Probabilistic Shared Cache Management (PriSM)

R Manikantan[1] Kaushik Rajan[3] R Govindarajan[1,2]

[1]Department of Computer Science and Automation, Indian Institute of Science, Bangalore
[2]Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore
[3]Microsoft Research India, Bangalore

Abstract

Effective sharing of the last level cache has a significant influence on the overall performance of a multicore system. We observe that existing solutions control cache occupancy at a coarser granularity, do not scale well to large core counts and in some cases lack the flexibility to support a variety of performance goals.

In this paper, we propose Probabilistic Shared Cache Management (PriSM), a framework to manage the cache occupancy of different cores at cache block granularity by controlling their eviction probabilities. The proposed framework requires only simple hardware changes to implement, can scale to larger core count and is flexible enough to support a variety of performance goals. We demonstrate the flexibility of PriSM, by computing the eviction probabilities needed to achieve goals like hit-maximization, fairness and QOS.

PriSM-HitMax improves performance by 18.7% over LRU and 11.8% over previously proposed schemes in a sixteen core machine. PriSM-Fairness improves fairness over existing solutions by 23.3% along with a performance improvement of 19.0%. PriSM-QOS successfully achieves the desired QOS targets.

1. Introduction

Efficient management of shared last level caches to achieve various performance related goals has received significant research attention [2, 5–10, 14, 15, 17, 19, 20]. Multiple cores can be allowed to share a cache by allocating each core a portion of the cache space. This partitioning can be done either at the coarser granularity of cache ways, as done in [9, 14, 15, 18], or at the finer granularity of cache blocks, as done in Vantage [17]. *Way-partitioning is popular because of its simplicity of design*. It allows for different performance goals like hit maximization [14] and fair sharing [9] to be enforced without introducing much additional hardware complexity. However way-partitioning can be inefficient as it only allows partition sizes to grow or shrink by a fixed large size(inversely proportional to the associativity) while it is possible that the optimal size for a partition falls in between. As the number of cores increases and becomes comparable to the number of ways, such scenarios are likely to occur more frequently. We observe that while way-partitioning works very well at low core counts, it is unable to deliver the same level of performance at higher core counts. Therefore *from a performance point of view partitioning of cache at finer granularity such as block level is more desirable* [17]. However achieving such a partitioning of the *whole cache* while also managing the partitions *without introducing additional hardware complexity* remains a challenge to-date. Vantage [17] is the only known scheme that proposes a fine grained partitioning framework. However it achieves this only for a portion and not all of the cache and does so with significant changes to the hardware.

In this paper, we propose *Probabilistic Shared-Cache Management (PriSM)*, a framework that achieves the scalable performance of partitioning the *entire cache at a fine granularity* while retaining the *simple hardware design* of way-partitioning. The proposed framework consists of two components – (i) A simple hardware component which we call the probabilistic cache manager to control the operation of the cache and (ii) an allocation policy [17], which is a mechanism to translate high level performance goals into an eviction probability distribution. Using a simple analytical model we show that eviction probabilities leads to fine-grained partitioning capability. The probabilistic cache manager of PriSM uses the same two step replacement mechanism as way-partitioning (i) identify a victim-core and (ii) leverage the baseline cache replacement mechanism to identify a victim line belonging to the core identified in the first step. Unlike way-partitioning where the victim core is identified based on the number of ways it occupies in the set, PriSM generates a victim-core id in line with the eviction probability distribution computed by the allocation policy. Thus PriSM achieves fine-grained partitioning at a hardware cost comparable to that of way-

978-1-4673-0476-4/12/$31.00(c)2012 IEEE

partitioning. Another salient feature of PriSM is that unlike way-partitioning, PriSM allows each cache set to make its own decision and controls the overall cache occupancy of each core at the whole cache level. As compared to Vantage, PriSM can partition the entire cache in a fine grained manner and achieves it with simple hardware.

