowered or is increased by only a small amount. Thus, in some configurations, all the colors may not be allocated to cores. Based on these, a cache partitioning configuration with minimum energy is selected for the next interval and the unused cache colors are power-gated for saving energy. Their technique provides higher energy saving and performance than thread-unaware leakage energy saving techniques.

Sundararajan et al. [2012] present a CPT for improving performance and saving energy in LLC. They use "lookahead algorithm" [Qureshi and Patt 2006] to determine cache quotas, except that ways are assigned to a core only if the reduction in miss rate from this exceeds a threshold. Thus, some ways may not be allocated to any core and they can be power-gated to save leakage energy. Further, their CPT enforces way-alignment such that in all the sets, data-blocks of a core are stored in its own ways only (refer to Figure 10). By using this information, on a cache access from a core, only the ways allocated to it need to be accessed, which saves dynamic energy. Their CPT brings large improvement in performance and energy efficiency.

Kotera et al. [2011] present a way-based CPT that saves power by power-gating cache ways. They compute locality of an application as the ratio of number of accesses to LRU blocks and MRU blocks. A lower value of this ratio shows higher locality (refer to Figure 11) and, thus, an application with most hits at MRU location requires few ways to still achieve high hit rate. By comparing this ratio for two applications, the decision about increasing or decreasing the number of ways allocated to them is taken. Also, by comparing the ratio with a higher and a lower threshold, the decision about power-gating is taken. By adapting these thresholds, their technique can favor either performance or energy optimization. A limitation of their technique is that it is only applicable to 2-core system.

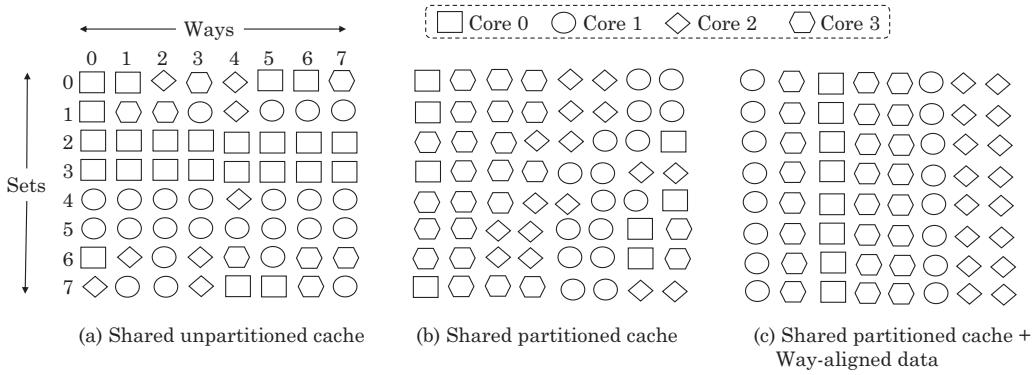

Fig. 10. Data layout in (a) unpartitioned cache, (b) partitioned cache, and (c) partitioned cache with way-alignment [Sundararajan et al. 2012].

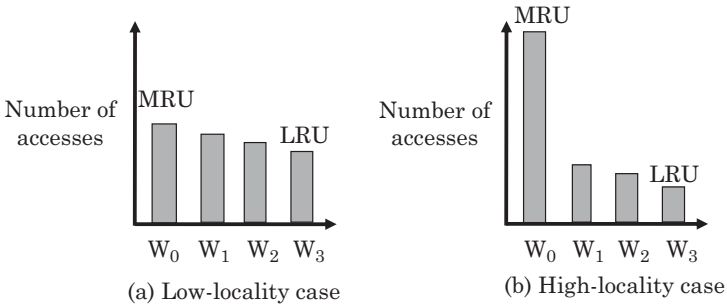

Fig. 11. Determining locality based on accesses to different ways [Kotera et al. 2011].

6.4. Reducing Overhead of CPTs

Xie and Loh [2010] note that CPTs generally incur large hardware overhead for profiling and latency overhead for algorithm execution. However, only one or few thrashing applications are responsible for interference in a shared cache and, hence, restraining these applications with a simple technique can provide performance benefits similar to complex CPTs. In their proposed technique, an application showing large cache accesses along with large solo miss rate or number of solo misses in a time period is identified as thrashing. Solo misses and solo miss rate are estimated from ATD and "large" or "small" are decided based on respective thresholds. Then, their technique allocates a fixed and small (e.g., 2) number of ways to each thrashing application. Remaining ways are shared by remaining applications, and thus, no CP is performed for those ways. Their technique only requires identifying the thrashing application and, thus, avoids the complexities of usual CPTs. By allowing better utilization of shared cache, their technique provides higher performance than complex CPTs, which use strict partitioning. To further reduce the overhead of their technique, they remove the ATD and simply use the absolute number of misses of each application to identify the thrashing application, which still provides reasonable performance. The limitation of their technique is that it cannot guarantee QoS/fairness and may not work for general case of applications with diverse cache requirements.

Kedzierski et al. [2010] present an approach to collect profiling information with pseudo-LRU RPs, such as "not recently used" (NRU) and "binary tree" for guiding way-based CPTs. Their work reduces the overhead of way-based CPTs by relaxing the requirement of true-LRU. The reader is referred to Kedzierski et al. [2010] for the

Table VIII. CPTs Proposed for Various Contexts

NUCA cache	[Chang and Sohi 2007; Herrero et al. 2010; Awasthi et al. 2009; Kaseridis et al. 2009; Jung et al. 2010; Yeh and Reinman 2005; Lee et al. 2011; Wang et al. 2012; Dybdahl and Stenstrom 2007]
PV-affected cache	[Kozhikkottu et al. 2014]
Multiple shared caches	[Kandemir et al. 2011a, 2011b]
Non-volatile main memory	[Zhou et al. 2012, 2016]

working of these RPs. For NRU, they compute U, the number of blocks with used bits set to 1. Thus, if an accessed block has its used bit as 1, it implies a stack distance between 1 and U and to account for over-estimation compared to true-LRU, they scale it with a factor x (≤ 1). If an accessed block has its used bit as 0, then it implies a stack distance between $U + 1$ and W_C and, hence, they assume the stack distance to be W. From these estimated values, they generate a stack distance histogram for guiding the CPT. Also, after CP, NRU is used for cache blocks owned by the core, for example, if the used bits of all the blocks of the core are set, then, all these bits are reset except that just accessed. They also propose profiling logic for binary-tree based LRU. Compared to CPTs using true-LRU, CPTs using pseudo-LRU incur small performance loss, which increases with the number of thread (e.g., 7% with 8-threads).

