Des Autom Embed Syst
DOI 10.1007/s10617-015-9168-7

CaPPS: cache partitioning with partial sharing for multi-core embedded systems

Wei Zang[1] · **Ann Gordon-Ross**[2]

Received: 19 May 2014 / Accepted: 15 October 2015

Abstract As the number of cores in chip multi-processor systems increases, the contention over shared last-level cache (LLC) resources increases, thus making LLC optimization critical, especially for embedded systems with strict area/energy/power constraints. We propose cache partitioning with partial sharing (CaPPS), which reduces LLC contention using cache partitioning and improves utilization with sharing configuration. Sharing configuration enables the partitions to be privately allocated to a single core, partially shared with a subset of cores, or fully shared with all cores based on the co-executing applications' requirements. CaPPS imposes low hardware overhead and affords an extensive design space to increase optimization potential. To facilitate fast design space exploration, we develop an analytical model to quickly estimate the miss rates of all CaPPS configurations using the applications' isolated LLC access traces to predict runtime LLC contention. Experimental results demonstrate that the analytical model estimates cache miss rates with an average error of only 0.73 % and with an average speedup of 3505× as compared to a cycle-accurate simulator. Due to CaPPS's extensive design space, CaPPS can reduce the average LLC miss rate by as much as 25 % as compared to baseline configurations and as much as 14–17 % as compared to prior works.

Keywords Cache memories · Modeling techniques · Optimization · Performance evaluation

✉ Wei Zang
zangweiufl@gmail.com

Ann Gordon-Ross
ann@ece.ufl.edu

1 SK Hynix Memory Solution, San Jose, CA, USA

2 Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA

1 Introduction

In chip multi-processor systems (CMPs), shared resources are optimized to manage access contention from multiple cores. Shared last-level caches (LLCs) (e.g., second-/third-level) are widely used in prevailing CMPs, such as the ARM Cortex-A, Intel Atom, and Sun T2 [1,15, 16], to improve cache utilization. LLCs should be large enough to accommodate all sharing cores' data, however, due to long access latencies and high power consumption, large LLCs are typically precluded from embedded systems with strict area/energy/power constraints. Therefore, optimizing small LLCs is significantly more challenging due to contention for limited cache space.

Shared LLCs afford high cache utilization and no coherence overhead, however, high contention and unfair cache utilization can significantly degrade performance. A core's LLC occupancy (utilized space) is flexible and dictated by the core's applications' demands. Cores with high LLC requirements can occupy a large LLC area and cause high, potentially unfair, contention. For example, streaming multimedia applications occupy the LLC with a large amount of single-accessed data and may unfairly evict the other cores' data, thus increasing LLC miss rates.

To eliminate shared LLC contention, cache partitioning [8,22,28] partitions the cache, allocates *quotas* (a subset of partitions) to the cores, and optionally configures the partitions/quotas (e.g., size and/or associativity) to the allocated core's requirements. Each core's cache occupancy is constrained to the core's quota to ensure fair utilization. *Set partitioning* partitions and allocates quotas at the cache set granularity and is typically implemented using operating system (OS)-based page coloring [17]. However, due to this OS modification requirement, hardware-based way partitioning is more widely used. *Way partitioning* partitions and allocates quotas at the cache way granularity [22,28] and is implemented using column caching or a modified replacement policy [8,9,18].

Way partitioning for shared LLCs typically uses *private partitioning*, which restricts quotas for exclusive use by the allocated core and can lead to poor cache utilization if a core does not occupy the core's entire allocated quota. Thus, partially sharing a core's quota with other cores can potentially improve cache utilization. In this work, we propose a hybrid LLC organization that combines the benefits of private and shared partitioning—cache partitioning with partial sharing (CaPPS).

CaPPS controls each core's cache utilization using *sharing configuration*, which enables a core's quota to be configured as private, partially shared with a subset of cores, or fully shared with all other cores. Although partial sharing exists in some previous private LLC partitioning works [9,14,19,27], these sharing configurations are not as flexible or as extensive as CaPPS and prior works' design spaces are subsets of CaPPS's design space. Therefore, we refer to these prior works' sharing configurations as *constrained partial sharing*. Although CaPPS partitions the cache with coarse-grained way granularity, partial sharing greatly extends the design space. The coarse-grained partitioning, however, enables lightweight hardware implementation, which will be discussed in Sect. 3.1. Whereas CaPPS's sharing configuration increases the design space and thus increases optimization potential, this large design space significantly increases design space exploration time.

Since using a CMP simulator to exhaustively simulate CaPPS's entire design space is prohibitively lengthy for realistic applications (several months or more), to facilitate fast design space exploration, we developed an offline analytical model to quickly estimate cache miss rates for all configurations, which enables determining optimal LLC configurations for any optimization that evaluates cache miss rates (e.g., performance, energy, energy delay product, power, etc.). During design space exploration, the offline analysis imposes no runtime overhead, in term of performance, area, and power/energy, thus, CaPPS is especially suitable for embedded systems with predictable applications and stable behaviors. For unpredictable systems, the analytical model could be incorporated into the OS scheduling routine and use online/offline phase change detection [25]. Since embedded applications only have a few phases, analyzing all combinations of each application's phases is still feasible. However, these details are beyond the scope of this work.

The analytical model probabilistically predicts the shared cache miss rates using co-executing applications' *isolated cache access distributions* (i.e., the application is run in isolation with no co-executing applications). This probabilistic prediction provides a fair and realistic offline method for evaluating any combination of co-executed applications, which cannot be determined at design time for dynamically scheduled systems. Although several previous works [4,7,10] have developed analytical models to predict shared LLC contention offline, these works' caches were fully shared by all cores, which precludes these works from being used in CaPPS.

Experiments reveal that the analytical model estimates cache miss rates for CaPPS's entire design space with an average error of only 0.73 % and is 3505× faster, on average, than a cycle-accurate simulator. Due to CaPPS's extensive design space, CaPPS can reduce the average LLC miss rate by as much as 25 % as compared to baseline configurations, and by as much as 14–17 % as compared to prior works' partitioning methods. Finally, CaPPS's hardware implementation has low energy and area overheads, and does not increase the cache access time. For easy reference, Table 1 defines notations used throughout this paper.

2 Related work

Since CaPPS uses physical way partitioning and our analytical model predicts the contention in the shared ways, we compare our work with prior works in these areas.

Previous shared cache partitioning typically used private partitioning. Qureshi and Patt [22] developed dynamic utility-based cache partitioning (UCP), which used an online monitor to track the cache misses for all possible numbers of ways assigned to each core. Greedy and refined heuristics determined the cores' quotas. Varadarajan et al. [30] dynamically partitioned the cache into small direct-mapped cache partitions, which were privately assigned to the cores and the cache partitions had configurable size, block size, and associativity. Kim et al. [18] developed cache partitioning for fairness optimization using static and dynamic partitioning. Static cache partitioning used the cache access's stack distance profile to determine the cores' requirements. Dynamic cache partitioning monitored the cache misses, and increased/decreased the cores' quotas in accordance with the miss rate changes between evaluation intervals. Suh et al. [28] partitioned the cache and developed a greedy dynamic partitioning method to allocate each partition to the cores for exclusive use. Although their method allowed partitions to be shared across multiple cores when the number of cores exceeded the number of partitions, the equation used to estimate the number of misses in the shared partitions did not consider the contention effects when a core's quota was shared with other cores. Sundararajan et al. [29] also partitioned the cache into private and shared regions. The partitioning was based on the ownership of the data instead of each core's requirements. Manikantan et al. [20] partitioned the shared cache at the block-level granularity, and controlled each cores' occupancy using the eviction probabilities. Futility Scaling [31] is another fine-grained partitioning method that precisely controlled the partitions' sizes by scaling the

Table 1 Notational reference

Notation	Definition
$\|\|$	Logical OR operator
$C(n, r)$	The number of combinations of r numbers selected from a set with n numbers
A	LLC associativity
N_C	Total number of CMP cores
C_i	The i-th core, where $i \in [1, N_C]$
m_{C_i}	Number of LLC misses for core C_i
h_{C_i}	Number of LLC hits for core C_i
K_{C_i}	Number of ways allocated to core C_i
K_S	Number of shared ways
K_{P,C_i}	Number of private ways allocated to core C_i
O_{C_i}	Number of blocks currently occupied by core C_i
n_{C_i}	C_i's number of accesses during the time period (t_1, t_2) (as shown in Fig 6)
R_{C_i}	The number of blocks evicted into the shared ways during the co-executed core C_i's n_{C_i} accesses
$p\left(n_{C_i}, R_{C_i}\right)$	The probability that R_{C_i} number of blocks are evicted in the n_{C_i} accesses
d	Stack distance, where $d \in [0, A]$
N_d	Number of accesses with stack distance d
r	Reuse distance
$Cycles_{exe}$	Number of CPU cycles required when the application is executed in isolation on a CMP
$Cycles_{base}$	Base CPU cycles, calculated from $Cycles_{exe}$ by assuming that all LLC accesses are hits
$LLC_{latency}$	CPU delay cycles incurred by an LLC miss
m_{exe}	Number of LLC misses in the application's isolated execution
$Cycles$	The number of CPU cycles required to execute the application when co-executing with other applications. Subscript C_i can be added to specify the core
$\hat{X}$	The estimate of X. X can be $Cycles$, m_{C_i}, or h_{C_i}
$\bar{X}$	The average of X. X can be r or n_i
$\|X\|$	Cardinality of X
$\vec{X}$	A vector

futility of the cache lines. These fine-grained partitioning methods provided more partitioning flexibility and scaled well to systems with a large number of cores and a limited number of cache ways.

Private LLCs also benefit from cache partitioning, where the cores' private caches are partitioned to be partially/fully shared with other cores. In CloudCache [19], the private caches were partitioned and a core could share the private caches' of nearby cores (to restrict the additional access latency). MorphCache [27] partitioned the level two and level three caches and allowed subsets of cores' private caches to be merged and fully shared by the subset. Huh et al. [14] subsetted the cores, partitioned the cache evenly, and each partition was fully shared by a subset. Dybdahl and Stenstrom [9] developed an adaptive cache partitioning method in which a core's private cache could be partially shared among all cores. An adaptive spill-receive caching method [21] classified private caches as spiller or receiver caches, where the spiller caches could store evicted blocks into receiver caches. Chang and Sohi [5] proposed cooperative caching that allowed evicted blocks from a core's private cache to be stored into the other cores' private caches if the other core's cache had spare capacity. In more recent works, the authors integrated multiple time-sharing partitions into cooperative caching [6] to improve throughput and fairness by time-sharing the cache among multiple unfair partitions that satisfies different applications' requirements.

Although private LLC partitioning enabled a core to share other cores' quotas, only two kinds of constrained partial sharing were provided (1) *subset sharing* the cores were subsetted and the cores' quotas were fully shared by the subset [14,19,27]; (2) *joint sharing* partial sharing allowed a portion/all of a core's quota to be shared by all cores [9]. As compared to partially sharing a core's quota with *all* cores, CaPPS is more flexible than these works by enabling a portion of a core's quota to be shared with *any* subset of cores, and thus the constrained partial sharings' design spaces are subsets of CaPPS's design space.

Prior works on analytical modeling to determine cache miss rates targeted only fully shared caches, therefore, our proposed analytical model can leverage the fundamentals established in these prior works. The first analytical model to calculate shared cache miss rates was proposed by Shedler and Slutz [23]. The authors modeled the shared cache accesses as a stochastic process and derived a closed form expression for the expected cache miss rate. Chandra et al. [4] proposed a model using access traces for isolated threads to predict inter-thread contention. Reuse distance profiles were analyzed to predict the extra cache misses for each thread due to cache sharing, but the model did not consider the interaction between cycles per instruction (CPI) variations and cache contention. Eklov et al. [10] proposed a simpler model that calculated the CPI considering the cache misses caused by contention by predicting the reuse distance distribution of an application when co-executed with other applications based on the applications' isolated reuse distance distribution. Chen and Aamodt [7] proposed a Markov model to estimate the cache miss rates for multi-threaded applications with inter-thread communication.

Since CaPPS allows a core's quota to be partially shared with a subset of cores, as compared to fully shared by all cores as in prior works, analytically predicting the cache miss rates is more challenging. In CaPPS, the time-ordered interleaving of the cores' accesses to the LLC must be considered, since only the LLC accesses that access partially shared ways affect a core's miss rate. However, since the analytical model executes offline, statically determining, quantifying, and predicting the dynamic effects significantly complicates miss rate estimations.

