Analytical Modeling of Partially Shared Caches in Embedded CMPs

Wei Zang and Ann Gordon-Ross*
Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, 32611, USA
weizang@ufl.edu & ann@ece.ufl.edu
*Also with the NSF Center for High-Performance Reconfigurable Computing (CHREC) at the University of Florida

***Abstract*—In modern ubiquitous devices, optimizing shared last-level caches (LLCs) in embedded chip multi-processor systems (CMPs) is critical due to the increased contention for limited cache space from multiple cores. We propose cache partitioning with partial sharing (CaPPS) to reduce LLC contention and improve utilization. CaPPS can reduce the average LLC miss rate by 25% and 17% as compared to baseline configurations and private partitioning, respectively. To facilitate fast design space exploration, we develop an analytical model to quickly estimate the miss rates of all CaPPS configurations with an average error of only 0.73% and with an average speedup of 3,966X as compared to a cycle-accurate simulator.**

Keywords—cache partitioning; analytical modeling

I. Introduction

Many chip multi-processor systems (CMPs) leverage shared last-level caches (LLCs) (e.g., second-/third-level), such as the ARM Cortex-A, Intel Xeon, and Sun T2 [1][10][11]. To improve cache utilization, LLCs should be large enough to accommodate all sharing cores' data, but long access latencies and high power consumption typically precludes large LLCs from embedded systems with strict area/energy/power constraints. Since battery-operated devices (e.g., cell phones, tablets, laptops, etc.) have limited energy reserves and satisfying the applications' quality-of-services (QoSs) is typically required, optimizing small LLCs' performance 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 degrades performance. A core's LLC occupancy (utilized space) is flexible and dictated by the core's application's demands. Cores with high LLC requirements 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 unfairly evict the other cores' data, thus increasing LLC miss rates. For example, this unfair cache utilization is common in mobile systems when a local music/movie player and other web-service applications are co-executed.

To eliminate shared LLC contention, cache partitioning [5][15][18] partitions the cache, allocates *quotas* (a subset of partitions) to the cores, and optionally configures the partitions/quotas (e.g., size and/or associativity [15][18]) 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 [12]. 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 [15][18]. However, way partitioning for shared LLCs typically uses *private partitioning*, which restricts quotas for exclusive use by the allocated core only and can lead to poor cache utilization if a core does not occupy the core's entire allocated quota.

In this paper, we propose to improve way partitioning's cache utilization using cache partitioning with partial sharing (CaPPS). CaPPS improves cache utilization via *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. Whereas sharing configuration increases the design space and thus increases optimization potential, this large design space significantly increases design space exploration time. To facilitate design space exploration, we develop an offline analytical model to quickly estimate cache miss rates for all partitioning and sharing configurations, which enables determining LLC configurations for any optimization that evaluates cache miss rates (e.g., performance, energy, energy delay product, power, etc.). The analytical model probabilistically predicts the miss rates when multiple applications are co-executing using the *isolated cache access distribution* for each application (i.e., the application is run in isolation with no co-executing applications). Although several previous works [3][4][6] have developed analytical models to predict shared LLC contention offline, these works' caches where completely shared by all cores and did not consider partial sharing, which vastly increases the design space and thus optimization potential. Due to CaPPS's extensive design space, experiments reveal that CaPPS can reduce the average LLC miss rates by as much as 25% and 17% as compared to baseline configurations and private partitioning, respectively. The analytical model estimates cache miss rates with an average error of only 0.73% and is 3,966X faster on average than a cycle-accurate simulator.

II. Related Work

Since CaPPS uses way partitioning and we developed an analytical model to predict the shared ways' cache contention, we compare our work with prior work in these areas.

For way partitioning, Qureshi and Patt [15] developed utility-based cache partitioning (UCP) that 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. [18] partitioned the cache into small direct-mapped cache units, which were privately assigned to the cores and the cache partitions had configurable size, block size, and associativity. Kim et al. [13] developed static and dynamic cache partitioning for fairness optimization. Static cache partitioning used the cache access's stack distance profile to determine the cores' requirements. Dynamic cache partitioning increased/decreased the cores' quotas in accordance with the miss rate changes between evaluation intervals. Private LLCs also benefit from way partitioning. In CloudCache [14], the private caches were partitioned, but a core could share nearby cores' (limited access latencies) private caches. MorphCache [17] partitioned the level two and level three caches and allowed subsets of cores' private caches to be merged and fully shared by the subset. Although some of these prior works in private LLC partitioning [14][17] enabled a core to share other cores' quotas, CaPPS is more flexible than these works by enabling a *portion/all* of a core's quota to be shared with *any* subset of cores.