Nikas et al. [2008] present a "Bloom filter" (BF)-based approach to find cache sensitivity of application to guide CP. For each core, they add a BF in each set. When a tag is evicted, its M LSBs (least significant bits) are used to index a BF entry, which is changed to 1. On a cache miss, corresponding BF entry is consulted using M LSBs of the tag, and if this stores 1, it is termed as "far-miss," indicating that assigning one extra way to the core *may* (due to false positives in BF) have converted this miss into a hit. The number of far misses is scaled to reduce the effect of false positives. They also monitor hits to LRU position for each core, which shows the number of hits that would convert into miss on taking away one way from the core. Based on these, in each iteration, their heuristic CP algorithm compares the smallest hit-gain with largest hit-loss for any core. If the latter is greater, then cache quota of core corresponding to "far-miss" counter is augmented by one way, and quota of core corresponding to "LRU counter" is reduced by one way. The iteration stops when no largest gain is lower than the smallest loss or all cores have been considered. They show that their CPT is effective in improving performance.

7. CPTS IN DIFFERENT CONTEXTS

Table VIII summarizes the works that present CPTs for specific contexts. We now review them.

7.1. CPTs in Context of NUCA Cache

Jung et al. [2010] present a CPT for 3D-stacked NUCA caches. They note that in NUCA cache, the memory access time depends on both number of misses and hit latency (which depends on the distance between core and cache bank). They model the problem of minimizing average memory access time as a graph coloring problem where assigning a cache bank to a core is equivalent to coloring the bank with same color as the core. Since solving this problem becomes infeasible for large number of cores, they propose a heuristic approach. First, the banks closest to the core are allocated to those cores. Then, for each unallocated bank, out of all the neighboring allocated banks, that bank ($B^\star$) is searched such that assigning an unallocated bank to the same core as this allocated bank ($B^\star$) leads to highest performance improvement. This is repeated until all the banks have been allocated. They show that their CPT provides larger performance than equally partitioned cache and a latency-variation-unaware CPT.

Herrero et al. [2010] present a CPT for tiled chip multiprocessors running MT applications with shared data. The processor has multiple independent L2 caches logically separated into a private and a shared portion. Blocks evicted from local L1 are stored in private regions and blocks spilled from neighboring caches are stored in shared regions. This allows creation of large private caches if applications have similar cache needs and large shared caches if additional capacity is beneficial for few applications only. Shared portions store unique blocks but shared data may be replicated in private portion upon a request. Thus, the relative size of private/shared portion control the amount of replication. They propose a node-local CPT scheme for controlling the sizes of private and shared portions, without requiring centralized components or heavy inter-node communication. They use a counter that is incremented or decremented on a hit to LRU of private or shared portion, respectively. At the end of an epoch, if the counter is lower/higher than a low/high threshold, their CPT adds a shared/private way, respectively. They also use a spilled block allocator that attempts to send more evicted blocks to L2 caches with larger shared portion. Their CPT improves performance and energy efficiency and by virtue of allowing distributed repartitioning also affords better scalability.

7.2. CPTs in Context of Process Variation

Kozhikkottu et al. [2014] note that within-die PV [Mittal 2016a] can lead to unequal core frequencies in a multicore CPU (e.g., 2.5, 2.4, 2.2, and 1.8 GHz in a 4-core CPU) and limiting the frequencies of faster core(s) reduces the performance. They present a PV-aware CPT that allows each core to run at its highest frequency and addresses the resulting performance difference by CP. They experiment with MT applications with synchronization barriers. Their CPT works in two steps. In the first step, they characterize the performance of all the threads by varying their cache quotas and discounting the time spent in stalling at barriers. In the second step, way-based CP is done to maximize the performance of the slowest thread. They model the spatial CP problem as a dynamic programming problem and further refine the solutions by temporally perturbing the quota at reconfiguration intervals to find a partition that maximizes the lowest throughput of any thread. They show that their CPT provides higher performance than shared and equipartitioned caches.

7.3. CPTs in Context of Multiple Levels of Shared Caches

Kandemir et al. [2011b] present a technique for simultaneously partitioning multiple shared caches (e.g., L2 and L3) for boosting the slowest thread in MT applications where threads show different cache demands. They note that since miss rates of L2/L3 show correlation and the penalty of their misses are different, misses or miss rates are not suitable for guiding CP. Hence, they use "average memory access time" (AMAT) for guiding CP, which is computed as the number of cycles consumed in memory instructions divided by the number of memory instructions. In their technique, OS binds an auxiliary thread with every program. Based on the performance with the L2 ways and L3 ways allocated to a thread in current and previous epochs, the auxiliary thread builds a performance model for each thread showing AMAT for different numbers of L2 and L3 ways. This is done by fitting a surface model using regression approach. Based on this model, the slowest and fastest threads (in terms of AMAT) are found, and if their performance differs by more than a threshold, cache ways are transferred from the fastest to the slowest thread. Their CPT outperforms shared cache, equal CP, best static CP, partitioning of L2 only or L3 only and uncoordinated partitioning of L2 and L3 based on independent models of their performance.