3 Cache partitioning with partial sharing (CaPPS)

Similarly to many commercial multi-core architectures [1,15,16], Fig. 1 depicts our system architecture, where each core has one or more levels of private caches, and all cores share the LLC. The last level private caches (LP cache) in all cores are connected to the shared LLC through a shared bus.

Our analytical model will predict the shared cache contention from each core's isolated cache accesses from the LP cache. Therefore, our model can be applied on the architectures with any number of private cache levels above the shared LLC.

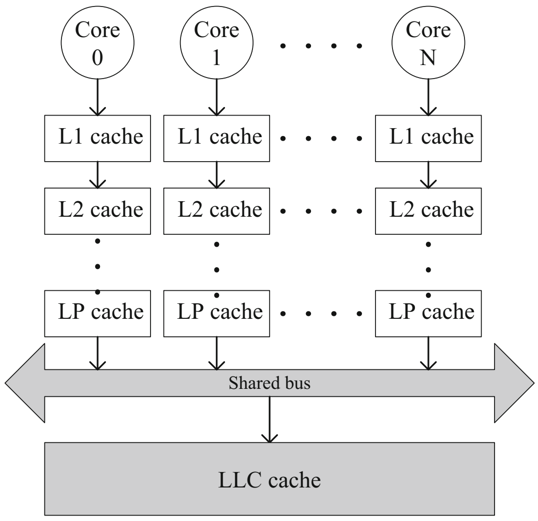

Fig. 1 System architecture

To accommodate LLC requirements for multiple applications co-executing in different cores on a CMP, CaPPS partitions the shared LLC at the way granularity and leverages sharing configuration to allocate the partitions to each core's quota. We discuss the architecture and sharing configurations in Sect. 3.1. To facilitate fast design space exploration, an analytical model (overviewed in Sect. 3.2) estimates the cache miss rates for the CaPPS configurations using the applications' isolated LLC access traces. Section 3.3 discusses the isolated access trace processing, and Sect. 3.4 presents the details of the analytical model.

We make several base assumptions regarding the targeted CMPs with respect to CaPPS's functionality. We assume that there is no shared instruction/data addresses between the cores, which is a common case in general purpose CMPs where each core executes a different application in an independent address space, and is similar to assumptions made in prior works [4,10]. In some cases, the system library functions are shared between applications, however, since these shared functions' code sizes are typically very small, especially in embedded systems, without loss of generality, we can omit the shared functions' effects and replicate the shared instruction addresses in LLC if necessary (dictated by the replacement policy as described in Sect. 3.1), with no special processing.

3.1 Architecture and sharing configurations

CaPPS's sharing configurations enable a core's quota to be configured as private, partially shared by a subset of cores, or fully shared by all cores. Figure 2a, b, c illustrates sample configurations, respectively, for a 4-core CMP (C_1 to C_4) and an 8-way LLC (a) each core's quota has a configurable number of private ways; (b) C_1's quota has three private ways and two shared ways with C_2, C_2's quota contains an additional private way, and C_3's quota has two ways, one is private and one way is shared with C_4; and (c) all of the four cores fully share all of the ways.

CaPPS uses the least recently used (LRU) replacement policy, but the analytical model can be extended to approximate cache miss rates for other replacement policies, such as pseudo-LRU, but is beyond the scope of this work. To reduce the sharing configurability with no effect on cache performance and to minimize contention, cores share an arbitrary number of ways starting with the LRU way, then second LRU way, and so on since these

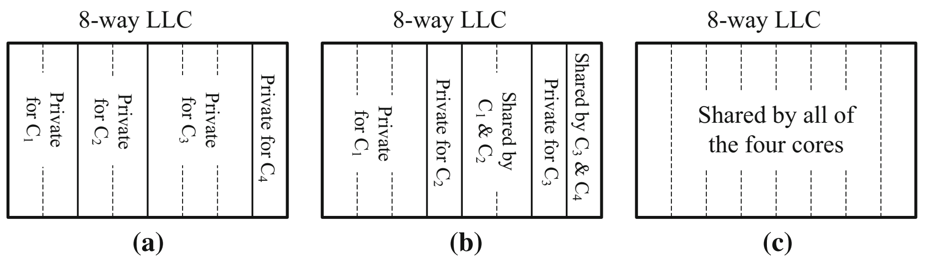

Fig. 2 Three sample sharing configurations **a** the cores' quotas are private; **b** some ways are partially shared with a subset of cores; and **c** the entire LLC is fully shared with all other cores

ways are least likely to be accessed. This pruning method intelligently removes the redundant sharing configurations that have higher contention potential. For example, in Fig. 2b two of C_1's ways are shared with C_2, therefore, C_1's two most recently used (MRU) blocks are cached in C_1's two private ways, and the two LRU blocks are cached in the two ways shared with C_2 and these two LRU blocks are the only replacement candidates for C_2's accesses.

Prior works primarily used two hardware support approaches for cache partitioning. One approach leveraged a modified LRU replacement policy [9,18,22] and selected replacement candidates based on the blocks' MRU orderings and the number of blocks occupied by each core. The other approach used column caching [8,28] to globally control which ways a core's data could be cached in (i.e., the ways that contained candidate replacement blocks for a particular core). Neither approach increased the cache access times since the new logic was only activated during a cache miss, and replacement block selection occurred in parallel with the miss fetch. The hardware implementation of the modified LRU replacement policy requires more hardware overhead than column caching. Maintaining the access order of the cache ways in each set and recording the per-block core ids consumes too many hardware resources. However, column caching can globally restrict the cache ways that each core can access, and the global control logic only requires lightweight hardware. In the following subsections, we detail how these approaches could be modified for CaPPS.

3.1.1 Modified LRU replacement policy

A conventional LRU cache associates a counter with each block to denote the blocks' MRU ordering in the cache set. To adapt this basic hardware for CaPPS, per-block core identification (ID) is required. For example, assume two cores, C_1 and C_2, share K_S number of ways in a CaPPS configuration, and each core has K_{P,C_1} and K_{P,C_2} number of private ways, respectively. On a cache hit, the blocks' counters are updated to indicate the new MRU ordering similarly to a conventional LRU cache. On a cache miss, C_1 and C_2's occupied number of blocks, O_{C_1} and O_{C_2}, in the set must be determined. If the C_1 caused the miss and there are unused/invalid blocks in C_1's private ways or in C_1 and C_2's shared ways, which can be dictated by validating $\left(O_{C_1} < K_{P,C_1}\right) \,||\, \left(\left(O_1 + \max\left(O_{C_2}, K_{P,C_2}\right)\right) < \left(K_{P,C_1} + K_{P,C_2} + K_S\right)\right)$, the new fetched data can be cached into an unused/invalid block, otherwise a replacement block must be selected. In C_1 and C_2's occupied blocks, after excluding the private ways' number of MRU blocks (i.e., K_{P,C_1} and K_{P,C_2}, respectively), the replacement block is the LRU block in the remaining blocks (i.e., the LRU block in the shared K_S number of ways). A similar method determines the replacement block when more than two cores share cache ways.

The additional hardware required to use this approach for CaPPS is the per-block core ID, which is $\log_2(N_c)$ bits, where N_C is the total number of cores. Additionally, when changing sharing configurations, all cache blocks must be invalided and dirty blocks written back in the case of a write back cache, which can induce additional cache misses, however, this overhead is required for any reconfigurable cache and is not an additional overhead for CaPPS as compared to prior work.

3.1.2 Column caching

Column caching uses a per-core *partition vector* to globally control the cores' candidate replacement ways for all cache sets. The partition vector is a bit vector where the number of bits is equal to the cache associativity and a set bit '1' denotes that the bit's associated way is assigned to that core. For example, if the cache associativity is eight and a core's partition vector is "00111001", four ways are allocated to the core: the third, fourth, fifth, and eighth ways. Cache fetches are the same as in a conventional cache (i.e., *all* tags in the cache set are compared with the fetched block's tag), thus the partition vector does not increase the cache access time. On a cache miss, the replacement block is selected from the core's allocated ways as denoted by the core's associated partition vector.

Column caching introduces minimal hardware overhead since only per-core partition vector are required and the vectors are globally used by all sets. Changing sharing configurations requires new partition vector contents to be loaded, but unlike the modified LRU replacement policy, cache block invalidation and dirty block write backs are not required since all of the tags in the ways are compared with the fetched block. On a cache miss, the replacement block is selected using the new partition vector, thus column caching does not induce additional cache misses as compared to the modified LRU replacement policy.

However, the conventional LRU cache implementation that uses counters to denote the MRU orderings cannot be used in column caching. Column caching uses the partition vectors to globally control which physical ways a core's data is cached in for all cache sets. Since CaPPS's sharing configurations restrict sharing from the LRU ways, a simple counter-implemented LRU cache cannot be used since the LRU blocks can be stored in any physical way (dictated by the associated counter's value) in different cache sets and at different times. Thus, there is no physical way (which stores the LRU blocks of a core in all cache sets) that can be shared with other cores, and thereby globally controlled by a partition vector bit.

Instead of counters, linked lists can be used to denote the MRU orderings, and prior works [11] showed that linked lists used less hardware resources and afforded faster cache access time as compared to counters for a conventional LRU cache implementation. In the linked list implementation, a cache set's blocks are indexed and the indexes of the blocks are separately maintained in a linked list. Figure 3a depicts a sample linked list for an 8-way cache, where the linked list registers D1 through D8 store the blocks' indexes. When a replacement is required, the index stored in the LRU register (i.e., D8) identifies the index of the way's replacement block.

Instead of associating the partition vector bits with each physical cache way [8], CaPPS can associate the partition vector bits with the linked list registers (i.e., the most and least significant bits are associated with the head and tail registers of the linked list, respectively). Thus, in a core's partition vector, partial sharing enables other cores to share ways starting from the right-most set bit's associated linked list register, which always stores the core's LRU block's index. Considering the example in Fig. 2b, in an 8-way cache where two cores, C_1 and C_2, share two ways, C_1 has three private ways, and C_2 has one private way. C_1 and C_2's partition vectors are "11101100" and "00011100", respectively. The other two cores C_3 and C_4 share one way and C_3 has one private way. C_3 and C_4's partition vectors are "00000011" and "00000001", respectively. Note that if the third core, C_3, also shares two

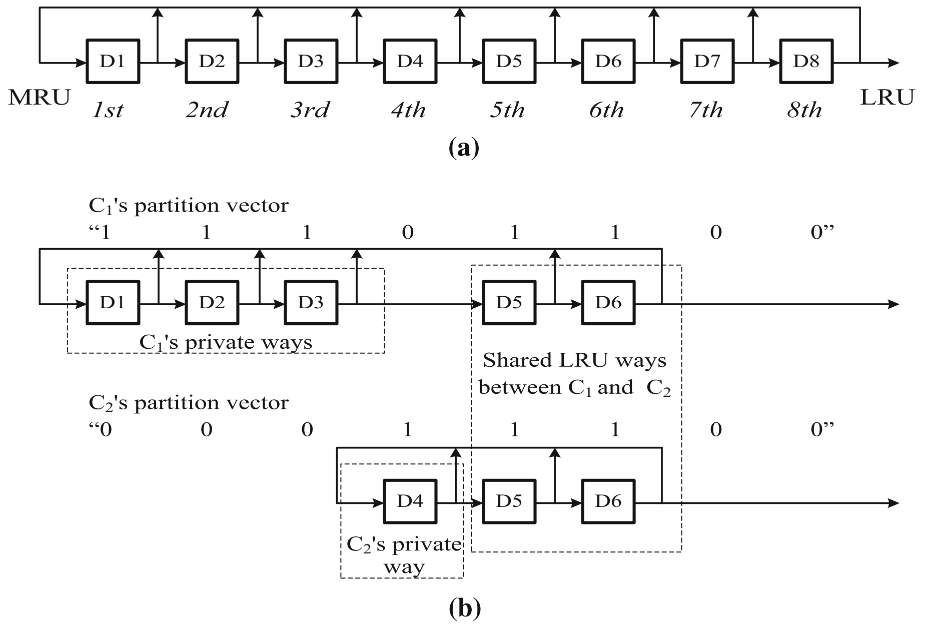

Fig. 3 Maintaining LRU information using a linked list for **a** a conventional LRU cache and **b** a CaPPS' sharing configuration where C1 and C2's partition vectors are "11101100" and "00011100", respectively

ways with C_1 and C_2, and C_3 has two private ways, the set bits of the two shared ways will be shifted right by two positions, correspondingly. The partition vectors for C_1, C_2, and C_3 are "11100011", "00010011", and "00001111", respectively. Figure 3b depicts the associated linked lists for C_1 and C_2 with partition vectors "11101100" and "00011100", respectively. Since each bit corresponds to a linked list register, the blocks allocated to C_1 are the indexes stored in D1, D2, D3, D5, and D6 and the blocks allocated to C_2 are the indexes stored in D4, D5, and D6, where the shared D5 and D6 by the two cores indicates the LRU and second LRU blocks for C_1 and C_2.