Prior works on analytical modeling to determine cache miss rates targeted only fully shared caches. Chandra et al. [3] proposed a model using access traces for isolated threads to predict inter-thread contention for a shared cache. 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. [6] 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 isolated reuse distance distribution of each application. Chen and Aamodt [4] proposed a Markov model to estimate the cache miss rates for multi-threaded applications with inter-thread communication.

Analytically predicting the cache miss rate for CaPPS is more challenging than prior works, since in CaPPS, only the interleaved LLC accesses of other cores that pollute the partially shared ways affect the core's miss rate. Determining the effects of these interleaved accesses on the miss rate introduces extensive complexity.

III. Cache Partitioning With Partial Sharing

To accommodate the LLC requirements for multiple applications co-executing on different cores, CaPPS partitions the shared LLC at the way granularity and leverages sharing configuration to allocate the partitions to

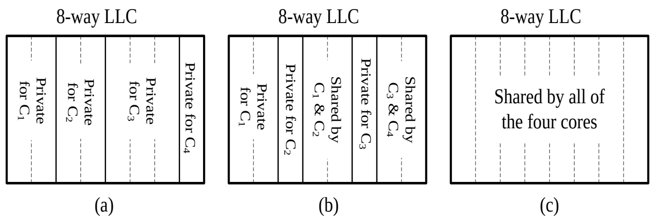

Figure 1. Three sample sharing configurations: (a) the cores' quotas are private; (b) some ways are partially shared with a subset of cores; and (c) the entire LLC is fully shared with all cores.

each core's quota. To facilitate fast design space exploration, an analytical model estimates the cache miss rates for the CaPPS configurations using the applications' isolated LLC access traces. We assume that the cores execute different applications in independent address spaces, thus there is no shared instruction/data address or coherence management, which is a common case in mobile systems running disparate applications and is similar to assumptions made in prior works [3][6].

A. *Architecture and Sharing Configurations*

CaPPS's sharing configurations enable a core's quota to be configured as private, partially shared with a subset of cores, or fully shared with all other cores. Fig. 1 (a)-(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) the cores' quotas are partially shared with subsets of cores; and (c) all of the four cores fully share all of the ways.

CaPPS uses the least recently used (LRU) replacement policy, but we note that the analytical model can be extended to approximate estimations for other replacement policies, such as pseudo-LRU. 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 ways are least likely to be accessed. For example, in Fig. 1 (b) 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. Maintaining this LRU ordering and determining replacement candidate can be easily implemented using a linked list or systolic array implementation [7] for conventional LRU caches with the integration of column caching [5] to achieve low hardware overhead and without increasing the cache access time. Since the hardware implementation is straightforward and is not the focus of this paper, we omit the implementation details for brevity.

B. *Analytical Modeling Overview*

For applications with fully/partially shared ways, the analytical model probabilistically determines the miss rates using the isolated cache access distributions for the co-executing applications. These distributions are recorded during *isolated access trace processing*. The isolated LLC access traces can be generated with a simulator/profiler by running each application in isolation on a single core with all other cores idle. 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.

Fig. 2 illustrates the contention 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's 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

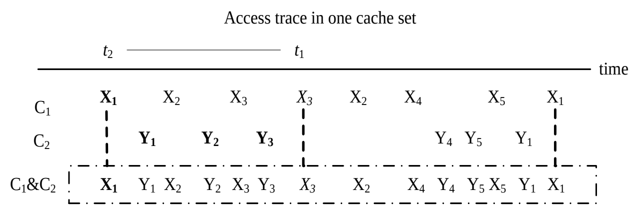
Access trace in one cache set
t_2
t_1
time
C_1
X_1 X_2 X_3 X_3 X_2 X_4 X_5 X_1
C_2
Y_1 Y_2 Y_3 Y_4 Y_5 Y_1
C_1&C_2
X_1 Y_1 X_2 Y_2 X_3 Y_3 X_3 X_2 X_4 Y_4 Y_5 X_5 Y_1 X_1

Figure 2. Two cores' isolated (C_1, C_2) and interleaved (C_1&C_2) access traces for an arbitrary cache set.

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 and t_2 dictates whether X_1 has been evicted from the cache or not. 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.