Table IX. Integration/Interaction of CPT with Other Approaches

Category	References
Processor partitioning	[Srikantaiah et al. 2009a; Lo et al. 2016]
DRAM-bank partitioning	[Liu et al. 2014]
DRAM-bandwidth partitioning	[Lo et al. 2016; Oh et al. 2011; Liu et al. 2010; Bitirgen et al. 2008; Zhou et al. 2016]
DVFS	[Wang and Martínez 2015; Lo et al. 2016; Bitirgen et al. 2008]
Cache block-size selection	[Gupta and Zhou 2015]
Replacement policy selection	[Zhan et al. 2014]
Prefetching	[Hasenplaugh et al. 2012; Jaleel et al. 2008; Gupta and Zhou 2015]
Cache locking	[Suhendra and Mitra 2008]
Network traffic management	[Lo et al. 2016]

7.4. CPTs in Context of PCM Main Memory

For a processor with PCM main memory, Zhou et al. [2012] present a CPT that seeks to minimize both misses and writebacks since PCM has high write energy/latency and low write endurance [Mittal et al. 2015, 2014b]. They note that a write access may lead to writeback. Also, a writeback does not happen if the dirty block remains in the cache till the next write to the same block. They define "writeback avoidance distance" (WAD) of a write W_i to block B as the highest stack distance of all accesses to B between W_i and next write to B. Thus, a write results in writeback if the cache associativity is less than its WAD. Further, similar to stack property for LRU, a write not leading to writeback in M-way cache also does not lead to writeback in a cache with more than M ways. Based on these, they extend utility monitors [Qureshi and Patt 2006] to also track dirty blocks and WAD, and using these, they estimate writebacks for different number of cache ways. From these, their CPT finds a partition for maximizing the weighted sum of the number of avoidable-writebacks and number of hits. By changing the weights, their CPT can optimize different targets, for example, throughput, PCM energy, and so on. Since optimal values of weights for different applications vary, they propose periodically adjusting these weights based on write queue congestion and write queue occupancy. By virtue of reducing writebacks, their CPT brings large improvement in throughput, energy efficiency, and PCM lifetime over shared cache and writeback-unaware CPT.

8. INTERACTION/INTEGRATION OF CPT WITH OTHER TECHNIQUES

Several researchers integrate CPT with other management approaches or study their interaction to exploit the synergy between them and benefit a larger range of applications. Table IX summarizes these works and we now discuss a few of them.

8.1. Integration with Processor Partitioning

Srikantaiah et al. [2009a] note that when the variation in degree of "thread-level parallelism" (TLP) between applications is high, equally distributing processors between them may not lead to optimal performance. Further, applications show varying requirements of compute and memory resources, and so, they propose performing "processor partitioning" (PP) and CP in integrated manner (refer to Figure 12). Since the number of possible combinations of PP and CP is very large, they use two insights to prune the search space. First, compared to CP, PP has larger impact on fair speedup and based on this, they use an iterative approach where PP is done first and is followed by CP. This also allows using any CP scheme with PP scheme. Second, there is a strong correlation between fair speedup and QoS metric (namely, the degradation in application performance should not exceed a threshold, e.g., 5%), and based on this, partitions deemed unacceptable on QoS metric can be removed without losing candidates that are good

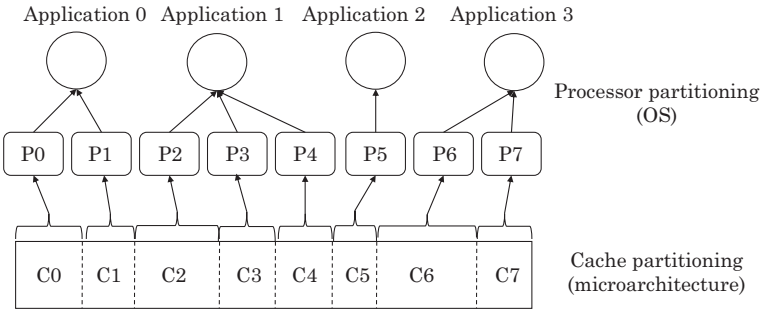

Fig. 12. An example of coordinated CP and PP with 4 applications, 8 processors, and shared LLC [Srikantaiah et al. 2009a].

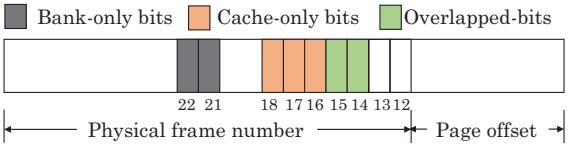

Fig. 13. Bank-only (21-22), cache-only (16-18) and overlapped (14-15) bits on a processor with 8GB memory and 64 banks [Liu et al. 2014].

on fair speedup metric. They evaluate application IPCs for two cases: equal partitioning of processors and "alternate partitioning" formed by alternatively increasing and decreasing processor shares in equal partitioning. These values are fed to a regression predictor that uses the "least-square error" method to find IPC for any processor share. Based on this, optimal PP on fair speedup metric is found while pruning partitions unacceptable on QoS metric. Similarly, for CP, they first record IPCs for equal and alternate partitioning of cache and then use a regression predictor to find IPC under any cache share. Using this information, optimal CP is found without violating QoS constraint. PP is done at coarser granularity (65M cycles) than CP (10M cycles), and thus, multiple cache partitions may be enforced for the same PP. They show that their technique performs better on fair speedup than the implicit partitioning enforced by OS and the equal partitioning. Also, integrating PP and CP performs better than using either of them individually.

8.2. Integration with DRAM-bank Partitioning

Liu et al. [2014] present a "vertical partitioning" (VP) approach, which partitions both LLC and DRAM banks to account for cache/memory needs of diverse range of applications. They note that in physical address, few (e.g., two) bits are common among those used for computing LLC set index and those for computing DRAM bank. This is illustrated in Figure 13. These common bits allow partitioning both cache and banks and by using these bits along with one of the bits from bank- or cache-only bits, VP techniques with different partition granularity can be obtained.