The additional hardware overhead when comparing Fig. 3a with Fig. 3b is the data path from D3 to D5, which can be implemented by adding a bypass transfer line with the linked list registers, as shown in Fig. 4. The transfer lines and the linked list registers' outputs are connected using n-MOS switches. Therefore, the bypass source and destination registers can be connected by turning on the registers' n-MOS switches. By carefully controlling the "on/off" status of the switches and the "load enable" of linked list registers based on the partition vector's bits' values, the linked lists, as in Fig. 3b, can be easily maintained. For example, to maintain C_1's linked list in Fig. 3b, the "load enable" of the registers associated with the '1' bit values in C_1's partition vector (i.e., En_1, En_2, En_3, En_5, and En_6) are set as valid, and the switches Sw_3 and Sw_4 are turned on to bypass the D4 register. To maintain C_2's linked list in Fig. 3b, the "load enable" of the registers associated with the '1' bit values in C_2's partition vector (i.e., En_4, En_5, and En_6) are set as valid. Since no bypass is required, all the switches are "off".

Since [11] provides the linked list implementation details for a conventional LRU cache, these details are excluded from Fig. 4, thus Fig. 4 only depicts the additional hardware required for CaPPS. The bypass transfer line and n-MOS switches can be shared by all sets,

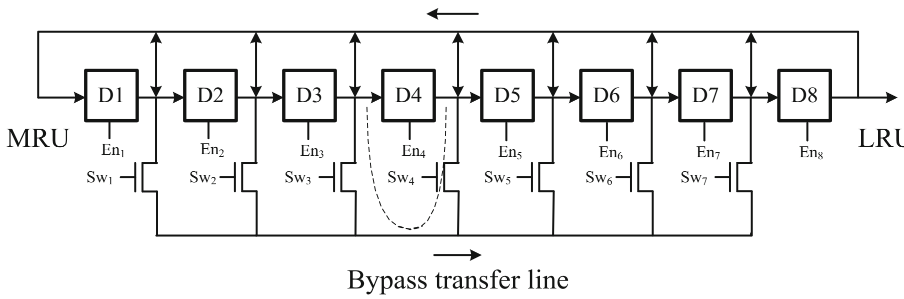

Fig. 4 Extension of the linked list implementation for CaPPS

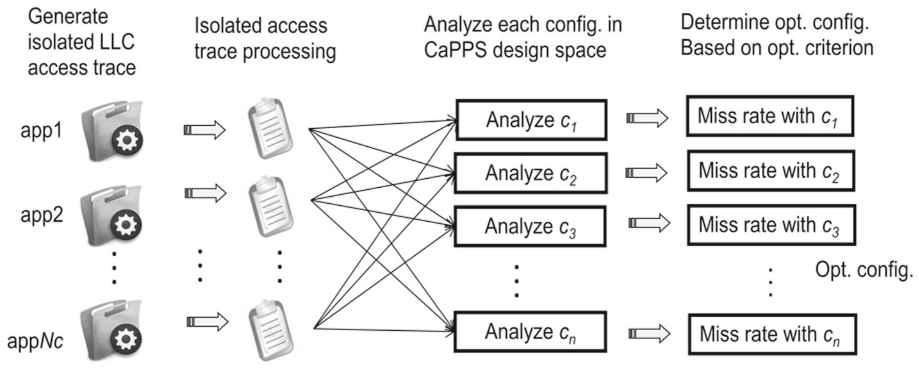
Generate isolated LLC access trace
Isolated access trace processing
Analyze each config. in CaPPS design space
Determine opt. config. Based on opt. criterion
app1
app2
appNc
Analyze c1
Analyze c2
Analyze c3
Analyze cn
Miss rate with c1
Miss rate with c2
Miss rate with c3
Miss rate with cn
Opt. config.

Fig. 5 Steps for using the analytical model for LLC optimization, where c_i represents one configuration in CaPPS's design space

therefore the additional hardware cost is minimal. The control logic for the "load enable" of the linked list registers and switches is straightforward, thus we omit those details.

3.2 Analytical modeling overview

For applications with fully/partially shared ways, the analytical model probabilistically determines the miss rates, considering contention effects, using the isolated cache access distributions for the co-executing applications. For applications with only private ways, there is no cache contention and the miss rate can be directly determined from the isolated LLC access trace distribution.

Figure 5 summarizes the steps for using the analytical model for LLC optimization. During offline processing, the isolated LLC access trace for each application is generated with a simulator/profiler by running each application in isolation on a single core with all other cores idle. Next, the isolated cache access distributions are determined during *isolated access trace processing*. For the possible combinations of co-executing applications, analytical modeling exhaustively estimates the LLC miss rates for each configuration in CaPPS's design space by analyzing the shared ways' contention. Finally, the optimal configuration for the co-executing applications is determined based on the optimization criterion (e.g., minimum average LLC miss rate or CPI, minimum energy consumption, etc.).

The offline-determined optimal configurations for the possible combinations of co-executing applications can be stored in a small, custom-hardware look-up table or a small hash table in memory. During runtime, the CMP directly selects the offline-determined optimal configurations for the co-executing applications. Since our analytical model probabilistically predicts the optimal configuration, each application can begin execution and terminate at any point in time without affecting our offline-determined optimal configurations, and therefore CaPPS inherently supports context switching.

Since our analytical model affords fast execution times (Sect. 4.2.2), the optimal configuration can be determined by exhaustively exploring CaPPS's design space, however, we note that an empirically-designed exploration heuristic could be employed to further reduce this design space exploration time at the cost of potentially suboptimal cache configurations. Our work focuses on exhaustive exploration since the main contributions of this paper are proposing a hybrid LLC organization that combines the benefits of private and shared partitioning, and developing an analytical model to quickly and accurately estimate the miss rates to assist in LLC optimization. Therefore, we leave heuristic exploration methods as future work. Additionally, since our analytical model's exhaustive exploration time is only 2–3 h (Sect. 4.2.2), which is reasonable for offline cache configuration, exploration heuristics would not afford a significant reduction in design-time effort.

3.3 Isolated access trace processing

To accumulate the isolated cache access distribution and capture the accesses' temporal behavior, we record the *reuse distance* and *stack distance* for each access in the isolated LLC access trace, which can be obtained using a stack-based trace-driven simulator [13]. For an accessed address T that maps to a cache set, the reuse distance r is the number of accesses to that set between this access to T and the previous access to any address in the same block as T, including this access to T. The stack distance d is the number of unique block addresses, or *conflicts*, in this set of accesses excluding T. For example, in Fig. 6, C1's second access to X_1 has $r = 7$ and $d = 4$.

In each cache set, we accumulate the number of accesses N_d for each stack distance d ($d \in [0, A]$), where A is the LLC associativity. We accumulate the number of accesses with $d > A$ in N_A together with the number of accesses with $d = A$, since all accesses with $d \geq A$ are cache misses in any configuration. Given this information, for any access, the probabilistic information for the access' stack distance is $p\,(d < d_i) = \left(\sum_{d=0}^{d=d_i-1} N_d\right) / \left(\sum N_d\right)$ and $p\,(d \geq d_i) = 1 - p\,(d < d_i)$, $(\forall d_i \in [1, \text{A}])$. For all of the accesses for each d, we accumulate a histogram of different r and calculate the average $\bar{r}$ over all r.

The analytical model uses the base (best case) CPU cycles $Cycles_{base}$ to calculate the CPU cycles required to complete the application when co-executed with other applications (Sect. 3.4.2). $Cycles_{base}$ assumes that all LLC accesses are hits. An application's total number of CPU cycles $Cycles_{exe}$ are recorded for a single isolated execution to calculate $Cycles_{base}$ using $Cycles_{base} = Cycles_{exe} - m_{exe} \cdot LLC_{latency}$, where m_{exe} is the number of LLC misses

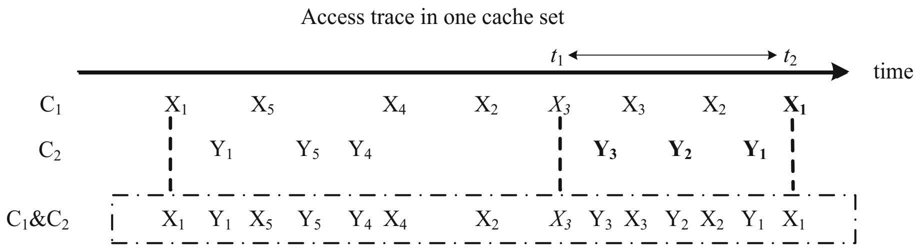
Access trace in one cache set
t_1
t_2
time
C₁
X₁ X₅ X₄ X₂ X₃ X₃ X₂ X₁
C₂
Y₁ Y₅ Y₄ Y₃ Y₂ Y₁
C₁&C₂
X₁ Y₁ X₅ Y₅ Y₄ X₄ X₂ X₃ Y₃ X₃ Y₂ X₂ Y₁ X₁

Fig. 6 Two cores' isolated (C_1, C_2) and interleaved (C_1&C_2) access traces for an arbitrary cache set

in the application's isolated execution and $LLC_{latency}$ is the delay cycles incurred by an LLC miss.

Since the access distributions across the cache sets are different, the distributions are individually accumulated and recorded for each set to estimate the number of misses in each set's accesses. Since the analysis is the same for all cache sets, we present the analytical model for one arbitrary cache set.

3.4 Analysis of the shared ways' contention

First, we describe the analytical model to analyze the contention in the shared ways for a sample CMP with two cores C_1 and C_2 and then generalize the analytical model to any number of cores in Sect. 3.4.4.

Figure 6 depicts the contention effects in the shared ways using sample time-ordered isolated (C_1, C_2) and interleaved/co-executed (C_1&C_2) access traces to an arbitrary cache set from cores C_1 and C_2. C_1 and C_2's accesses are denoted as X_i and Y_i, respectively, where i differentiates accesses to unique cache blocks. The first access to X_3 and the second access to X_1 occurred at times t_1 and t_2, respectively. C_1's second access to X_1 will be a cache hit if C_1's number of private ways is greater than or equal to five because four unique blocks are accessed between the two accesses to X_1. Alternatively, if C_1's number of private ways is smaller than five and C_1 shares ways with C_2, X_1's hit/miss is dictated by the interleaved accesses from C_2. For example, if C_1 has six allocated ways and two of the LRU ways are shared with C_2, X_3 evicts X_1 from C_1's private way into a shared way. Therefore, C_2's accesses between t_1 (when X_1 is evicted from C_1's private way) and t_2 (when the X_1 is re-accessed) dictates whether X_1 is in a shared way or has been evicted from the cache. If C_2's accesses between t_1 and t_2 evict two or more blocks into the shared ways, X_1's second access will be a cache miss.

To generalize the example in Fig. 6, we consider a sharing configuration that allocates K_{C_1} number of ways to core C_1, where K_{P,C_1} ways are private and the remaining $K_S \left(K_S = K_{C_1} - K_{P,C_1}\right)$ ways are shared with core C_2. K_{C_2} and K_{P,C_2} similarly denote these values for C_2. For C_1, all accesses with a stack distance $d \leq K_{P,C_1} - 1$ result in cache hits in the private ways and all accesses with $d \geq K_{C_1}$ are cache misses. The only undetermined cache hits/misses are the accesses where $K_{P,C_1} \leq d \leq K_{C_1} - 1$, which depend on the interleaved accesses from C_2. Thus, the following subsections elaborate on the estimation method for these accesses. If C_1 only has private ways, then $K_{P,C_1} = K_{C_1}$, and estimating the contention in the shared ways contention is not required. The number of hits for C_1 can be directly calculated using $\sum_{d=0}^{d=K_{C_1}-1} N_{d,C_1}$.