In order to determine the contention effects to C_1's miss rate, C_1 and C_2's number of accesses n_1 (Section III.D.a) and n_2 (Section III.D.b), respectively, during the time period (t_1, t_2) must be estimated. Since the number of blocks R from n_2 evicted into the shared ways dictates whether C_1's blocks (e.g., X_1 in Fig. 4) are still in the shared ways, we calculate the probability $p(n_2, R)$ that R number of blocks are evicted into the shared ways (Section III.D.c) to estimate C_1's miss rate (Section III.D.d).

C. Isolated Access Trace Processing

To accumulate the isolated cache access distribution, 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 [9]. 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. The stack distance d is the number of unique block addresses, or *conflicts*, in this set of accesses. For example, in Fig. 2, C_1'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 $p(d < d_i) = (\sum_{d=0}^{d=d_i-1} N_d)/(\sum N_d)$ and $p(d \geq d_i) = 1 - p(d < d_i)$, ($\forall d_i \in [1, A]$). 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. $Cycles_{base}$ assumes that all LLC accesses are hits. An application's total number of CPU cycles $Cycles_{exe}$ are recorded in the 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 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.

D. Analysis of the Shared Ways' Contention

First, we describe the analytical model to analyze the shared ways' contention for a sample CMP with two cores C_1 and C_2 and then generalize the analytical model to any number of cores. A sharing configuration allocates K_{C_1} number of ways to core C_1, where K_{P,C_1} ways are private and the remaining K_S ($K_S = K_{C_1} - K_{P,C_1}$) 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 and all accesses with $d \geq K_{C_1}$ are cache misses. The cache hit/miss determination of the accesses where $K_{P,C_1} \leq d \leq K_{C_1} - 1$ depends on the interleaved accesses from C_2, and 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 these estimations are not required since the number of misses for C_1 can be directly calculated using $\sum_{d=0}^{d=K_{C_1}-1} N_{d,C_1}$.

a. Calculation of n_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_1 for C_1's accesses with stack distance D based on $\bar{r}$.

Fig. 3 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 numbers of conflicts before and after X_3 are $(K_{P,C_1} - 1)$ and $(D - (K_{P,C_1} - 1))$, 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, which can be any integer in $[K_{P,C_1} - 1, \bar{r} - (D - K_{P,C_1}) - 2]$. After

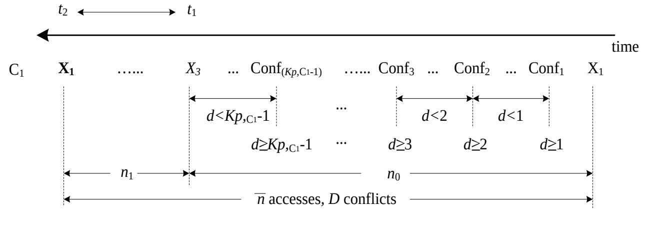
Access trace in one cache set
t_2
t_1
time
C_1
X_1 …… X_3 … $Conf_{(Kp,C_1-1)}$ …… $Conf_3$ … $Conf_2$ … $Conf_1$ X_1
$d<Kp,_{C_1}-1$
$d<2$
$d<1$
$d \geq Kp,_{C_1}-1$
$d \geq 3$
$d \geq 2$
$d \geq 1$
n_1
n_0
$\overline{n}$ accesses, D conflicts

Figure 3. C_1's isolated access trace to an arbitrary cache set for calculating $\boldsymbol{n_1}$.

determining the probability $p(n_0, (K_{P,C_1} - 1))$ 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 $\bar{n}_0 = \sum(n_0 \cdot p(n_0, (K_{P,C_1} - 1)))$, and n_1's expected value is: $\bar{n}_1 = \bar{r} - \bar{n}_0 - 1$.