They show that for several workloads, VP provides higher performance than cache/bank-only partitioning. Also, for MT applications with high data sharing, random-interleaved page allocation performs better than any partitioning. They evaluate these techniques over a large number of workloads and use data mining approach to determine partitioning and coalescing rules. For example, one partitioning rule is that workloads with combination of LLC thrashing and other types of applications should use VP controlling common-bits only or VP controlling common-bits with cache-bits. Similarly, workloads with high and medium cache intensity applications but without

thrashing applications should use bank-only partitioning. MT applications use random-page allocation. To bring the best of both partitioning and sharing together, they form coalescing rules, which allow merging partitions of certain application types. For example, the partitions of high and medium cache intensity applications are shared, and those of cache-fitting and thrashing applications are shared. Thus, flexible partitioning and coalescing allow accounting for arbitrary beginning and termination of applications. They implement their VP approach in Linux kernel and show that it provides higher performance than unmodified Linux and cache-only and bank-only partitioning.

8.3. Integration with Bandwidth Partitioning

Liu et al. [2010] study the interaction of CP and BW partitioning. They formulate an analytical model of BW partitioning, which accounts for a number of L2 misses, instruction- and memory-level parallelism, queuing delay, contention from multiple cores, and so on. From this model, they observe that whether BW partitioning can improve performance depends on the difference in miss frequencies between applications and with decreasing bandwidth, the scope of performance improvement increases. Hence, CP may lower the impact of BW partitioning on performance by reducing the difference in miss frequencies of applications and by reducing total cache misses, which relieves BW pressure. However, if CP increases the difference in miss frequencies, then it can increase the impact of BW partitioning on performance. Thus, for example, for cache insensitive applications, CP cannot improve performance, but by virtue of changing the difference in miss frequencies, CP enhances the effectiveness of BW partitioning in boosting performance. Their microarchitectural simulation experiments confirm the insights obtained from analytical model.

8.4. Integration with DVFS Scheme

Wang and Martínez [2015] model the problem of dividing shared resource (chip power budget and LLC capacity) between the applications as a dynamic distributed market, where each application running on a core is an agent and the resource prices change based on "demand" and "supply." Initially, each agent has a budget to purchase resources and it develops a model of performance as a function of allocated resources. A global arbiter fixes initial prices of all resources, and each agent bids for the resources to maximize its utility (performance). Based on all these bids, the arbiter increases and reduces the price of resources in high and low demand, respectively. Agents bid again under new prices, and this iterative process stops upon convergence of the market, that is, when change in price within iterations is very small (or a threshold number of iterations have been performed) and a change in bid by an agent does not improve its own utility. At this point, resource-allocation is performed. Thus, compared to other techniques that only account for marginal utility of a resource, their technique also accounts for how contended a resource is. Thus, if a resource "P" is highly priced (i.e., highly contended), then the agent will start bidding for a cheaper resource (i.e., less contended), even though "P" may provide higher "marginal utility" to the agent. All agents work in decentralized manner and the only centralized function in their technique is pricing scheme, which is relatively simple. Also, for optimizing throughput, larger budgets can be assigned to agents with larger marginal utility, and for optimizing fairness, equal budgets can be assigned for all agents. They compute the cache utility values by dividing execution time into compute and memory phase, estimating misses under different ways using sampled ATD and finding the length of memory phases for each of these miss-counts. They also find power-utility values for change in frequency using DVFS. They show that their technique is effective in improving throughput and

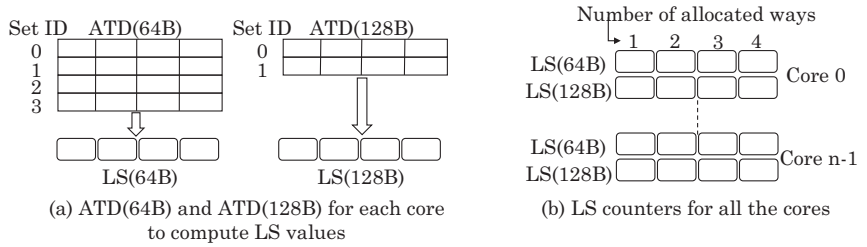

Fig. 14. Per-core ATDs for different block sizes (256B and 512B sizes omitted) [Gupta and Zhou 2015].

fairness and scales to large number of cores (e.g., 256) by virtue of using decentralized approach.

8.5. Integration with Cache Block-size Selection

Gupta and Zhou [2015] note that for several memory-intensive applications, use of large block size brings significant reduction in capacity requirement and, thus, working set size (or capacity requirement) depend crucially on block size. Due to this, a smaller cache with large block size can provide higher performance than a much bigger cache with a smaller block size. Thus, using large block size for such applications with strong locality allows them to donate capacity to others, which can boost the performance of applications with good temporal locality. Based on these insights, Gupta and Zhou [2015] present a technique that dynamically changes both cache block size and cache quotas of applications to leverage both spatial and temporal locality for improving performance. For a unified measurement of both temporal and spatial locality, they compute "locality score" (LS), which, for a recently accessed address, measures the probability that a neighboring address will be accessed in near future. LS depends on the size of neighborhood (i.e., cache block size) and near-future window (i.e., reuse distance). For each core, they use ATDs to compute LS values for each block size (64B, 128B, 256B, and 512B), as shown in Figure 14. Since LS indicates hit rates, for improving the throughput, their CP algorithm seeks to maximize weighted sum of LS values of all the running applications under the constraint of total cache capacity. The weight for any application is its LLC access rate. For each application, their algorithm determines the suitable block size for each possible quota allocation. A higher block size is chosen only when the extra memory traffic generated by it is justified by significant improvement in LS. Then, "lookahead algorithm" [Qureshi and Patt 2006] is used where, in each iteration, a way is assigned to a core that shows the largest increase in locality per unit capacity. Using this approach, their technique finds quota allocation and corresponding block size. Compared to CP-only techniques, their technique provides higher performance by virtue of performing CP and block-size selection synergistically.