In order to determine the contention effects to C_1's miss rate, C_1 and C_2's number of accesses n_{C_1} (Sect. 3.4.1) and n_{C_2} (Sect. 3.4.2), respectively, during the time period (t_1, t_2), must be estimated. Since the number of blocks R_{C_2} from n_{C_2} evicted into the shared ways dictates whether C_1's blocks (e.g., X_1 in Fig. 6) are still in the shared ways, we calculate the probability $p\left(n_{C_2}, R_{C_2}\right)$ that R_{C_2} number of blocks are evicted into the shared ways (Sect. 3.4.3) to estimate C_1's miss rate (Sect. 3.4.4).

3.4.1 Calculation of n_{C_1}

For an arbitrary stack distance D in $[K_{P,C_1}, K_{C_1} - 1]$, the associated $\bar{r}$ was determined during isolated access trace processing. This subsection presents the calculation of n_{C_1} for C_1's accesses with stack distance D based on $\bar{r}$.

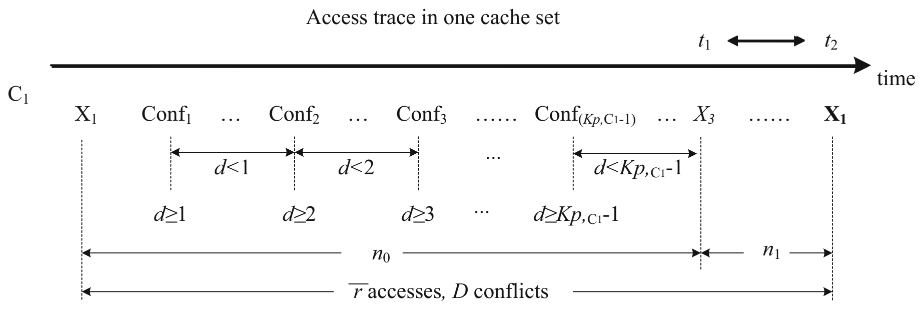

Fig. 7 C_1's isolated access trace to an arbitrary cache set to illustrate the calculation of n_1

Figure 7 depicts C_1's isolated access trace to an arbitrary cache set, where the second access to X_1 has a stack distance D and reuse distance $\bar{r}$. X_3's access evicts X_1 from C_1's private ways, therefore, the number of conflicts before and after X_3 are $K_{P,C_1}-1$ (excluding X_3) and $D-(K_{P,C_1}-1)$ (including X_3), respectively. Conf_i denotes the first access of the i-th conflict with X_1. We denote the number of accesses before X_3 as n_0 and n_{C_1} can be calculated by $n_{C_1}=\bar{r}-n_0-1$. To simplify the computation, we represent n_0 and n_{C_1} using the expected values $\bar{n}_0$ and $\bar{n}_{C_1}$, respectively, in the subsequent calculations in Sects. 3.4.2, 3.4.3, 3.4.4.

n_0 can be any integer in $\left[K_{P,C_1}-1,\ \bar{r}-\left(D-K_{P,C_1}\right)-2\right]$ since there are at least $K_{P,C_1}-1$ conflicts before X_3 and at least $D-K_{P,C_1}$ conflicts after X_3. After determining the probability $p\left(n_0,\left(K_{P,C_1}-1\right)\right)$ for each n_0 (where $K_{P,C_1}-1$ indicates the number of conflicts in the n_0 accesses), we can calculate n_0's expected value $\bar{n}_0$ for the evaluated configuration's associated K_{P,C_1} using:

$$\bar{n}_0=\sum\left(n_0\cdot p\left(n_0,\left(K_{P,C_1}-1\right)\right)\right) \tag{1}$$

and n_{C_1}'s expected value is:

$$\bar{n}_{C_1}=\bar{r}-\bar{n}_0-1 \tag{2}$$

For a particular $n_0\in\left[K_{P,C_1}-1,\bar{r}-\left(D-K_{P,C_1}\right)-2\right]$, the probability $p\left(n_0,\left(K_{P,C_1}-1\right)\right)$ is:

$$p\left(n_0,\left(K_{P,C_1}-1\right)\right)=p\left(E_A,E_B|E_C\right)=\frac{p_{before}\left(E_A\right)\cdot p_{after}\left(E_B\right)}{p_{total}\left(E_C\right)} \tag{3}$$

where E_A is the event that the n_0 accesses have exactly $K_{P,C_1}-1$ conflicts and E_B is the event that the n_{C_1} accesses have exactly $D-(K_{P,C_1}-1)$ conflicts. $p_{before}\left(E_A\right)$ and $p_{after}\left(E_B\right)$ are the occurrence probabilities of E_A and E_B, respectively. E_C is the event that the $\bar{r}$ accesses have exactly D conflicts and $p_{total}\left(E_C\right)$ is the probability of E_C's occurrence, which is the summation of $\left(p_{before}\left(E_A\right)\cdot p_{after}\left(E_B\right)\right)$ for all possible n_0.

To calculate $p_{before}\left(E_A\right)$ and $p_{after}\left(E_B\right)$, we examine the sufficient conditions that E_A and E_B occur. In the example of Fig. 7, the first access following X_1 must be different from X_1 (for $D>0$), which is Conf_1 satisfying $d\geq1$, since Conf_1 has at least one conflict: X_1. The second conflict Conf_2 satisfies $d\geq2$, since Conf_2 has at least two conflicts: Conf_1 and X_1. The accesses between Conf_1 and Conf_2 satisfy $d<1$ since these accesses can only be Conf_1. Conf_3 satisfies $d\geq3$ since Conf_3 has at least three conflicts: Conf_2, Conf_1, and X_1. The accesses between Conf_2 and Conf_3 satisfy $d<2$, since these conflicts can only be Conf_2 or Conf_1, etc. Similarly, $\text{Conf}_{K_{p,C_1}-1}$ satisfies $d\geq\left(K_{P,C_1}-1\right)$ and the accesses between X_3 and $\mathrm{Conf}_{K_{p,\mathrm{C}_1}-1}$ satisfy $d < \left(K_{P,\mathrm{C}_1} - 1\right)$. Therefore, defining a vector $\vec{a} = \left(a_1, a_2, \ldots, a_{K_{p,\mathrm{C}_1}-1}\right)$ where $a_i \in \left[0, n_0 - \left(K_{P,\mathrm{C}_1} - 1\right)\right]$, $p_{before}\left(E_A\right)$ is:

$$p_{before}\left(E_A\right) = \left\{\prod_{i=1}^{i=K_{P,\mathrm{C}_1}-1} p\left(d \geq i\right)\right\} \cdot \left\{\sum_{\forall \vec{a} \in S_a} \left(\prod_{i=1}^{i=K_{P,\mathrm{C}_1}-1} p\left(d < i\right)^{a_i}\right)\right\} \tag{4}$$

where S_a is a set including all $\vec{a}$ satisfying $\sum a_i = n_0 - \left(K_{P,\mathrm{C}_1} - 1\right)$. The first multiplicand in the equation computes the probability that there are $K_{P,\mathrm{C}_1} - 1$ number of Conf_i and the second multiplicand computes the probability of all of the cases that in the remaining $n_0 - \left(K_{P,\mathrm{C}_1} - 1\right)$ accesses, there are exactly a_i number of accesses occur between Conf_i and Conf_{i+1} for each $i \in \left[1, K_{P,\mathrm{C}_1} - 1\right]$. Similarly, defining a vector $\vec{b} = \left(b_0, b_1, \ldots, b_{D-K_{p,\mathrm{C}_1}}\right)$ where $b_i \in \left[0, n_{\mathrm{C}_1} - \left(D - K_{P,\mathrm{C}_1} + 1\right)\right]$, $p_{after}\left(E_B\right)$ is:

$$\begin{aligned} p_{after}\left(E_B\right) = &\left\{\prod_{i=0}^{i=D-K_{P,C_1}} p\left(d \geq i + K_{P,C_1}\right)\right\} \\ &\times \left\{\sum_{\forall \vec{b} \in S_b} \left(\prod_{i=0}^{i=D-K_{P,C_1}} p\left(d < i + K_{P,C_1}\right)^{b_i}\right)\right\} \end{aligned} \tag{5}$$

where S_b is a set including all $\vec{b}$ satisfying $\sum b_i = n_{\mathrm{C}_1} - \left(D - K_{P,\mathrm{C}_1} + 1\right)$.

To reduce the computational complexity, the second multiplicand in (4) can be substituted with $p\left(l, k\right)$, where k represents the number of Conf_i and l represents the remaining number of accesses in the n_0 accesses before X_3, $\left(\text{i.e., } k = \left(K_{P,\mathrm{C}_1} - 1\right) \text{ and } l = n_0 - \left(K_{P,\mathrm{C}_1} - 1\right)\right)$. Thus:

$$p\left(l, k\right) = \sum_{\forall \vec{a} \in S_a} \left(\prod_{i=1}^{k} p\left(d < i\right)^{a_i}\right) \tag{6}$$

where $\vec{a} = \left(a_1, a_2, \ldots, a_k\right)$ satisfying $a_i \in [0, l]$ and S_a is a set including all $\vec{a}$ satisfying $\sum a_i = l \cdot p\left(l, k\right)$ can be derived using induction as:

$$p\left(l, k\right) = p\left(l - 1, k\right) \cdot p\left(d < k\right) + p\left(l, k - 1\right) \tag{7}$$

with the initial cases $p\left(l, 1\right) = p\left(d < 1\right)^l$ and $p\left(0, k\right) = 1$. The induction of $p\left(l, k\right)$ is calculated from $k = 1$ (i.e., $K_{P,\mathrm{C}_1} = 2$) since $K_{P,\mathrm{C}_1} = 0$ indicates that there is no private way and $\bar{n}_{\mathrm{C}_1} = \bar{r}$, and $K_{P,\mathrm{C}_1} = 1$ indicates one private way and the first access after X_1 evicts X_1 into the shared ways, thus $\bar{n}_{\mathrm{C}_1} = \bar{r} - 1$. The induction of $p\left(l, k\right)$ means that in the n_0 accesses, if the last access is not Conf_k, the previous $n_0 - 1$ accesses must contain all the k number of Conf_i and the last access satisfies $d < k$. If the last access in the n_0 accesses is Conf_k, the previous $n_0 - 1$ accesses must contain $k - 1$ number of Conf_i.

Similarly, the second multiplicand in (5) can be substituted with $\mathrm{p}\left(\mathrm{l}', \mathrm{k}'\right)$ where $l' = n_{C_1} - \left(D - K_{P,C_1} + 1\right)$ and $\mathrm{k}' = D - K_{P,C_1} + 1$, and the induction of $p\left(l', k'\right)$ is:

$$p\left(l', k'\right) = p\left(l' - 1, k'\right) \cdot p\left(d < D - k' + 1\right) + p\left(l', k' - 1\right) \tag{8}$$

with the initial cases $p\left(l', 1\right) = p\left(d < D\right)^{l'}$ and $p\left(0, k'\right) = 1$.

After substituting $p\left(l, k\right)$ and $p\left(l', k'\right)$ in (4) and (5), $p\left(n_0, \left(K_{P,\mathrm{C}_1} - 1\right)\right)$ is calculated using (3), and $\bar{n}_0$ and $\bar{n}_{\mathrm{C}_1}$ can be determined using (1) and (2).

3.4.2 Calculation of n_{C_2}

We model n_{C_i} using a Poisson distribution $p(n_{\mathrm{C}_i}) = \mathit{Poisson}\left(n_{\mathrm{C}_i}, \lambda_{\mathrm{C}_i}\right)$, where λ_{C_i} is the number of cache set accesses per cycle. $\lambda_{\mathrm{C}_i} = \sum N_{d,\mathrm{C_i}}/\widehat{Cycles}_{\mathrm{C_i}}$ if the LLC is accessed randomly. $\sum N_{d,\mathrm{C_i}}$ is the total number of LLC accesses from $\mathrm{C_i}$. $\widehat{Cycles}_{\mathrm{C_i}}$ is the number of CPU cycles required to execute the application on $\mathrm{C_i}$. However, since the LLC's accesses are generally not strictly random in time, we use an empirical variable e to adjust λ_{C_i} to λ_{C_i}/e. Our experiments indicated that $e = 5$ was appropriate for our training benchmark suite, which contains a wide variety of typical CMP applications, and is thus generally applicable.