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

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

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_1 accesses have exactly $(D - (K_{P,C_1} - 1))$ conflicts. $p_{before}(E_A)$ and $p_{after}(E_B)$ 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 $p_{total}(E_C)$ is the probability of E_C's occurrence, which is the summation of $(p_{after}(E_B) \cdot p_{before}(E_A))$ for all n_0. To calculate $p_{before}(E_A)$ and $p_{after}(E_B)$, we examine the sufficient conditions that E_A and E_B occur. In Fig. 3, the first access following X_1 must be different from X_1 (for $D > 0$), which is $Conf_1$ satisfying $d \geq 1$, since $Conf_1$ has at least one conflict: X_1. The second conflict $Conf_2$ satisfies $d \geq 2$, 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 $d \geq 3$ 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, $Conf_{Kp,C1-1}$ satisfies $d \geq (K_{P,C_1} - 1)$ and the accesses between X_3 and $Conf_{Kp,C1-1}$ satisfy $d < (K_{P,C_1} - 1)$. Thus, defining $\vec{a} = (a_1, a_2, \ldots, a_{K_{P,C_1}-1})$ where $a_i \in [0, n_0 - (K_{P,C_1} - 1)]$, $p_{before}(E_A)$ is:

$$p_{before}(E_A) = \left\{ \prod_{i=1}^{i=K_{P,C_1}-1} p(d \geq i) \right\} \cdot \left\{ \sum_{\forall \vec{a} \in S_a} \left(\prod_{i=1}^{i=K_{P,C_1}-1} p(d < i)^{a_i} \right) \right\} \quad (2)$$

where S_a is a set including all $\vec{a}$ satisfying $\sum a_i = n_0 - (K_{P,C_1} - 1)$. Similarly, defining $\vec{b} = (b_0, b_1, \ldots, b_{D-K_{P,C_1}})$ where $b_i \in [0, n_1 - (D - K_{P,C_1} + 1)]$, $p_{after}(E_B)$ is:

$$p_{after}(E_B) = \left\{ \prod_{i=0}^{i=D-K_{P,C_1}} p\left(d \geq i + K_{P,C_1}\right) \right\} \cdot \left\{ \sum_{\forall \vec{b} \in S_b} \left(\prod_{i=0}^{i=D-K_{P,C_1}} p\left(d < i + K_{P,C_1}\right)^{b_i} \right) \right\} \quad (3)$$

where S_b is a set including all $\vec{b}$ satisfying $\sum b_i = n_1 - (D - K_{P,C_1} + 1)$.

b. Calculation of n_2

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

$$\frac{\bar{n}_1}{\bar{n}_2} = \frac{\sum N_{d,C_1} / \widehat{Cycles}_{C_1}}{\sum N_{d,C_2} / \widehat{Cycles}_{C_2}} \quad (4)$$

where $\sum N_{d,C_1}$ and $\sum N_{d,C_2}$ are the total number of LLC accesses from C_1 and C_2, respectively. $\widehat{Cycles}_{C_1}$ is the number of CPU cycles required to execute the application on C_1 when C_2 is co-executing another application, and $\widehat{Cycles}_{C_2}$ is similarly defined. $\widehat{Cycles}_{C_i}$ can be calculated using $Cycles_{base}$ and the number of LLC misses $\hat{m}$ estimated with the contention:

$$\widehat{Cycles}_{C_i} = Cycles_{base} + \hat{m} \cdot LLC_{latency} + delay_{bus\_contention} \quad (5)$$

where $delay_{bus\_contention}$ 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. $delay_{bus\_contention}$ is derived by calculating the *bus contention probability* that another core is sending a read/write request to the LLC and the LLC is returning that core's requested block simultaneously with the evaluated core's bus request. The bus contention probability is dictated by each core's bus request probability, which is equal to the total number of bus requests generated from the core's higher level cache misses divided by $\widehat{Cycles}_{C_i}$.

c. Calculation of $p(n_2, R)$

$p(n_2, R)$ is the probability that R number of blocks are evicted from C_2's private ways in the n_2 accesses. Directly using the expected n_2 to calculate $p(\bar{n}_2, R)$ will introduce a large bias (approximate 10% error) in the estimated LLC miss rate, since different values of n_2 result in different hit/miss determinations and using one expected value $\bar{n}_2$ will estimate all n_2 as hits/misses. Thus, we model n_2 using a Poisson distribution $p(n_2) = Poisson(n_2, \lambda)$, where λ is $\bar{n}_2$ if the LLC is accessed randomly. However, since the LLC's accesses are generally not random and not uniformly distributed in time (which makes (4) valid), we use an empirical variable e to adjust λ to $\lambda = \bar{n}_2 / e$. 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. Since the range of n_2 is infinite in the Poisson distribution, and n_2 with very small $p(n_2)$ has minimal effect on the miss rate estimation, we only consider the n_2 with $p(n_2) > 0.01$ and calculate the associated $p(n_2, R)$.