We demonstrate the versatility of our framework by building a hit maximization scheme(PriSM-H), a fair cache sharing scheme(PriSM-F) and a QoS scheme(PriSM-Q) on top of PriSM. The key performance benefits achieved by the PriSM framework include

- PriSM performs 7.8% - 11.8% better than Vantage across a wide variety of cache configurations.
- PriSM-H outperforms LRU at various core counts. It achieves an improvement in performance of 18.7% over LRU in a sixteen core machine. It also consistently achieves better performance than current best performing solutions like UCP [14] and PIPP [20] from 4—32 cores.
- PriSM-F improves the fairness by 23.3% over and above an existing solution that achieves fairness by managing cache using way-partitioning [9]. This improved fairness is achieved along with a 4.8% improvement in performance over the existing solution.
- PriSM-Q successfully achieves the QOS targets in 38 out of 41 workloads tried out.
- We show that the proposed cache management scheme can work with other replacement schemes by studying its performance with DIP [13] as the replacement policy.

2. Motivation

In this section we motivate the need for a fine-grained and scalable partitioning mechanism by demonstrating the (i) lack of scalability exhibited by current solutions (way-partitioning in particular) for higher core counts and (ii) the performance benefits provided by partitioning at a finer granularity.

Scalability. Figure 1(a) shows the performance of hit-maximization schemes UCP [14], PIPP [20] and the fairness provided by a way-partitioning based fairness mechanism [9] for different machine configurations. The core count varies from 4 to 32 (16 for fairness). The performance of PIPP and UCP is shown in terms of Average Normalized Turnaround Time (ANTT) [3] (Refer to Section 5 for simulation methodology and performance metrics). It is a lower-is-better metric. The performance is shown normalized to that of the corresponding LRU and is averaged (geomean) across all the workloads. With larger core counts the performance benefits provided over LRU by UCP and PIPP reduces. PIPP for instance performs worse than LRU at 32-core configuration. A similar behaviour is observed in the case of fairness (fairness is a higher-is-better metric), where going from 4 to 8 and then 16 cores reduces the overall fairness by a significant margin.

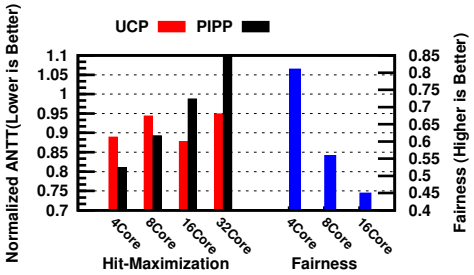
Fine-grained partitioning. Way-partitioning increases the allocation at a coarser granularity of one way in all the cache sets. For instance, in a 16 way cache allocating one more way translates to providing 6.25% more cache space. To demonstrate the benefits to be had by fine-grained partitioning, we study the performance of UCP [14] in 4MB caches with 16/64/256 way associativity. In the case of the 64 and 256 way associative caches allocating one more way translates to increasing the occupancy by 1.6% and 0.39% respectively. As the cache size remains unmodified, the high associative configurations help us mimic the impact of partitioning at finer granularities. And increasing associativity ensures that an allocation policy like UCP is aware of the granularity of partition and hence can be applied as is without any modifications. A set of quad and eight core workloads is used in this study.

Figure 1(b) summarizes the performance in terms of IPC-Throughput (sum of IPC) [3] for LRU and UCP in the three cache configurations. It can be seen that increasing associativity and the resultant finer-grained control over cache occupancy helps improve the performance of UCP. It is interesting to note that the magnitude of gain observed in UCP as we move from 16 to 64 to 256 way associativity is higher than the corresponding gain seen by LRU. This shows that performance of existing allocation policies could be improved by fine-grained partitioning control. But building caches of very high associativity entails additional hardware and different cache organizations like Zcache [16]. Hence it is required to enable partitioning at granularities finer than that of associativity.