8.6. Integration with Cache Replacement Policy Selection

Zhan et al. [2014] note that CPTs and thrash-resistant RPs (e.g., BIP Qureshi et al. [2007], thread-aware DIP [Jaleel et al. 2008]) have complementary properties. RPs *temporally* share LLC based on applications' "locality," whereas CPTs *spatially* divide LLC based on applications' "utility." Hence, RPs that aim to avoid thrashing work well for workloads with poor-locality applications and CPTs work well for workloads with applications having significantly different utility values. Zhan et al. [2014] propose performing CPT in conjunction with selection of RP to optimize both locality and utility. They use two monitoring units to find hits for different numbers of ways: one sampled ATD with LRU and a second unit for BIP. Since BIP does not obey stack property,

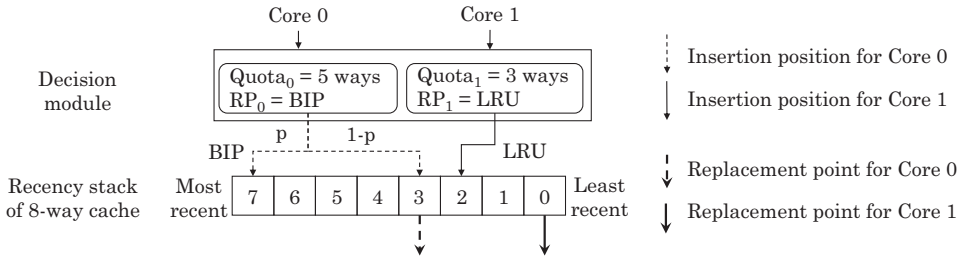

Fig. 15. Determining optimal RP and CP [Zhan et al. 2014].

its monitoring unit is designed differently based on its properties [Zhan et al. 2014]. This unit uses $\log_2(W_C)$ sampled ATDs with associativities of 2, 4, ..., $W_C/2$ and W_C, respectively, such that the ATD for associativity w shows hits at w-ways under BIP. BIP hit rate at other way-counts is deduced by linearly interpolating from the above values. Based on utility and locality information obtained from these monitoring units, their technique determines optimal CP and optimal RP, as illustrated in Figure 15. For CP, they use the "lookahead algorithm" [Qureshi and Patt 2006], except that MMU is computed from a hit-curve generated from higher portions of LRU and BIP hit curves. The RP chosen for a core is implemented in its cache portion and as for promotion policy, a block seeing hit is promoted by a single position [Xie and Loh 2009]. By virtue of leveraging largest utility provided by the best RP, their technique provides higher throughput and fairness compared to other RPs and CPTs.

8.7. Interaction with Cache Prefetching

Hasenplaugh et al. [2012] use gradient descent algorithm to partition the cache between various threads and prefetchers. Their technique computes a threshold for each thread, which decides the fraction of each thread's memory accesses that are inserted with higher-than-normal replacement probability. The prefetcher is also modeled as a thread for this purpose. Each thread is assigned a group of cache sets, called "gradient region." In half of these sets, their technique experiments with giving higher and lower (respectively) than current allocation of cache quota to the thread. If one half performs significantly better than the other, then the threshold of the thread is adapted, which subsequently changes the cache quota of the thread. The evaluation metric for determining the better half can be miss rate, IPC (instantaneous throughput of the thread) or weighted IPC for versatile optimization and by choosing the weights in weighted IPC metric, QoS targets can be achieved. Using this hill-climbing approach, their technique seeks to achieve globally optimal partition by making local decisions. Their CPT can also adapt the aggressiveness of the prefetcher based on the usefulness of blocks brought by it. Their technique improves throughput while achieving QoS targets.

8.8. Integration with Cache Locking

Suhendra and Mitra [2008] evaluate interaction of set-based CP with cache locking in shared L2 cache of a multicore system and study their impact on application WCET (worst case execution time). They compare following combinations of locking/partitioning: (a) "static/no-partition" (b) "static/core-level," and (c) "dynamic/core-level (refer to Figure 16)." In core-level CP, each task executing on that core can occupy the entire quota allocated to that core. They find that static/core-level scheme outperforms static/no-partition scheme since only the active tasks occupy the cache quota in the former scheme, whereas even idle tasks use the cache space in static/no-partition scheme. However, on every preemption, static/core-level scheme requires cache

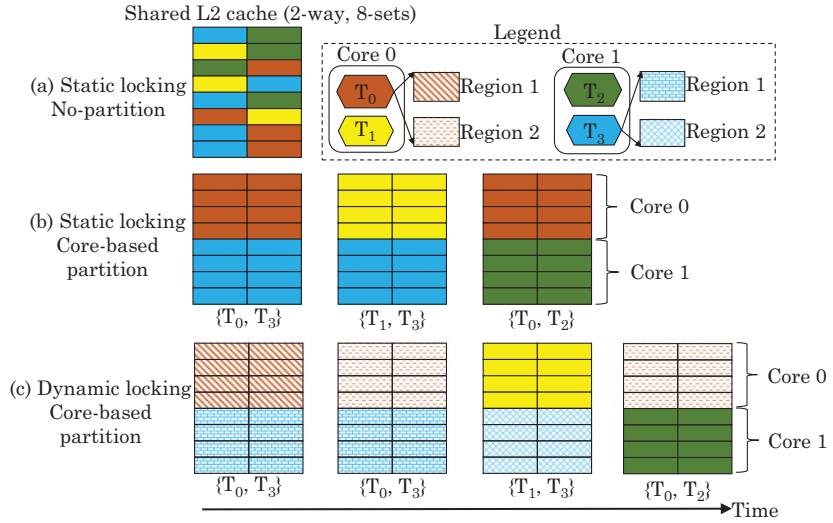

Fig. 16. Various CP and locking approaches and their integration [Suhendra and Mitra 2008]. T_0 to T_3 are four tasks. T_0 and T_3 are divided into two regions each for dynamic locking.

partition reloading and locking, which is its disadvantage. As for comparison between dynamic/core-level and static/core-level schemes, the former is advantageous for tasks with many regions, whereas the latter is easier to implement. Their study provides insights for developing management techniques for caches in real-time systems.