To calculate $p(n_2, R)$ for an arbitrary n_2, R is determined by evaluating the n_2 accesses in chronological order with an initial value of $R = 0$. If there is one access with $d > K_{P,C_2} + current\ R$, fetching this address into C_2's private ways will evict one block into the shared ways and thus R is incremented by 1. Therefore, we can calculate $p(n_2, R)$ inductively:

$$p(n_2, R) = \begin{cases} p(n_2 - 1, R - 1) \cdot p\left(d \geq K_{P,C_2} + (R - 1)\right), & R = n_2 \\ p(n_2 - 1, R) \cdot p\left(d < K_{P,C_2} + R\right) \\ \quad + p(n_2 - 1, R - 1) \cdot p\left(d \geq K_{P,C_2} + (R - 1)\right), & R < n_2 \\ p(n_2 - 1, R) \cdot p\left(d < K_{P,C_2} + R\right), & R = 0 \end{cases}$$

with the initial case $p(n_2 = 0, R = 0) = 1$.

d. Calculation of the LLC Miss Rates

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

$$\widehat{h_{\mathrm{C}_1}} = \sum_{d=0}^{d=K_{p,\mathrm{C}_1}-1} N_{d,\mathrm{C}_1} + \sum_{d=K_{p,\mathrm{C}_1}}^{d=K_{\mathrm{C}_1}-1} \left(N_{d,\mathrm{C}_1} \cdot \sum_{\forall n_2:\, p(n_2)>0.01} \left(\left(\sum_{R=0}^{R=K_{\mathrm{C}_1}-d-1} p(n_2,R) \right) \cdot p(n_2) \right) \right) \quad (7)$$

After accumulating $\widehat{h_{\mathrm{C}_1}}$ for all cache sets, the number of LLC misses $\widehat{m_{\mathrm{C}_1}}$ 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 $\bar{n}_{\mathrm{C}_j}$ from the additional cores during the time (t_1, t_2) and then estimating $p(n_{\mathrm{C}_j}, R_{\mathrm{C}_j})$ similarly as estimating $\bar{n}_2$ and $p(n_2, R)$ for C_2. The generalized expression of (6) is:

$$\widehat{h_{\mathrm{C}_i}} = \sum_{d=0}^{d=K_{p,\mathrm{C}_i}-1} N_{d,\mathrm{C}_i} + \sum_{d=K_{p,\mathrm{C}_i}}^{d=K_{\mathrm{C}_i}-1} \left(N_{d,\mathrm{C}_i} \cdot ph \right) \quad (8)$$

where:

$$ph = \sum_{\forall \vec{\mathrm{C}} \in S_{\mathrm{C}}} \left(\prod_{\mathrm{C}_j \in \vec{\mathrm{C}}} \left(p(n_{\mathrm{C}_j}) \cdot p(n_{\mathrm{C}_j}, R_{\mathrm{C}_j}) \right) \right) \quad (9)$$

where $\vec{\mathrm{C}} = (n_{\mathrm{C}_1}, n_{\mathrm{C}_2}, \dots, n_j)$ with $p(n_{\mathrm{C}_j}) > 0.01$ and S_{C} is a set including all $\vec{\mathrm{C}}$ satisfying $\sum R_{\mathrm{C}_j} \leq K_{\mathrm{C}_i} - d - 1$.

According to (5), 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 for $\hat{m}$. 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 (5) 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 evaluated sharing configuration and the isolated cache access distribution for each application. Due to the large number of complex and interdependent variables and unknowns, the complexity of the model is intractable, thus in our experiments, we evaluate the analytical model's measured execution time.

TABLE I. CMP SYSTEM 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

IV. EXPERIMENT RESULTS

We verified the advantages of CaPPS as compared to two baseline configurations and private partitioning. We also verified the accuracy of our estimated LLC miss rates obtained via 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 illustrate the analytical model's efficiency by comparing the time required to calculate the LLC miss rates as compared to using a cycle-accurate simulator that generates the exact cache miss rates for all configurations.

A. *Experiment Setup*

We used twelve benchmarks from the SPEC CPU2006 suite [16], which were compiled to Alpha_OSF binaries and executed using "ref" input data sets. Due to incorrect execution, we could not evaluate the complete suite. Even though our work is targeted towards 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 performed phase classification using SimPoint [8] to select 500 million consecutive instructions with similar behavior as the *simulation interval* to mimic an embedded application with high LLC occupancy.