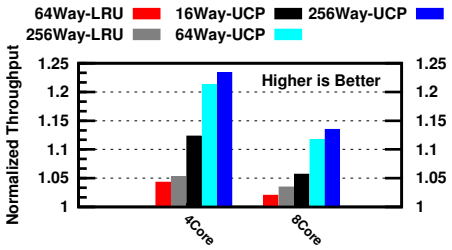
In this work, we propose PriSM a probabilistic framework that meets both the above requirements. By moving from a way-partitioning based scheme to a probabilistic scheme PriSM provides the ability to reduce/increase the space allocated in steps of $1/N$, where N is the number of cache blocks. Second, unlike way-partitioning where each set of the cache is partitioned in the same manner, PriSM introduces the flexibility of allowing each cache set to make its own decisions and yet be able to control the overall occupancy at the shared cache level.

3. The Probabilistic Shared Cache Management Framework

This section describes the proposed probabilistic shared cache management (PriSM) framework. PriSM associates an eviction probability with each context that shares the cache. PriSM consists of three components that together



(a) Scalability



(b) Finer-Grained Partitioning

Figure 1. Motivation – (a) Performance of UCP, PIPP and Fairness [9] schemes with increasing core count and (b) Benefits of fine-grained partitioning with UCP.

Table 1. Terms used in the paper

N – Number of Cache blocks
W – Interval length. Measured in terms of number of cache misses.
C_i – Fraction of cache space occupied by $Core_i$ at the beginning of interval. $0 \leq C_i \leq 1$.
M_i – Fraction of Misses caused by $Core_i$ at the beginning of interval. $0 \leq M_i \leq 1$.
T_i – Desired cache occupancy at the end of interval for $Core_i$. $0 \leq T_i \leq 1$.
τ_i – Achieved target cache occupancy of $Core_i$ at the end of interval. $0 \leq \tau_i \leq 1$.
E_i – Eviction probability for $Core_i$. $0 \leq E_i \leq 1$.

enable it to achieve high level performance goals (i) a simple cache hardware that uses the eviction probabilities during a replacement to identify a victim block; (ii) an analytical model that shows how eviction probabilities actually provide fine-grained partitioning capability and (iii) an allocation policy that takes a high level performance goal, computes the desired cache occupancy to meet the goal and translates it into eviction probabilities using the analytical model.

Some of the common terms used throughout this section are defined below (also listed in Table 1). W represents an interval of execution, measured in terms of misses in the shared cache. The allocation policy recomputes the cache space allocated to each program at the end of every interval. N is the number of cache blocks. C_i is the cache occupancy of $core_i$ and is defined as the ratio of the cache blocks occupied by $core_i$ to the total number of blocks in the cache across all sets. T_i is the target occupancy for $core_i$ and represents the desired cache occupancy to meet a certain performance/fairness goal (details in Section 3.3). τ_i is the occupancy likely to be achieved at the end of an interval. M_i represents the fraction of misses contributed by $core_i$ to the total misses in an interval. E_i is the probability of choosing a block belonging to $core_i$ for eviction.

We use the term core interchangeably with program, as is done in all the earlier works [6,7,9,14,15,18,20]. Further we assume that the number of programs (used during evaluation) is same as the number of cores and there is no program migration. In practice all the solutions in this space require the partitioning information to be associated with the program (identified using a unique-ID). Also all cache accesses need to be tagged with the unique-ID of the program causing the access so that the cache-controller is able to keep track of occupancy. Note that these requirements are common to all the cache partitioning/management schemes. In short, in the rest of the paper, when we say occupancy or eviction probability of $core_i$, we actually mean the occupancy or eviction probability associated with the program running in $core_i$.