9. FUTURE DIRECTIONS AND CONCLUSION

To respond to increasingly diverse workloads, usage patterns, and optimization objectives, system-designers are moving toward heterogeneous processors, such as CPU-GPU heterogeneous systems [Mittal and Vetter 2015]. Due to fundamental differences between the microarchitectures of and workloads running on the CPU and the GPU, partitioning the shared cache between them presents different challenges and opportunities than those seen in multicore CPUs. Clearly, extending existing CPTs to and designing novel CPTs for CPU-GPU processors will be an interesting next-step for researchers especially as these processors gradually become the mainstream computing systems.

Modern processors employ multiple cache management approaches, for example, power-gating, data compression, cache prefetching, cache bypassing, and so on. Adoption of CP in modern processors will depend on its synergistic integration with these approaches. Going forward, a detailed study of interaction of CP with existing cache management approaches is definitely required.

Some researchers observe that insights obtained from simulators and real-systems may be different [Cook et al. 2013; Lin et al. 2008]. The differences in their execution windows/lengths, interaction with other factors, and so on may lead to vastly different conclusions, for example, about the cache-sensitivity of an application. Given this, adopting more rigorous standards for evaluation will promote reproducible research leading to universally applicable conclusions.

This article presented a survey on cache partitioning techniques in multicore systems. To emphasize the common and distinct features of different techniques, we classified them in many categories and reviewed their main ideas. It is hoped that more than merely synthesizing the state-of-art in cache partitioning research, the contribution of this article will be to inspire more efforts in this field toward enhancing

the effectiveness of partitioning in managing caches of next-generation computing systems.

REFERENCES

Manu Awasthi, Kshitij Sudan, Rajeev Balasubramonian, and John Carter. 2009. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'09)*. 250–261.

Nathan Beckmann and Daniel Sanchez. 2016. Modeling cache performance beyond LRU. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'16)*.

Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. 2008. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the International Symposium on Microarchitecture*. 318–329.

Jacob Brock, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. 2015. Optimal cache partition-sharing. In *Proceedings of the International Conference on Parallel Processing (ICPP'15)*. 749–758.

J. Chang and G. S. Sohi. 2007. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the International Conference on Supercomputing*. 242–252.

Jichuan Chang and Gurindar S. Sohi. 2006. Cooperative caching for chip multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA'06)*. 264–276.

Derek Chiou, Prabhat Jain, Larry Rudolph, and Srinivas Devadas. 2000. Application-specific memory management for embedded systems using software-controlled caches. In *Proceedings of the Design Automation Conference*. 416–419.

Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. 2010. Cache hierarchy and memory subsystem of the AMD opteron processor. *IEEE Micro.* 30, 2 (2010), 16–29.

Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. 2013. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *Proceedings of the International Symposium on Computer Architecture (ISCA'13).* 308–319.

Zehan Cui, Licheng Chen, Yungang Bao, and Mingyu Chen. 2014. A swap-based cache set index scheme to leverage both superpage and page coloring optimizations. In *Proceedings of the Design Automation Conference*. 1–6.

Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. 2012. Improving cache management policies using dynamic reuse distances. In *Proceedings of the International Symposium on Microarchitecture*. 389–400.

Haakon Dybdahl and Per Stenstrom. 2007. An adaptive shared/private NUCA cache partitioning scheme for chip multiprocessors. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'07)*. 2–12.

Saurabh Gupta and Huiyang Zhou. 2015. Spatial locality-aware cache partitioning for effective cache sharing. In *Proceedings of the International Conference on Parallel Processing (ICPP'15)*. 150–159.

Prateek D. Halwe, Shirshendu Das, and Hemangee K. Kapoor. 2013. Towards a better cache utilization using controlled cache partitioning. In *Proceedings of 2013 IEEE 11th International Conference on Dependable, Autonomic and Secure Computing (DASC'13).* 179–186.

William Hasenplaugh, Pritpal S. Ahuja, Aamer Jaleel, Simon Steely Jr., and Joel Emer. 2012. The gradient-based cache partitioning algorithm. *ACM Trans. Architect. Code Optim. (TACO)* 8, 4 (2012), 44.

Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. 2016. Cache QoS: From concept to reality in the Intel Xeon processor E5-2600 v3 product family. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'16)*. 657–668.

Enric Herrero, José González, and Ramon Canal. 2010. Elastic cooperative caching: An autonomous dynamically adaptive memory hierarchy for chip multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA'10).* 419–428.

Lisa R. Hsu, Steven K. Reinhardt, Ravishankar Iyer, and Srihari Makineni. 2006. Communist, utilitarian, and capitalist cache policies on CMPs: Caches as a shared resource. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'06)*. 13–22.

Ravi Iyer. 2004. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *Proceedings of the International Conference on Supercomputing*. 257–266.

Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. 2007. QoS policies and architecture for cache/memory in CMP platforms. *ACM SIGMETRICS Perform. Eval. Rev.* 35, 1 (2007), 25–36.

Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely Jr., and Joel Emer. 2008. Adaptive insertion policies for managing shared caches. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. 208–219.

Xinxin Jin, Haogang Chen, Xiaolin Wang, Zhenlin Wang, Xiang Wen, Yingwei Luo, and Xiaoming Li. 2009. A simple cache partitioning approach in a virtualized environment. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA'09)*. 519–524.

Jongpil Jung, Seonpil Kim, and Chong-Min Kyung. 2010. Latency-aware utility-based NUCA cache partitioning in 3D-stacked multi-processor systems. In *Proceedings of the VLSI System on Chip Conference (VLSI-SoC'10)*. 125–130.

Mahmut Kandemir, Ramya Prabhakar, Mustafa Karakoy, and Yuanrui Zhang. 2011a. Multilayer cache partitioning for multiprogram workloads. In *Proceedings of the European Conference on Parallel Processing*. 130–141.

Mahmut Kandemir, Taylan Yemliha, and Emre Kultursay. 2011b. A helper thread based dynamic cache partitioning scheme for multithreaded applications. In *Proceedings of the Design Automation Conference*. 954–959.