To determine the contention effects from C_2, the expected number of accesses $\bar{n}_{\mathrm{C}_2}$ from C_2 is estimated based on the ratio of the expected number of accesses from C_1 and C_2 per cycle:

$$\frac{\bar{n}_{\mathrm{C}_1}}{\bar{n}_{\mathrm{C}_2}} = \frac{\lambda_{\mathrm{C}_1}/e}{\lambda_{\mathrm{C}_2}/e} = \frac{\sum N_{d,\mathrm{C}_1}/\widehat{Cycles}_{\mathrm{C}_1}}{\sum N_{d,\mathrm{C}_2}/\widehat{Cycles}_{\mathrm{C}_2}} \tag{9}$$

$\widehat{Cycles}_{\mathrm{C}_i}$ can be calculated using $Cycles_{\mathrm{C}_i,base}$ and the estimated number of LLC misses $\widehat{m_{\mathrm{C}_i}}$ considering the contention in the shared ways:

$$\widehat{Cycles}_{\mathrm{C}_i} = Cycles_{\mathrm{C}_i,base} + \widehat{m_{\mathrm{C}_i}} \cdot LLC_{latency} + delay_{bus_{cont}} \tag{10}$$

where $delay_{bus\_cont}$ is the delay imposed by the shared bus contention from the higher level caches (closer to the CPU) of each core to the shared LLC.

$delay_{bus\_cont}$ can be estimated considering that each higher level cache miss requests the bus twice when a read/write request is sent to the LLC and the LLC is returning the requested block. Thus, the bus cycles used by each higher level cache miss is $busCycles = busCycles_{send\_req} + busCycles_{send\_block}$ and the probability pb_i that one bus cycle is used by a core C_i is $pb_i = \exp(-\lambda) \cdot \lambda$, where $\lambda = \left(\sum N_{d,\mathrm{C_i}} \cdot busCycles\right) / \left(\widehat{Cycles}_{\mathrm{C_i}} / \left(f_{CPU}/f_{bus}\right)\right) / e$, f_{CPU} and f_{bus} are the CPU and bus frequencies, respectively.

To calculate $delay_{bus\_cont}$, we assume random bus arbitration. When j additional cores are using/requesting the bus concurrently with a C_1's bus request, there are three cases to consider. In the first case, v_1 number of cores are sending read/write requests to the LLC, and the expected delayed bus cycles for C_1 is:

$$busdelay_1(v_1) = \frac{1}{v_1+1} \cdot \sum_{i=1}^{v_1} i \cdot busCycles_{send\_req} = \frac{v_1}{2} \cdot busCycles_{send\_req} \tag{11}$$

This equation indicates that C_1's bus request may directly be serviced or may stall for one, two, three, or more cycles for v_1 number of cores to be serviced. Each of the possible waiting time's occurrence's probability is $1/(v_1+1)$.

In the second case, v_2 cores are sending requests to the bus to receive requested blocks from the LLC. Similar to (11), the expected delayed bus cycles for C_1 is:

$$busdelay_2(v_2) = \frac{1}{v_2+1} \cdot \sum_{i=1}^{v_2} i \cdot busCycles_{send\_block} = \frac{v_2}{2} \cdot busCycles_{send\_block} \tag{12}$$

In the third case, v_3 cores are in the process of receiving the requested block from the LLC and C_1's request must stall while the other cores are using the bus. Defining a vector

$\vec{e} = \left(e_1, e_2, \ldots, e_{Cycles_{send\_block}-1}\right)$ where $e_i \in [0, v_3]$, the expected delayed bus cycles for C_1 is:

$$busdelay_3\left(v_3\right) = \sum_{\forall \vec{e} \in S_e}\left(\left(\sum_{i=1}^{busCycles_{send\_block}-1}\left(e_i \cdot i\right)\right) \cdot p\left(e_{comb}\right)\right) \tag{13}$$

where S_e is a set including all $\vec{e}$ satisfying $\sum e_i = v_3$. e_i denotes that e_i number of cores have i number of bus cycles remaining to finish the transfer of the requested block. The probability of each $\vec{e}$ in S_e is equal, $p\left(e_{comb}\right) = 1/\left|S_e\right|$, where $\left|S_e\right|$ is the cardinality of S_e and is calculated by $\left|S_e\right| = C\left(busCycles_{send\_block} + v_3 - 2,\ busCycles_{send\_block} - 2\right)$, where $C\left(n, r\right)$ denotes the number of combinations of r numbers selected from a set with n numbers.

Considering the three cases and defining a vector $\vec{v} = \left(v_1, v_2, v_3\right)$ where $v_i \in [0, j]$, the expected delayed bus cycles for C_1 when j number of additional cores are using/requesting the bus is:

$$busdelay\left(j\right) = \sum_{\forall \vec{v} \in S_v}\left(\left(busdelay_1\left(v_1\right) + busdelay_2\left(v_2\right) + busdelay_3\left(v_3\right)\right) \cdot p\left(v_{comb}\right)\right) \tag{14}$$

where S_v is a set including all $\vec{v}$ satisfying $\sum v_i = j$. The probability of each $\vec{v}$ in S_v is equal, $p\left(v_{comb}\right) = 1/\left|S_v\right|$, where $\left|S_v\right|$ is the cardinality of S_v and is calculated by $\left|S_v\right| = C\left(j+2, 2\right)$.

Therefore, the $delay_{bus\_cont}$ for C_1 with respect to the CPU cycles is:

$$delay_{bus\_\mathrm{cont}, \mathrm{C}_1} = 2 \cdot \sum N_{d,\mathrm{C}_1} \cdot \left(\frac{f_{CPU}}{f_{bus}}\right) \cdot \sum_{j=1}^{j=N_C-1}\left(p\left(j\right) \cdot busdelay\left(j\right)\right) \tag{15}$$

where N_C is the total number of cores, $(2 \cdot \sum N_{d,\mathrm{C}_1})$ is the total number of bus requests from C_1, and $p\left(j\right)$ is the probability that j number of additional cores are using/requesting the bus when C_1 is requesting the bus. Using a_i and b_i to represent the identification of the cores that are and are not using/requesting the bus when C_1 is requesting the bus, respectively, and defining a vector $\vec{w} = \left(a_1,\ a_2, \ldots, a_j,\ b_1, b_2, b_3, \ldots, b_{N_C-1-j}\right)$ where $a_i, b_i \in [1, N_C]$, $p\left(j\right)$ can be calculated as:

$$p\left(j\right) = \frac{1}{C\left(N_\mathrm{C} - 1, j\right)} \cdot \sum_{\forall \vec{w} \in S_w}\left\{\left(\prod_{i=1}^{j} pb_{a_i}\right) \cdot \left(\prod_{i=1}^{N_\mathrm{C}-1-j}\left(1 - pb_{b_i}\right)\right)\right\} \tag{16}$$

where S_w is a set including all $\vec{w}$ satisfying $a_1 < a_2 < a_3 < \cdots < a_j$, $b_1 < b_2 < b_3 < \cdots < b_{N_C-1-j}$, and $a_i \neq b_i \neq 1$.

3.4.3 Calculation of $p\left(n_{\mathrm{C}_2}, R_{\mathrm{C}_2}\right)$

$p\left(n_{\mathrm{C}_2}, R_{\mathrm{C}_2}\right)$ is the probability that R_{C_2} number of blocks are evicted from C_2's private ways in the n_{C_2} accesses. Directly using the expected n_{C_2} (Sect. 3.4.2) to calculate $p\left(\bar{n}_{\mathrm{C}_2},\ R_{\mathrm{C}_2}\right)$ will introduce a large bias (approximate 10 % error) in the estimated LLC miss rate, since different values of n_{C_2} result in different hit/miss determinations and using one expected value $\bar{n}_{\mathrm{C}_2}$ will estimate all n_{C_2} as hits/misses. For example, in an extreme case where $K_S = 1$ and $D = K_{P,\mathrm{C}_1} + 1$, the accesses with this D result in cache hits in a shared way if $n_{\mathrm{C}_2} = 0$.

Therefore, if the expected value $\bar{n}_{C_2}$ is used to determine the cache hit/miss and if $\bar{n}_{C_2} > 0$, all of the accesses with this D will be evaluated as cache misses. However, although the expected value $\bar{n}_{C_2} > 0$, some n_{C_2} can be zeros, which result in cache hits.

Thus, instead of directly using $\bar{n}_{C_2}$, we use distributed n_{C_2} values. Since the range of n_{C_2} is infinite in the Poisson distribution, and n_{C_2} with very small $p(n_{C_2})$ has a minimal effect on the miss rate estimation, we only consider the n_{C_2} with $p(n_{C_2}) > 0.01$ and calculate the associated $p\left(n_{C_2}, R_{C_2}\right)$.

To calculate $p\left(n_{C_2}, R_{C_2}\right)$ for an arbitrary n_{C_2}, R_{C_2} is determined by evaluating the n_{C_2} accesses in chronological order with an initial value of $R_{C_2} = 0$. If there is one access with $d > K_{P,C_2}$, fetching this address into C_2's private ways will evict one block into the shared ways and thus R_{C_2} is incremented by 1. R_{C_2} remains the same until the subsequent accesses include one access with $d > K_{P,C_2} + 1$, and one additional block will be evicted into the shared ways by fetching the accessed block into C_2's private ways, in which case R_{C_2} is incremented. The same condition (i.e., one access with $d > K_{P,C_2} + current\ R_{C_2}$) to increment R_{C_2} applies to the remaining accesses, therefore, we can calculate $p\left(n_{C_2}, R_{C_2}\right)$ inductively:

$$p\left(n_{C_2}, R_{C_2}\right) = \begin{cases} p\left(n_{C_2}-1, R_{C_2}-1\right) \cdot p\left(d \geq K_{P,C_2} + \left(R_{C_2}-1\right)\right), & R_{C_2} = n_{C_2} \\ p\left(n_{C_2}-1, R_{C_2}\right) \cdot p\left(d < K_{P,C_2} + R_{C_2}\right) & \\ + p\left(n_{C_2}-1, R_{C_2}-1\right) \cdot p\left(d \geq K_{P,C_2} + \left(R_{C_2}-1\right)\right), & R_{C_2} = n_{C_2} \\ p\left(n_{C_2}-1, R_{C_2}\right) \cdot p\left(d < K_{P,C_2} + R_{C_2}\right), & R_{C_2} = 0 \end{cases} \tag{17}$$

with the initial case $p(n_{C_2} = 0, R_{C_2} = 0) = 1$. There are three cases in the induction. In the first case, $R_{C_2} = n_{C_2}$, all n_{C_2} accesses will evict one additional block into the shared ways, thus, in the n_{C_2} accesses, the previous $n_{C_2} - 1$ accesses evict $R_{C_2} - 1$ blocks and the last access satisfies $d \geq K_{P,C_2} + \left(R_{C_2} - 1\right)$. In the second case, $R_{C_2} < n_{C_2}$, in the n_{C_2} accesses, the previous $n_{C_2} - 1$ accesses either have evicted R_{C_2} blocks into the shared ways, in which case the last access satisfies $d < K_{P,C_2} + R_{C_2}$, or the previous $n_{C_2} - 1$ accesses have evicted $R_{C_2} - 1$ blocks, in which case the last access satisfies $d \geq K_{P,C_2} + \left(R_{C_2} - 1\right)$ and evicts one additional block. In the third case, $R_{C_2} = 0$, no block is evicted into the shared ways in the n_{C_2} accesses.

3.4.4 Calculation of the LLC miss rates

Considering the impact of R_{C_2} to the accesses with $d \in [K_{P,C_1}, K_{C_1} - 1]$, the number of cache hits for C_1 is:

$$\widehat{h_{C_1}} = \sum_{d=0}^{d=K_{p,C_1}-1} N_{d,C_1} + \sum_{d=K_{p,C_1}}^{d=K_{C_1}-1} \left(N_{d,C_1} \cdot ph_d\right) \tag{18}$$

where the first addend calculates the total number of hits in the private ways and the second addend calculates the total number of hits in the shared ways for the accesses satisfying $d \in [K_{P,C_1}, K_{C_1} - 1]$. ph_d is the probability of hits for the N_{d,C_1} number of accesses with stack distance d, which is calculated as:

$$ph_d = \sum_{\forall n_{C_2}:\ p\left(n_{C_2}\right) > 0.01} \left(\left(\sum_{R_{C_2}=0}^{R_{C_2}=K_{C_1}-d-1} p(n_{C_2}, R_{C_2}) \right) \cdot p\left(n_{C_2}\right) \right) \tag{19}$$

The calculation of ph_d includes all of the n_{C_2} with $p(n_{C_2}) > 0.01$, and for each n_{C_2}, if the number of blocks R_{C_2} evicted from C_2's private ways is smaller than $K_{C_1} - d$, C_1's accesses with d results in cache hits in the shared ways.