3.1. Cache operations under PriSM

The operation of a cache under PriSM is governed by a discrete probability distribution $\mathcal{E}$ that associates with each $Core_i$ an eviction probability E_i such that $\Sigma(E_i) = 1$. Eviction probabilities are recomputed for all cores at the end of each interval by the allocation policy. In PriSM, hits behave the same way as in a traditional cache. On a miss there are two steps required to identify the victim block:

- *Core-Selection*: A core ID is generated in accordance with $\mathcal{E}$, the eviction probability distribution. Note that it is possible for a core to have an eviction probability of zero, which means it will never be selected for eviction by Core-Selection step.
- *Victim-Identification*: A victim block belonging to the core selected in the *Core-Selection* step is identified using the underlying cache replacement policy.

In rare cases it is possible that the selected core does not have any block in the given cache set. Under such a scenario, we use the underlying replacement policy to select the first replacement candidate that belongs to a core with non-zero eviction probability. Decoupling core selection from victim identification brings in the advantage that PriSM can be used with any underlying replacement policy.

3.2. Controlling cache occupancy via eviction probabilities

Next we describe how eviction probabilities influence the fraction of a cache occupied by a given core. This allows us to derive eviction probabilities to achieve the desired target cache occupancies.

Influence on cache occupancy over a single interval. In any interval (of W misses) under consideration, let us assume that $core_i$ accounts for a fraction M_i of the total misses. Thus during the interval, $core_i$ contributes $M_i \times W$ misses. If we had not evicted any cache lines belonging to $core_i$ from any of the cache sets, then its cache occupancy would have increased from C_i at the beginning of the interval to $(C_i + (M_i \times W/N))$ at the end of the interval. However, with eviction probability $E_i > 0$, $(E_i \times W)$ lines would have been evicted over the interval. This amounts to a reduction in cache occupancy by $E_i \times W/N$ and the cache occupancy of $core_i$ at the end of the interval (a value in $[0, 1]$) becomes $\tau_i = C_i + ((M_i - E_i) \times W/N)$. The occupancy achieved at the end of an interval, τ_i, can take any value between $[0, 1]$. As the smallest allocation unit is a cache block, in practice, τ_i can take values in $[0, 1]$ in steps of 1/N. Compared to this, all way-partitioning schemes are restricted to choose the target cache occupancy at the end of an interval to be one among A discrete values where A is the associativity of the cache. In a 4MB32Way cache with block size of 64B, N=65536 and A=32. As N is much larger than A, PriSM allows fine-grained partitioning of the shared cache.

Deriving eviction probabilities from target occupancy. Let T_i denote the desired target occupancy for $core_i$ to achieve a given performance goal. From the above calculation we know that $\tau_i = (C_i + ((M_i - E_i) \times W/N))$ is the occupancy achieved at the end of an interval. Now our goal is to determine E_i so that T_i can be reached as quickly as possible. However, given specific values of C_i, M_i, N and W it may not be possible to achieve T_i in a single interval (for example, when $T_i > C_i + (M_i \times W/N)$). Hence multiple intervals might be required for acheiving T_i. This brings about the complication of not knowing both τ_i and E_i. We deal with this situation as follows. When T_i cannot be reached from C_i in the current interval E_i should be either 0 or 1 depending on whether $(C_i < T_i)$ or $(C_i > T_i)$. Otherwise, E_i can be computed by solving the equation $T_i = \tau_i = C_i + ((M_i - E_i) \times W/N$. In summary we calculate E_i in every interval using the following equation.

$$E_i = \begin{cases} 0, & \text{if } ((C_i - T_i) \times N/W + M_i) < 0, \\ 1, & \text{if } ((C_i - T_i) \times N/W + M_i) > 1, \\ (C_i - T_i) \times N/W + M_i, & \text{otherwise.} \end{cases} \quad (1)$$

To compute E_i, we require C_i and M_i in addition to T_i. C_i can be obtained easily by having a single counter per core to provide the number of cache blocks (integral value) occupied by each core. We use the value of M_i from the previous interval which could either be obtained from performance counters or through a set of hardware counters, one for each core. As mentioned earlier T_i is computed by the allocation policy. In fact in our proposed approach, we compute both T_i and E_i in the allocation policy. N is typically a power of two. As long W is a power of two, E_i can be computed using addition and shift operations on integers. In the following subsection, We discuss a few simple allocation policies that translate a variety of common high-level objectives into target-cache occupancy T_i and subsequently to eviction probability E_i.