Dimitris Kaseridis, Muhammad Faisal Iqbal, and Lizy Kurian John. 2014. Cache friendliness-aware managementof shared last-level caches for high performance multi-core systems. *IEEE Trans. Comput.* 63, 4 (2014), 874–887.

Dimitris Kaseridis, J. Stuecheli, Jian Chen, and Lizy K. John. 2010. A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large CMP systems. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'10)*. 1–11.

Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy K. John. 2009. Bank-aware dynamic cache partitioning for multicore architectures. In *Proceedings of the International Conference on Parallel Processing (ICPP'09)*. 18–25.

Harshad Kasture and Daniel Sanchez. 2014. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. 729–742.

Kamil Kedzierski, Miquel Moreto, Francisco J. Cazorla, and Mateo Valero. 2010. Adapting cache partitioning algorithms to pseudo-LRU replacement policies. In *Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS'10)*. 1–12.

Samira Khan, Alaa R. Alameldeen, Chris Wilkerson, Onur Mutlu, and Daniel A. Jiménez. 2014. Improving cache performance by exploiting read-write disparity. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'14)*.

Seongbeom Kim, Dhruba Chandra, and Yan Solihin. 2004. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*. 111–122.

I. Kotera, K. Abe, R. Egawa, H. Takizawa, and H. Kobayashi. 2011. Power-aware dynamic cache partitioning for CMPs. *Trans. HiPEAC* (2011), 135–153.

Vivek Kozhikkottu, Abhisek Pan, Vijay Pai, Sujit Dey, and Anand Raghunathan. 2014. Variation aware cache partitioning for multithreaded programs. In *Proceedings of the Design Automation Conference*. 1–6.

Hyunjin Lee, Sangyeun Cho, and Bruce R. Childers. 2011. CloudCache: Expanding and shrinking private caches. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'11)*. 219–230.

Jaekyu Lee and Hyesoon Kim. 2012. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'12)*. 1–12.

J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'08)*. 367–378.

J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. 2009. Enabling software multicore cache management with lightweight hardware support. In *Proceedings of the Conference on Supercomputing (SC)*.

Xing Lin and Rajeev Balasubramonian. 2011. Refining the utility metric for utility-based cache partitioning. In *9th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD), in conjunction with the 38th International Symposium on Computer Architecture (ISCA-38)* (2011).

Fang Liu, Xiaowei Jiang, and Yan Solihin. 2010. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'10)*. 1–12.

Lei Liu, Yong Li, Zehan Cui, Yungang Bao, Mingyu Chen, and Chengyong Wu. 2014. Going vertical in memory management: Handling multiplicity by multi-policy. In *Proceedings of the International Symposium on Computer Architecture (ISCA'14)*. 169–180.

David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2016. Improving resource efficiency at scale with heracles. *ACM Trans. Comput. Syst. (TOCS)* 34, 2 (2016), 6.

Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Systems Journal* 9, 2 (1970), 78–117.

Intel Corporation. 2016. Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3B, System Programming Guide, Part 2. Retrieved from http://goo.gl/sw24WL.

OSU-CSE News. 2010. Intel Puts OSU-CSE Inside. Retrieved from http://web.cse.ohio-state.edu/news/news118.shtml.

Vineeth Mekkat, Anup Holey, Pen-Chung Yew, and Antonia Zhai. 2013. Managing shared last-level cache in a heterogeneous multicore processor. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*. 225–234.

Sparsh Mittal. 2016a. A survey of architectural techniques for managing process variation. *Comput. Surveys* 48, 4 (2016), 54:1–54:29.

Sparsh Mittal. 2016b. A survey of cache bypassing techniques. *J. Low Power Elect. Appl.* 6, 2 (2016), 5:1–5:30.

Sparsh Mittal, Yanan Cao, and Zhao Zhang. 2014a. MASTER: A multicore cache energy saving technique using dynamic cache reconfiguration. *IEEE Trans. VLSI Syst.* 22, 8 (2014), 1653–1665.

Sparsh Mittal, Matthew Poremba, Jeffrey Vetter, and Yuan Xie. 2014b. *Exploring Design Space of 3D NVM and eDRAM Caches Using DESTINY Tool*. Technical Report ORNL/TM-2014/636. Oak Ridge National Laboratory, USA.

Sparsh Mittal and Jeffrey Vetter. 2015. A survey of CPU-GPU heterogeneous computing techniques. *Comput. Surveys* 47, 4 (2015), 69:1–69:35.

Sparsh Mittal and Jeffrey Vetter. 2016. A survey of techniques for architecting DRAM caches. *IEEE Trans. Parallel. Distrib. Syst. (TPDS)* 27, 6 (2016), 1852–1863.

Sparsh Mittal, Jeffrey S. Vetter, and Dong Li. 2015. A survey of architectural approaches for managing embedded DRAM and non-volatile on-chip caches. *IEEE Trans. Parallel Distrib. Syst. (TPDS)* 26, 6 (2015), 1524–1537.

Sparsh Mittal and Zhao Zhang. 2013. *MANAGER: A Multicore Shared Cache Energy Saving Technique for QoS Systems*. Technical Report. Iowa State University.

Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, Rizos Sakellariou, and Mateo Valero. 2009. FlexDCP: A QoS framework for CMP architectures. *ACM SIGOPS Operat. Syst. Rev.* 43, 2 (2009), 86–96.

Miquel Moreto, Francisco J Cazorla, Alex Ramirez, and Mateo Valero. 2008. MLP-aware dynamic cache partitioning. In *High Performance Embedded Architectures and Compilers*. 337–352.

Sai Prashanth Muralidhara, Mahmut Kandemir, and Padma Raghavan. 2010. Intra-application cache partitioning. In *Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS'10)*. 1–12.

Konstantinos Nikas, Matthew Horsnell, and Jim Garside. 2008. An adaptive Bloom filter cache partitioning scheme for multicore architectures. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'08)*. 25–32.