We generated the exact cache miss rates for comparison purposes using gem5 [2] 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 CaPPS. TABLE I shows the parameters used for each system component. Since four cores shared the eight-way LLC (i.e., L2 cache), CaPPS's design space had 3,347 configurations.

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. The full-system checkpoint was created by aggregating the *isolated-benchmark checkpoints*, which were generated by fast-forwarding the benchmark to the starting instruction of the benchmark's associated simulation interval when the benchmark was executed in isolation.

For each simulation, 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 rather than the absolute number of cache misses, the incomplete benchmarks' execution had no impact on the 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 hierarchical level of the LLC, or the cache parameters (e.g., total size, block size, and associativity for our experiments).

B. CaPPS Evaluation

To validate the advantages of CaPPS, we compared CaPPS's ability to reduce the LLC miss rate as compared to two baseline configurations and private partitioning, since shared LLC partitioning in previous works [13][15][18] only provided private partitioning.

Fig. 4 depicts the average LLC miss rate reductions for CaPPS's optimal configurations (the configurations with minimum average LLC miss rate in CaPPS's design space) as compared to two baseline configurations: 1) *even-private-partitioning*: the LLC is evenly partitioned using private partitioning (first bar); and 2) *fully-shared*: the LLC is fully shared by all cores (second bar). Across all benchmark sets, the average and maximum average LLC miss rate reductions 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*.

The third bar in Fig. 4 depicts the average LLC miss rate reductions for CaPPS's optimal configuration as compared to private partitioning's optimal configuration, 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 all benchmark sets, the average and maximum reductions in

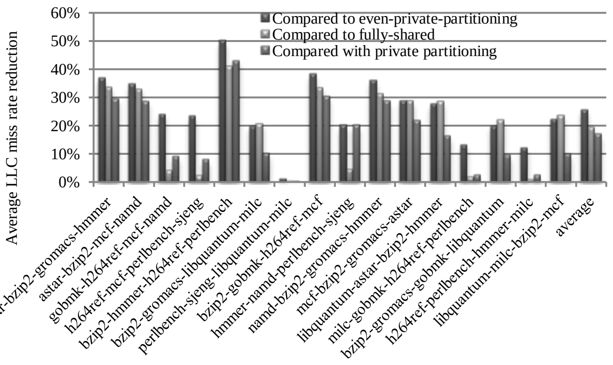

Figure 4. Average LLC miss rate reductions for CaPPS's optimal configurations compared to *even-private-partitioning*, *fully-shared*, and private partitioning.

CaPPS's average LLC miss rates as compared to private partitioning were 16.92% and 43.02%, respectively.

C. Analytical Model's 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 3,347 configurations. Fig. 5 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 deviations. Averaged over all sixteen benchmark sets, the average miss rate error and standard deviation are -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 was negligible, 1.3% and 0.36%, respectively.

D. Analytical Model's 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 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 gigabytes of RAM. Fig. 6 depicts the speedup of the analytical model for each benchmark set as compared to gem5. Over all benchmark sets, the average speedup is 3,966X, with maximum and minimum speedups of 13,554X and 1,277X, respectively. For one benchmark set, the time for simulating all 3,347 configurations using gem5 was approximately three months, and comparatively, the analytical model took only

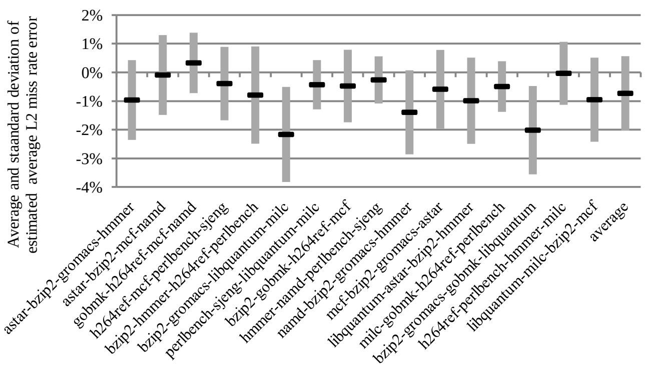

Figure 5. The average and standard deviation of the average LLC miss rate error determined by the analytical model.