After accumulating $\widehat{h_{C_1}}$ for all cache sets, the number of LLC misses $\widehat{m_{C_1}}$ and the LLC miss rates can be determined.

Finally, we generalize the analytical model to estimate the LLC miss rate for any core C_i when j additional cores (denoted as C_j) share cache ways with C_i by calculating the expected number of accesses $\bar{n}_{C_j}$ from the additional cores during the time (t_1, t_2) and then estimating $p\left(n_{C_j}, R_{C_j}\right)$ similarly as estimating $\bar{n}_{C_2}$ and $p\left(n_{C_2}, R_{C_2}\right)$ for C_2. The generalized expression of (18) is:

$$\widehat{h_{C_i}} = \sum_{d=0}^{d=K_{p,C_i}-1} N_{d,C_i} + \sum_{d=K_{p,C_i}}^{d=K_{C_i}-1} \left(N_{d,C_i} \cdot ph_d\right) \tag{20}$$

where

$$ph_d = \sum_{\forall \vec{C} \in S_C} \left(\prod_{C_j \in \vec{C}} \left(p(n_{C_j}) \cdot p(n_{C_j}, R_{C_j}) \right) \right) \tag{21}$$

where the vector $\vec{C} = \left(n_{C_1}, n_{C_2}, \ldots, n_{C_j}\right)$ with $p(n_{C_j}) > 0.01$ and S_C is a set including all $\vec{C}$ satisfying $\sum R_{C_j} < K_{C_i} - d$.

According to (10), a circular dependency exists where $\widehat{Cycles}$ is used to estimate $\hat{m}$ and $\hat{m}$ is used to calculate $\widehat{Cycles}$. The solution cannot be represented using a closed form, thus we iteratively solve $\hat{m}$. The initial value of $\hat{m}$ is acquired assuming there is no contention (i.e., all K_{C_i} number of ways are privately used by C_i), and $\hat{m}$ is used in (10) to calculate the initial value of $\widehat{Cycles}$. $\widehat{Cycles}$ is provided back into the analytical model to update $\hat{m}$ and the new $\hat{m}$ is used to update $\widehat{Cycles}$. This iterative process continues until a stable $\hat{m}$ (with a precision of 0.001 %) is achieved. Experimental results indicated that only four iterations were required for the results to converge.

The analytical model's runtime complexity depends on the shared LLCs associativity, the number of cores in the CMP, the evaluated sharing configuration (such as the number of ways shared among cores), and the isolated cache access distribution for each application (such as the average reuse distance for each stack distance value). Due to the large number of complex and interdependent variables and unknowns, the complexity analysis is intractable, thus in Sect. 4.2.2, we evaluate the analytical model's measured execution time.

4 Experiment results

Our experiments evaluated the accuracy and time efficiency of the analytical model in estimating the LLC miss rates for CaPPS. We also verified the advantages of partial sharing as compared to two baseline configurations, private partitioning, and constrained partial sharing.

4.1 Experiment setup

We used twelve benchmarks from the SPEC CPU2006 suite [26], which were compiled to Alpha_OSF binaries and executed using the "ref" input data sets. Due to incorrect execution in SimpleScalar 3.0d [3] and gem5 [2], the two simulators used in our experiment, we could not evaluate the complete suite. Even though our work is targeted for embedded systems, we did not use embedded system benchmark suites since these suites contain only small kernels, which do not sufficiently access the LLC, and do not represent our targeted embedded CMP domain.

Since complete execution of the large SPEC benchmarks prohibits exhaustive examination of the entire CaPPS design space, and since most embedded benchmarks have stable behavior during execution, for each SPEC benchmark, we selected 500 million consecutive instructions with similar behavior to use as the *simulation interval* to mimic an embedded application with high LLC occupancy. To select the simulation interval, we performed phase classification on the SPEC benchmarks using SimpleScalar 3.0d and SimPoint 3.2 [12]. Within a benchmark's entire execution, non-overlapping intervals with a fixed length of 100 million instructions were classified into phases, where all of the intervals in the same phase had similar behavior. Since all of the intervals belonging to a phase were not necessarily contiguous, we selected five contiguous intervals that were classified as belonging to the same phase to form the benchmarks' simulation intervals. The SPEC benchmarks' phases were long enough such that every benchmark had five contiguous intervals belonging to the same phase. Table 2 lists the starting instructions for the benchmarks' simulation intervals. Our work can easily be extended to applications with varying behavior (i.e., multiple phases throughout execution) by integrating offline phase change detection methodologies [12,24].

We generated the exact cache miss rates for comparison purposes using SE-mode gem5 and modeled four in-order cores with the TimingSimple CPU model, which stalls the CPU when fetching from the caches and memory. Each core had private level-one (L1) instruction and data caches. The unified level-two (L2) cache and all lower level memory hierarchy components were shared among all cores. We modified the L2 cache replacement operation in gem5 to model the shared LLC for CaPPS. Table 3 shows the parameters used for each system component. Since four cores shared the eight-way LLC, there were 3347 configurations in the CaPPS design space.

Before CaPPS simulation, we executed each benchmark in isolation during the benchmark's simulation interval and recorded the isolated LLC access traces and the CPU cycles $Cycles_{exe}$. For CaPPS simulation, we arbitrarily selected four benchmarks to be co-executed, which formed a benchmark set, and we evaluated sixteen benchmark sets. Since the four benchmarks' simulation intervals were at different execution points, we forced the four cores to simultaneously begin executing at each benchmark's associated simulation interval's starting instruction using a *full-system checkpoint*. A full-system checkpoint gives a

Table 2 The starting instructions (counted from the beginning of the benchmark's execution) for the benchmarks' simulation intervals

Benchmark	Starting instruction (million)	Benchmark	Starting instruction (million)
400.perlbench	6000	445.gobmk	13,500
401.bzip2	64,100	456.hmmer	97,000
429.mcf	13,500	458.sjeng	10,700
433.milc	30,000	462.libquantum	67,800
435.gromacs	15,400	464.h264ref	13,500
444.namd	13,500	473.astar	19,400

Table 3 CMP system parameters

Components	Parameters
CPU	2 GHz clock, single thread
L1 instruction cache	Private, total size of 8 KB, block size of 64 B, 2-way associativity, LRU replacement, access latency of 2 CPU cycles
L1 data cache	Private, total size of 8 KB, block size of 64 B, 2-way associativity, LRU replacement, access latency of 2 CPU cycles
L2 unified cache	Shared, total size of 1 MB, block size of 64 B, 8-way associativity, LRU replacement, access latency of 20 CPU cycles, non-inclusive
Memory	Total size of 3 GB, access latency of 200 CPU cycles
L1 caches to L2 cache bus	Shared, width of 64 B, 1 GHz clock, first come first serve (FCFS) scheduling
Memory bus	Width of 64 B, 1 GHz clock

snapshot of the four-core system state, including the register state, the memory state, the system calls' inputs and outputs, etc. To create the full-system checkpoint, the CMP simulator must terminate each core individually after that core reaches the simulation interval's starting instruction for that core's benchmark. However, since this instruction number is different for each benchmark and gem5 does not support selective core termination (all cores must terminate simultaneously), we created the full-system checkpoint by aggregating the checkpoints of the individual benchmarks executing in isolation. We refer to these checkpoints as *isolated-benchmark checkpoints*, and we generated these checkpoints by fast forwarding each benchmark to the starting instruction of the benchmark's corresponding simulation interval.

For each simulation, the full-system checkpoint was restored and then the system started execution. The system execution was terminated when any core reached 500 million instructions. Due to varying CPU stall cycles across the benchmarks, at the termination point, not all cores had completed executing the simulation interval. However, this termination approach guaranteed that the cache miss rates reflected a fully-loaded system (i.e., full LLC contention since all cores were running during the entire system execution). Since we focused on the cache miss rates and not the absolute number of cache misses, the incomplete benchmarks' execution had no impact on our evaluation. Similarly, due to statistical predictions, the applications are not required to begin execution simultaneously to garner accurate results.

Although our experiments used only four cores and the LLC was a shared 8-way L2 cache, the analytical model itself does not include any limitations on the number of cores, the LLC's hierarchical level, or the cache parameters (e.g., total size, block size, and associativity). Since the design space increases exponentially as the number of cores and number of ways in the LLC increase, experiments on very large systems is not feasible due to the prohibitively long simulation time. Additionally, embedded systems generally have limited scale on the shared cores and cache sizes. Therefore, four cores sharing 8-way LLC can represents most embedded systems.

4.2 Analytical model evaluation

We verified the accuracy of our estimated LLC miss rates obtained using the analytical model and evaluated the analytical model's ability to determine the optimal (minimum LLC miss rate) configuration in the CaPPS design space. Additionally, we illustrated the analytical model's efficiency by comparing the time required to calculate the LLC miss rates as

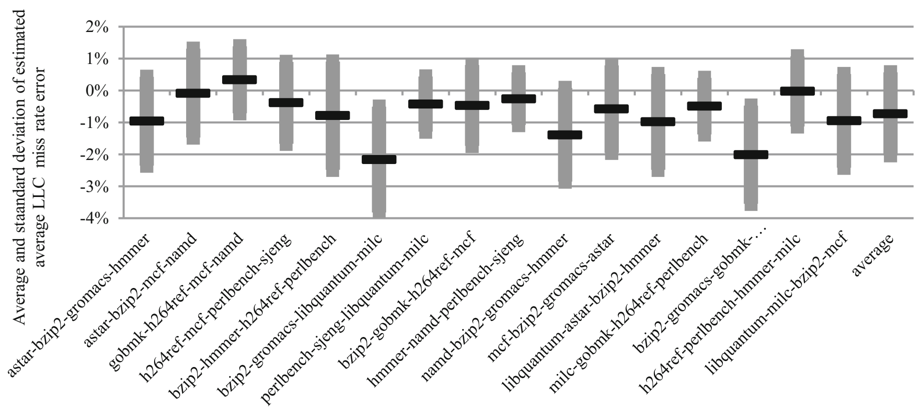

Fig. 8 The average and standard deviation of the average LLC miss rate error determined by the analytical model

compared to using a cycle-accurate simulator to generate the exact cache miss rates for all configurations.

4.2.1 Accuracy evaluation

For each benchmark set, we compared the average LLC miss rate for the four cores determined by the analytical model with the exact miss rate determined by gem5 for each configuration in CaPPS's design space. We calculated the average and standard deviation of the miss rate errors across the 3347 configurations. Figure 8 depicts the results for each benchmark set. The black markers indicate the average miss rate errors and the gray-shaded upper and lower ranges are the corresponding standard deviation. On average, over all sixteen benchmark sets, the absolute average miss rate error and standard deviation were −0.73 and 1.30 %, respectively.

Since the analytical model's cache miss rates are inaccurate, we compared the absolute difference between the LLC miss rates of the analytical model's minimum LLC miss rate configuration and the actual minimum LLC miss rate configuration as determined via exhaustive search. Comparing with an exhaustive search is appropriate for evaluating the analytical model's efficacy, which is only affected by the estimated miss rate errors in determining the optimal configuration. The results indicate that fourteen out of sixteen benchmark sets' differences were less than 1 % and the maximum and average differences over all benchmark sets were 1.30 and 0.35 %, respectively.