Taecheol Oh, Kiyeon Lee, and Sangyeun Cho. 2011. An analytical performance model for co-management of last-level cache and bandwidth sharing. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'11)*. 150–158.

Abhisek Pan and Vijay S. Pai. 2013. Imbalanced cache partitioning for balanced data-parallel programs. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*. 297–309.

Pavlos Petoumenos, Georgios Keramidas, Håkan Zeffer, Stefanos Kaxiras, and Erik Hagersten. 2006. StatShare: A statistical model for managing cache sharing via decay. In *Proceedings of the Workshop on Modeling, Benchmarking and Simulation (MoBS'06)*.

Miquel Moreto Planas, Francisco Cazorla, Alex Ramirez, and Mateo Valero. 2007. Explaining dynamic cache partitioning speed ups. *IEEE Comput. Arch. Lett.* 6, 1 (2007), 1–4.

Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. In *Proceedings of the International Symposium on Computer Architecture* (2007), 381–391.

Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. 423–432.

Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. 2006. Architectural support for operating system-driven CMP cache management. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 2–12.

R. Reddy and P. Petrov. 2010. Cache partitioning for energy-efficient and interference-free embedded multitasking. *ACM Trans. Embed. Comput. Syst. (TECS)* 9, 3 (2010), 16.

Daniel Sanchez and Christos Kozyrakis. 2010. The ZCache: Decoupling ways and associativity. In *Proceedings of the International Symposium on Microarchitecture*. 187–198.

D. Sanchez and C. Kozyrakis. 2011. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the International Symposium on Computer Architecture*. 57–68.

Alex Settle, Dan Connors, Enric Gibert, and Antonio González. 2006. A dynamically reconfigurable cache for multithreaded processors. *J. Embed. Comput.* 2, 2 (2006), 221–233.

Shekhar Srikantaiah, Reetuparna Das, Asit K. Mishra, Chita R. Das, and Mahmut Kandemir. 2009a. A case for integrated processor-cache partitioning in chip multiprocessors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09)*. 6.

Shekhar Srikantaiah, Mahmut Kandemir, and Qian Wang. 2009b. SHARP control: Controlled shared cache management in chip multiprocessors. In *Proceedings of the 42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42'09)*. 517–528.

H. S. Stone, J. Turek, and J. L. Wolf. 1992. Optimal partitioning of cache memory. *IEEE Trans. Comput.* 41, 9 (1992), 1054–1068.

Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. 2015. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. 62–75.

G. Edward Suh, Srinivas Devadas, and Larry Rudolph. 2001. Analytical cache models with applications to cache partitioning. In *Proceedings of the International Conference on Supercomputing*. 1–12.

G. E. Suh, L. Rudolph, and S. Devadas. 2004. Dynamic partitioning of shared cache memory. *J. Supercomput.* 28, 1 (2004), 7–26.

Vivy Suhendra and Tulika Mitra. 2008. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proceedings of the Design Automation Conference*. 300–303.

Karthik T. Sundararajan, Vasileios Porpodas, Timothy M. Jones, Nigel P. Topham, and Bjorn Franke. 2012. Cooperative partitioning: Energy-efficient cache partitioning for high-performance CMPs. In *Proceedings of the International Symposium on High-Performance Computer Architecture,* (2012), 1–12.

David Tam, Reza Azimi, Livio Soares, and Michael Stumm. 2007. Managing shared L2 caches on multicore systems in software. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*. 26–33.

Keshavan Varadarajan, S. K. Nandy, Vishal Sharda, Amrutur Bharadwaj, Ravi Iyer, Srihari Makineni, and Donald Newell. 2006. Molecular caches: A caching structure for dynamic creation of application-specific heterogeneous cache regions. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 433–442.

Ruisheng Wang and Lizhong Chen. 2014. Futility scaling: High-associativity cache partitioning. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. 356–367.

Xiaorui Wang, Kai Ma, and Yefu Wang. 2012. Cache latency control for application fairness or differentiation in power-constrained chip multiprocessors. *IEEE Trans. Comput.* 61, 10 (2012), 1371–1385.

Xiaodong Wang and José F. Martínez. 2015. XChange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'15)*. 113–125.

Y. Xie and G. H. Loh. 2009. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 174–183.

Yuejian Xie and Gabriel H. Loh. 2010. Scalable shared-cache management by containing thrashing workloads. In *Proceedings of the International Conference on High-Performance Embedded Architectures and Compilers*. 262–276.

Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. 2014. Coloris: A dynamic cache partitioning system using page coloring. In *Proceedings of the International Conference on Parallel Architectures and Compilation*. 381–392.

Thomas Y. Yeh and Glenn Reinman. 2005. Fast and fair: Data-stream quality of service. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'05)*. 237–248.

Chenjie Yu and Peter Petrov. 2010. Off-chip memory bandwidth minimization through cache partitioning for multi-core platforms. In *Proceedings of the Design Automation Conference*. 132–137.

Heechul Yun and Prathap Kumar Valsan. 2015. Evaluating the isolation effect of cache partitioning on COTS multicore platforms. In *Proceedings of the 11th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'15)*. 45.

Dongyuan Zhan, Hong Jiang, and Sharad C. Seth. 2014. CLU: Co-optimizing locality and utility in thread-aware capacity management for shared last level caches. *IEEE Trans. Comput.* 63, 7 (2014), 1656–1667.

Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. 2009. Towards practical page coloring-based multicore cache management. In *Proceedings of the European Conference on Computer Systems*. 89–102.

Miao Zhou, Yu Du, Bruce Childers, Rami Melhem, and Daniel Mossé. 2012. Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems. *ACM Trans. Arch. Code Optim. (TACO)* 8, 4 (2012), 53.

Miao Zhou, Yu Du, Bruce Childers, Daniel Mosse, and Rami Melhem. 2016. Symmetry-agnostic coordinated management of the memory hierarchy in multicore systems. *ACM Trans. Arch. Code Optim. (TACO)* 12, 4 (2016), 61.

Received July 2016; revised January 2017; accepted March 2017