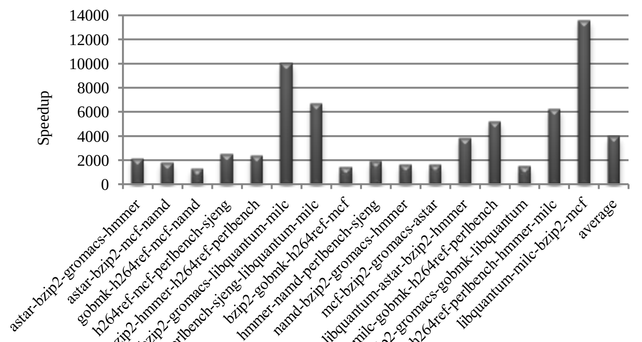

Figure 6. The analytical model's simulation time speedup compared to gem5.

two to three hours.

V. Conclusions and Future Work

In this paper, we presented cache partitioning with partial sharing (CaPPS)—a novel cache partitioning and sharing architecture that improves shared last-level cache (LLC) performance with low hardware overhead for chip multi-processor systems (CMPs). Since CaPPS affords an extensive design space for increased optimization potential, CaPPS can reduce the average LLC miss rate by as much as 25% and 17% as compared to baseline configurations and private partitioning, respectively. 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 (since no heuristics exist) of the CaPPS design space to determine the lowest energy cache configuration, the analytical model affords an average speedup of 3,966X. Finally, CaPPS and the analytical model are applicable to CMPs with any number of cores and place no limitations on the cache parameters.

Future work includes extending the analytical model to optimize for any design goal, such as performance or energy delay product, leveraging the offline analytical results to guide online scheduling for performance optimizations in real-time embedded systems, including accesses to shared address space, incorporating cache prefetching in our analytical model, and extending CaPPS to proximity-aware cache partitioning for caches with non-uniform accesses.

Acknowledgements

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] N. Binkert, et. al. The gem5 Simulator, http://gem5.org [retrieved: Feb., 2013].

[3] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture", In Proceedings of HPCA, Feb. 2005, pp. 340-351.

[4] X. E. Chen and T. M. Aamodt, "A first-order fine-grained multithreaded throughput model", In Proceedings of HPCA, Feb. 2009, pp. 329-340.

[5] D. Chiou, D. Chiouy, L. Rudolph, S. Devadas, and B. S. Ang, "Dynamic Cache Partitioning via Columnization", Computation Structures Group Memo 430. M.I.T. 2000.

[6] D. Eklov, D. Black-schaffer, and E. Hagersten, "Fast Modeling of Shared Cache in Multicore Systems", In Proceedings of HiPEAC, Jan. 2011, pp. 147-157.

[7] J. P. Grossman, "A Systolic Array for Implementing LRU Replacement," Project Aries Technical Memo ARIES-TM-18, AI Lab, M.I.T., Cambridge, MA, 2002.

[8] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "SimPoint 3.0: Faster and More Flexible Program Analysis", Journal of Instruction-level Parallelism, 2005, pp. 1-28.

[9] M. D. Hill and A. J. Smith, "Evaluating Associativity in CPU Caches," IEEE Trans. on Computers, Vol. 38, No. 12, 1989, pp. 1612-1630.

[10] Intel Core Duo Processor, http://ark.intel.com/products/family/22731.

[11] K. Johnson and M. Rathbone, "Sun's Niagara Processor", NYU Multicore Programming, 2010.

[12] R. E. Kessler and M. D. Hill, "Page Placement Algorithms for Large Real-indexed Caches", ACM Trans. on Computer Systems, Vol. 10, No. 4, 1992, pp. 338-359.

[13] S. Kim, D. Chandra, and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture", In Proceedings of PACT, Sep.-Oct. 2004, pp. 111-122.

[14] H. Lee, S. Cho, and B. Childers, "CloudCache: Expanding and Shrinking Private Caches", In Proceedings of HPCA, Feb. 2011, pp. 219-230.

[15] M. Qureshi and Y. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches", In Proceedings of MICRO, Nov. 2006, pp. 423-432.

[16] SPEC CPU2006. http://www.spec.org/cpu2006 [retrieved: Sep., 2011].

[17] S. Srikantaiah, E., T. Zhang, M. Kandemir, M. Irwin, and Y. Xie, "MorphCache: a Reconfigurable Adaptive Multi-level Cache Hierarchy for CMPs", In Proceedings of HPCA, Feb. 2011, pp. 231-242.

[18] K. Varadarajan, et al., "Molecular Caches: A Caching Structure for Dynamic Creation of Application-specific Heterogeneous Cache Regions", In Proceedings of MICRO, Nov. 2006, pp. 433-442.