4.2.2 Simulation time evaluation

To evaluate the execution time efficiency of the analytical model, we compared the time required to estimate the LLC miss rates (including the time for isolated trace access generation) for all configurations in the CaPPS design space as compared to using gem5. We compared to exhaustive exploration since we are the first to propose CaPPS and therefore, there is no heuristic search to compare to. Furthermore, since the analytical model evaluates each configuration individually using gem5, the average simulation time speedups were nearly independent of the number of evaluated configurations. Therefore the analytical model

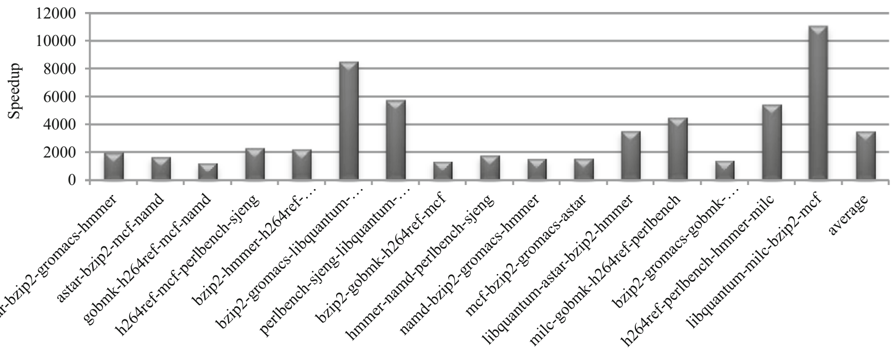

Fig. 9 The analytical model's simulation time speedup as compared to gem5

would show similar speedups even if compared with a heuristic search since the heuristic method could be leveraged by both the analytical model and gem5.

We implemented the analytical model in C++ compiled with O3 optimizations. We tabulated the *user time* reported from the Linux *time* command for the simulations running on a Red Hat Linux Server v5.2 with a 2.66 GHz processor and 4 GB of RAM. Figure 9 depicts the speedup of the analytical model for each benchmark set as compared to gem5. Over all benchmark sets, the average speedup was 3505×, with maximum and minimum speedups of 11,070× and 1235×, respectively. For one benchmark set, the time for simulating all 3347 configurations using gem5 was approximately three months, and comparatively, the analytical model took only 2–3 h.

4.3 CaPPS evaluation

To validate the advantages of CaPPS, we compared CaPPS's ability to reduce the LLC miss rate (i.e., the optimal configuration) as compared to two baseline configurations and configurations as proposed in prior works, including private partitioning and constrained partial sharing.

4.3.1 Comparison with baseline configurations

Figure 10 depicts the average LLC miss rate reductions for CaPPS's optimal configurations as compared to two baseline configurations (1) *even-private-partitioning* the LLC is evenly partitioned using private partitioning; and (2) *fully-shared* the LLC is fully shared by all cores. Across all benchmark sets, the average and maximum average LLC miss rate reductions for CaPPS's optimal configurations were 25.58 and 50.15 %, respectively, as compared to *even-private-partitioning*, and 19.39, and 41.10 %, respectively, as compared to *fully-shared*.

4.3.2 Comparison with private partitioning

We compared CaPPS with private partitioning since prior works typically partition shared caches using private partitioning. Figure 11 depicts the average LLC miss rate reductions for CaPPS's optimal configurations as compared to private partitioning's optimal configurations, which is the configuration with minimum LLC miss rate in the private partitioning's design space consisting of 35 configurations—approximately 1 % of CaPPS's design space. Across

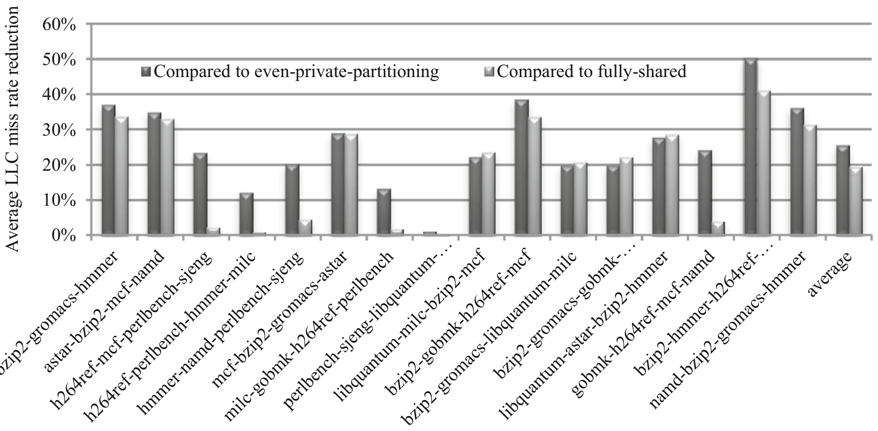

Fig. 10 Average LLC miss rate reductions for CaPPS's optimal configurations as compared to the two baseline configurations: *even-private-partitioning* and *fully-shared*

all benchmark sets, the average and maximum reductions in CaPPS's average LLC miss rates as compared to private partitioning were 16.92 and 43.02 %, respectively. The first three benchmark sets in the figure showed small reductions (less than 2.5 %), which indicates that for these combinations of co-executing applications, exploring the private partitioning design space is sufficient to obtain small LLC miss rates.

Since private partitioning was sufficient for the three benchmark sets, we further evaluated the results to determine which combinations of co-executing applications most benefit from CaPPS's increased design space. We examined the benchmarks' temporal locality characteristics using an in-house-implemented stack-based trace-driven cache simulator [13] to process the isolated L2 cache access traces. Figure 12 depicts the LLC miss rates for varying numbers of cache ways and sizes for each benchmark executing in isolation. Based on these results, we determined the *best number of cache ways* for each benchmark, which allowed the benchmark's entire working set to fit into the LLC. The best number of cache ways is the number of cache ways wherein allocating additional ways does not reduce the miss rate, and therefore wastes cache resources.

We classified the benchmarks into three groups based on the benchmarks' best number of cache ways by evaluating the LLC miss rate trends in Fig. 12, (1) the LLC miss rates for 401.bzip2, 473.astar, and 435.gromacs show a continual decrease as the number of cache ways increases, and the best number of cache ways for these benchmarks is the maximum LLC associativity; (2) the LLC miss rates for 444.namd, 458.sjeng, 445.gobmk, 464.h264ref, 400.perlbench, 456.hmmer, and 429.mcf reach a minimum plateau at a certain number of ways, and the plateau point is the best number of cache ways for the point's associated benchmark; and (3) the LLC miss rates for 462.libquantum and 433.milc are independent of the number of cache ways since the majority of the cache accesses are compulsory misses, and the best number of cache ways for these benchmarks is one.

We can extend this benchmark analysis to predict any set of co-execution applications' LLC partitioning requirements. If the summation of the best number of cache ways for all of the co-executing applications exceeds the LLC associativity, a privately partitioned cache cannot meet the applications' demands. Partial sharing enables co-executing applications to collectively share the LRU ways to enable a larger number of cache ways to be allocated to each core to further reduce the LLC miss rates. However, if the LLC is large enough to

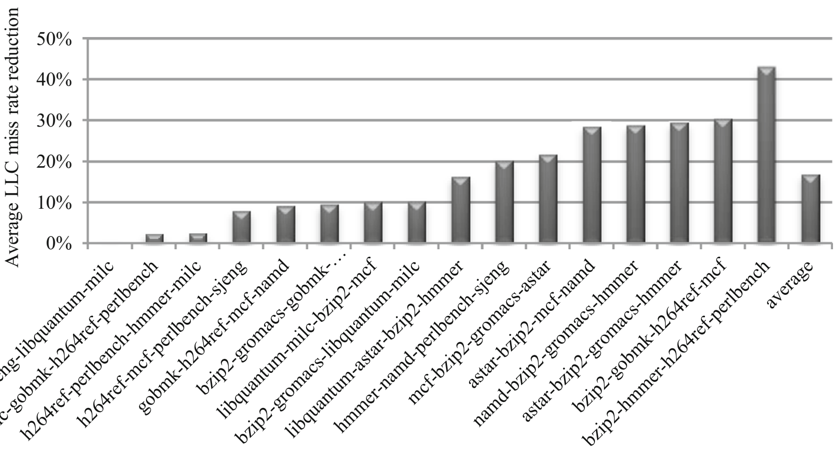

Fig. 11 Average LLC miss rate reductions for CaPPS's optimal configurations as compared to private partitioning's optimal configurations for a 1MB LLC

accommodate the summation of the applications' best number of cache ways, which is the case for the first three benchmark sets in Fig. 11, private partitioning satisfies the applications' requirements, and partial sharing is not necessary.

We also evaluated the benefits of partial sharing for small LLCs since energy- and size-constrained embedded systems must typically use a small LLC. Since the number of cache sets decreases with the cache size, more blocks may map to same cache set, which may increase cache conflicts. This is evident in Fig. 12, where the miss rate trends for the 512 and 256 KB LLCs show that the best number of cache ways for group two benchmarks increases as the LLC size decreases. Additionally, the group one and two benchmarks' LLC miss rates decrease more rapidly in a small LLC as compared to a large LLC.

Therefore, more cache ways are required in a small LLC to obtain low miss rates as compared to a large LLC. Partial sharing alleviates this requirement by enabling larger quotas to be assigned to group one and two applications for small LLCs, which can significantly improve overall cache performance as compared to private partitioning.

To further verify that CaPPS is more suitable for embedded systems with small cache sizes as compared to private partitioning, we created six benchmark sets. Each set contained four arbitrarily selected benchmarks such that the summation of the benchmarks' best number of cache ways exceeded the LLC associativity. Figure 13 depicts the average LLC miss rate reductions for CaPPS's optimal configurations as compared to private partitioning's optimal configurations for 512 and 256 KB LLCs, respectively. For the 512 KB LLC, the average and maximum LLC miss rate reductions were 28.99 and 69.75 %, respectively, and were 30.63 and 45.36 %, respectively, for the 256 KB LLC. The first five benchmark sets showed that LLC miss rate reductions increase as the cache size decreases. For the last benchmark set, the LLC miss rate reduction for the 256 KB LLC was smaller than the 512 KB LLC since the 256 KB LLC was too small to accommodate all of the applications' data, thus, the LLC miss rate reduction in such a small LLC is limited.

4.3.3 Comparison with constrained partial sharing

Even though CaPPS targets shared cache partitioning, the fundamentals associated with partitioning shared and private caches are similar, thus we compare to prior works using constrained partial sharing [9,14,19,27]. If a shared LLC is evenly partitioned across the

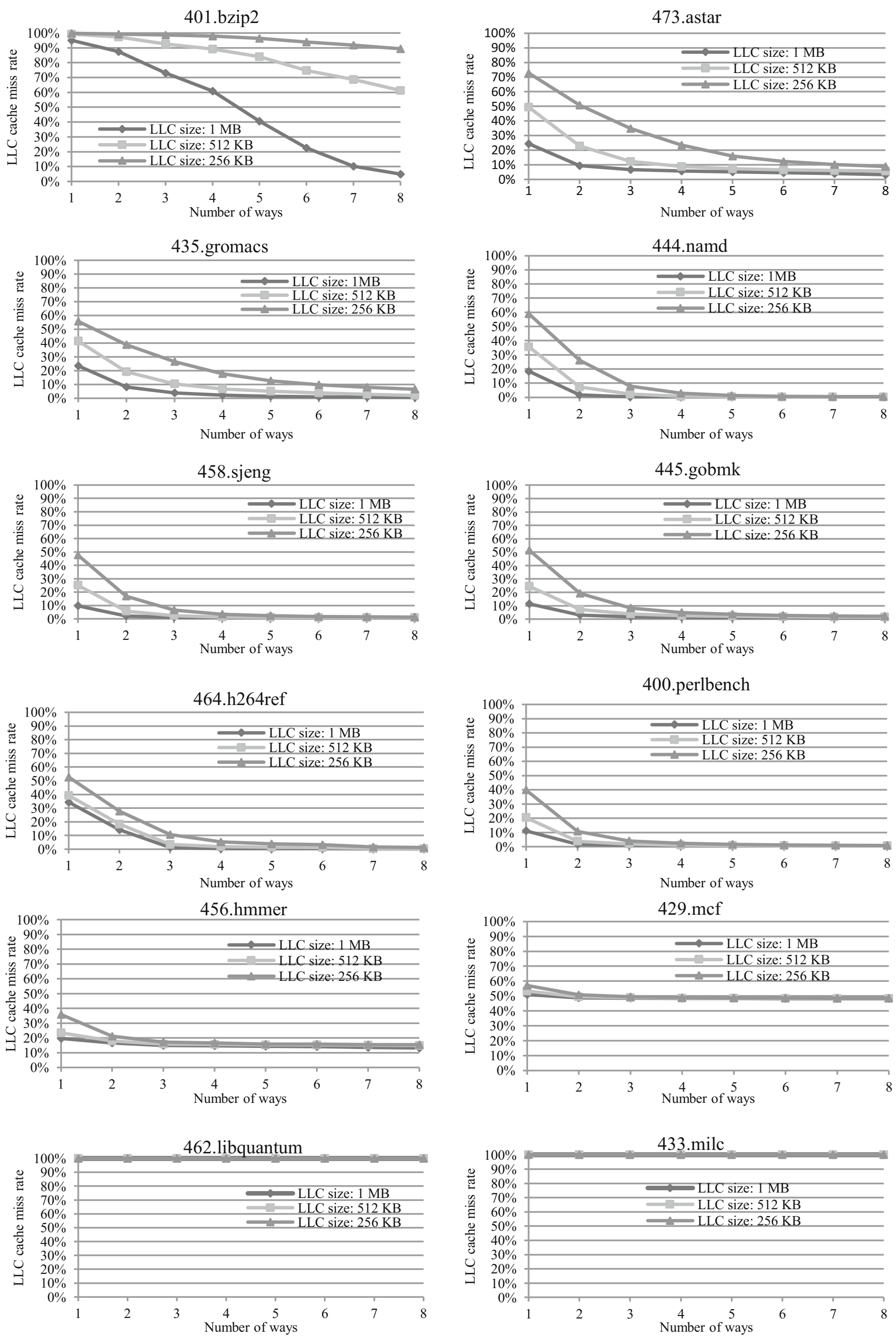

Fig. 12 LLC miss rates for different numbers of cache ways and cache sizes when the benchmarks execute in isolation

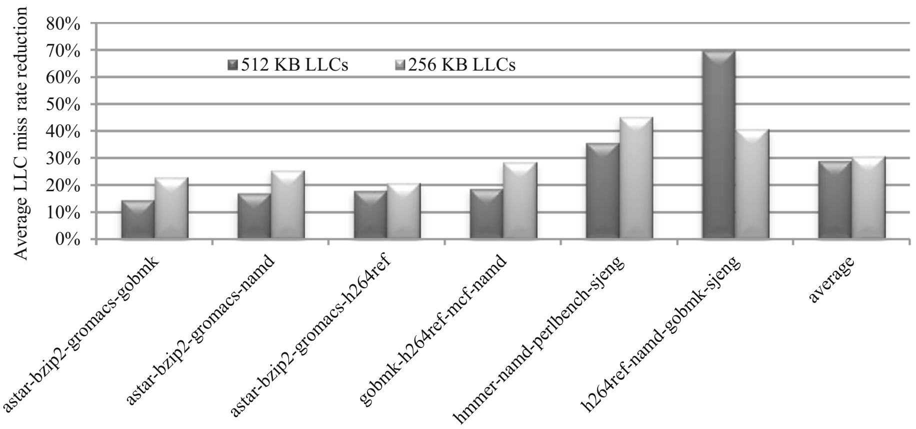

Fig. 13 Average LLC miss rate reductions for CaPPS's optimal configuration as compared to private partitioning for 512 KB and 256 KB LLCs, respectively

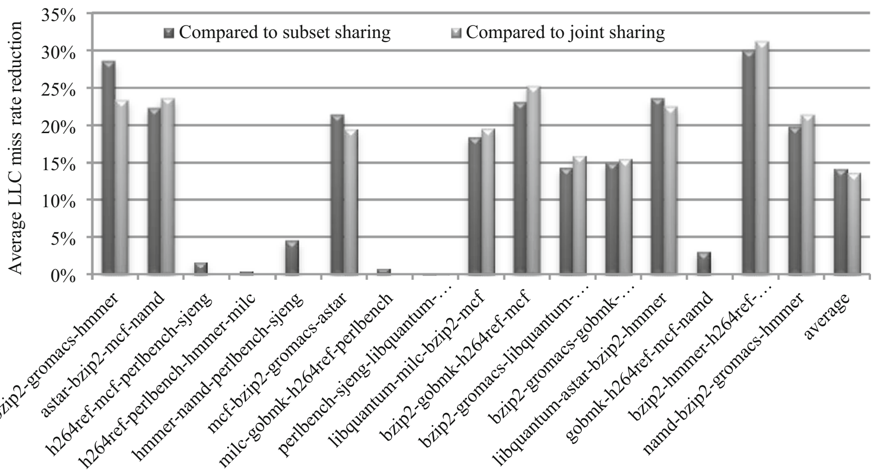

Fig. 14 Average LLC miss rate reductions for CaPPS's optimal configuration as compared to the two constrained partial sharing design spaces

cores and each partition is considered as the cores' private LLC, the constrained partial sharing's design space is a subset of CaPPS's design space. In Sect. 2, we classified prior works into two kinds of constrained partial sharing: subset sharing and joint sharing based on the partitioning and sharing configurability. Using the experiment settings in Table 3, the numbers of configurations in the subset and joint sharings' design spaces are 15 and 81, which account for 0.4 and 2 % of CaPPS's design space, respectively.

Since CaPPS's determines optimal configurations offline and prior works on constrained partial sharing determines configurations online by monitoring the cache performance and then greedily or heuristically determining the configurations, providing a fair comparison is difficult. Since prior constrained partial sharing works used online greedy/heuristic methods, the determined configurations may be suboptimal. Therefore, we determined the *optimal-offline configurations* for subset and joint sharing via exhaustive exploration of the design spaces, and evaluated the miss rate reductions afforded by CaPPS's optimal configurations as compared to the subset and joint sharing's optimal-offline configurations. Exhaustive exploration ensures that the optimal configurations were determined for subset and joint sharing, which places a lower bound on the results and in practice, CaPPS's miss rate reductions would likely be greater than reported.

Figure 14 depicts the average LLC miss rate reductions for CaPPS's optimal configurations as compared to subset and joint sharings' optimal-offline configurations for a 1 MB LLC. The average and maximum LLC miss rate reductions for CaPPS's optimal configurations as compared to the subset sharing were 14.20 and 29.99 %, respectively, and were 13.61 and 31.18 %, respectively, for joint sharing.

5 Conclusions and future work

In this paper, we presented CaPPS—a novel cache partitioning and sharing architecture that improves shared LLC performance with low hardware overhead for CMPs. Our experiments showed that CaPPS reduced the average LLC miss rates by 20–25 % as compared to two baseline configurations, by 17 % as compared to private partitioning, and by 14 % as compared to constrained partial sharing. To quickly estimate the miss rates of CaPPS's sharing configurations, we developed an offline, analytical model that achieved an average miss rate estimation error of only 0.73 %. As compared to exhaustive exploration of the CaPPS design space to determine the lowest energy cache configuration, the analytical model affords an average speedup of 3505×. Finally, CaPPS and the analytical model are applicable to CMPs with any number of cores and place no limitations on the configurable cache parameters.

Future work includes leveraging the offline analytical results to guide online scheduling for performance optimizations in real-time embedded systems, including accesses to a shared address space, and further extending to CMPs executing multi-thread applications, incorporating cache prefetching, and extending CaPPS to proximity-aware cache partitioning for caches with non-uniform accesses.

Acknowledgments This work was supported by the National Science Foundation (CNS-0953447). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. ARM Cortex-A Series. http://www.arm.com/products/processors/cortex-a/index.php
2. Binkert N, Beckmann B, Black G et al. The gem5 Simulator. http://gem5.org
3. Burger D, Austin TM, Bennett S (2000) Evaluating future microprocessors: the Simplescalar Toolset. In: Technical Report, CS-TR-1308. Computer Science Department, University of Wisconsin-Madison, Wisconsin
4. Chandra D, Guo F, Kim S, Solihin Y (2005) Predicting inter-thread cache contention on a chip multiprocessor architecture. In: Proceedings of HPCA, pp 340–351
5. Chang J, Sohi G (2006) Co-operative caching for chip multiprocessors. In: Proceedings of the 33rd annual international symposium on Computer Architecture (ISCA). IEEE, Los Alamitos, pp 264–276
6. Chang J, Sohi G (2014) Cooperative cache partitioning for chip multiprocessors. In: 25th Anniversary international conference on supercomputing anniversary volume. ACM, New York
7. Chen XE, Aamodt TM (2009) A first-order fine-grained multithreaded throughput model. In: Proceedings of HPCA, pp 329–340
8. Chiou D, Chiouy D, Rudolph L, Rudolphy L, Devadas S, Devadasy S, Ang BS (2000) Dynamic cache partitioning via columnization. Computation Structures Group Memo 430. MIT, Cambridge

9. Dybdahl H, Stenstrom P (2007) An adaptive shared/private NUCA cache partitioning scheme for chip multiprocessors. In: Proceedings of HPCA, pp 2–12
10. Eklov D, Black-Schaffer D, Hagersten E (2011) Fast modeling of shared cache in multicore systems. In: Proceedings of HiPEAC, pp 147–157
11. Ghasemzadeh H, Mazrouee S, Moghaddam HG, Shojaei H, Kakoee MR (2006) Hardware implementation of stack-based replacement algorithms. In: Proceedings of world academy of science and technology, vol 16
12. Hamerly G, Perelman E, Lau J, Calder B (2005) SimPoint 3.0: faster and more flexible program analysis. J Instr Level Parallel 7(4):1–28
13. Hill MD, Smith AJ (1989) Evaluating associativity in CPU caches. IEEE Trans Comput 38(12):1612–1630
14. Huh J, Kim C, Shafi H, Zhang L, Burger D, Keckler SW (2007) A NUCA substrate for flexible CMP cache sharing. IEEE Trans Parallel Distrib Syst 18(8):1028–1040
15. Intel Atom Processor. http://www.intel.com/content/www/us/en/intelligent-systems/bay-trail/atom-processor-e3800-family-overview.html
16. Johnson K, Rathbone M (2010) Sun's Niagara Processor. NYU Multicore Programming
17. Kessler RE, Hill MD (1992) Page placement algorithms for large real-indexed caches. ACM Trans Comput Syst 10(4):338–359
18. Kim S, Chandra D, Solihin Y (2004) Fair cache sharing and partitioning in a chip multiprocessor architecture. In: Proceedings of PACT, pp 111–122
19. Lee H, Cho S, Childers BR (2011) CloudCache: expanding and shrinking private caches. In: Proceedings of HPCA, pp 219–230
20. Manikantan R, Kaushik R, Govindarajan R (2012) Probabilistic shared cache management (PriSM). In: ACM SIGARCH computer architecture news, vol. 40(3). IEEE Computer Society, New York
21. Qureshi MK (2009) Adaptive spill-receive for robust high-performance caching in CMPs. In: Proceedings of HPCA, pp 45–54
22. Qureshi MK, Patt YN (2006) Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches. In: Proceedings of MICRO, pp 423–432
23. Shedler GS, Slutz DR (1976) Derivation of miss ratios for merged access streams. IBM J Res Dev 20(5):505–517
24. Shen X, Zhong Y, Din C (2004) Locality phase prediction. In: Proceedings of ASPLOS, pp 165–176
25. Sherwood T, Perelman E, Hamerly G, Sair S, Calder B (2003) Discovering and exploiting program phases. IEEE Micro: top picks from computer architecture conference, pp 84–93
26. SPEC CPU2006. http://www.spec.org/cpu2006
27. Srikantaiah S, Kultursay E, Zhang T, Kandemir M, Irwin MJ, Xie Y (2011) MorphCache: a reconfigurable adaptive multi-level cache hierarchy for CMPs. In: Proceedings of HPCA, pp 231–242
28. Suh E, Rudolph L, Devadas S (2001) Dynamic cache partitioning for simultaneous multithreading systems. In: Proceedings of the IASTED international conference on parallel and distributed computing and systems, pp 116–127
29. Sundararajan KT, Jones TM, Topham NP (2013) RECAP: region-aware cache partitioning. In: IEEE 31st international conference on computer design, pp 294–301
30. Varadarajan K, Nandy SK, Sharda V, Bharadwa A, Iyer R, Makineni S, Newell D (2006) Molecular caches: a caching structure for dynamic creation of application-specific heterogeneous cache regions. In: Proceedings of MICRO, pp 433–442
31. Wang R, Hsieh M, Chen L (2014) Futility scaling: high-associativity cache partitioning. In: Proceedings of MICRO-47, pp 356–367