3.3. Enforcing performance goals via PriSM

It is the job of an allocation policy [17] to convert a higher level performance goal into a partitioning of the cache. The allocation policy could be either implemented in hardware [9, 14] or software [15, 18]. In this work we envision the allocation policy to be implemented in software as it provides the flexibility to implement and choose from a variety of performance goals and the ability to incorporate newer and better versions of the allocation policy. It works by reading an augmented set of performance counters, computes the eviction probability distribution and communicates it to the cache controller. The performance counters used are either already present in modern processors or proposed in earlier works [4](The only extension to the set of performance counters required will be ones to provide statistics regarding hits and misses experienced in shadow tags by each core). The software scheme requires ISA support to communicate the computed partition information (eviction probabilities) to the hardware [15]. As we show later in Section 5, it is enough to use 6 bits to capture this probability information.

As most allocation policies are tuned to work with way-partitioning, they are not able to take advantage of the fine-grained partitioning capability without any modifications. Hence to demonstrate the versatility of our framework, we propose allocation policies that are aware of the ability of PriSM to support fine-grained partitioning and can use it constructively.

Hit Maximization. The allocation policy to achieve hit-maximization is shown in Algorithm 1. The algorithm tries to provide more cache space to the core that has the maximum potential to gain hits. It computes the target occupancy by scaling the cache occupancy based on how well it is likely to perform if the whole cache were assigned to it. We assume the presence of shadow tags [14] to estimate

N: Number of cores;
$TotalGain = 0$;
for $core \leftarrow 1$ **to** N **do**
  $PotentialGain[core] =$
  $StandAloneHits[core] - SharedHits[core]$;
  $TotalGain+ = PotentialGain[core]$;
end
// Core with a higher potential to gain hits gets more cache space.
for $core \leftarrow 1$ **to** N **do**
  $T_{core} =$
  $C_{core} \times (1 + (PotentialGain[core] \div TotalGain))$;
end
Normalize the target cache occupancy T_{core}.
$T_{core} = T_{core} \div \sum T_{core}$;
Compute eviction probability E_i using Equation 1 ;

Algorithm 1: Hit-Maximization

stand-alone hit values. For hardware implementations, it is possible to round off $TotalGain$ to the nearest power of 2 and convert the division to a shift operation. The total computations(arithmetic operations) performed by this algorithm ranges from 20 for 4-cores to 160 for 32-cores.

Fairness. Fairness is defined as all the cores suffering equal slowdown in terms of execution time compared to the stand-alone case [3]. Our proposed allocation policy to achieve fairness is shown in Algorithm 2. To measure the slowdown due to sharing, it is essential to know the performance when the program runs alone(CPI$_{standAlone}$) and when it runs with other programs as part of a workload (CPI$_{shared}$). CPI$_{shared}$ is the actual performance observed when the workload is running and could be obtained by reading the apporpriate performance counter. So the goal is to estimate CPI$_{standAlone}$.

Our algorithm works by treating the program performance(expressed in terms of Cycles Per Instruction, CPI) as having two components: (i) CPI$_{ideal}$ – the performance if all accesses are to hit in the LLC and (ii)CPI$_{llc}$ – the impact of LLC on performance. $CPI = CPI_{ideal} + CPI_{llc}$. Similar formulations have been used in earlier works too [4]. The difference between a program running alone and with other programs will reflect in the CPI_{llc} component.

In modern processors, performance counters provide the number of cycles commit was stalled due to a long latency load [4]. We use this value as an estimate for CPI_{llc} for a shared cache. Now CPI_{ideal} can be computed as $(CPI_{shared} - CPI_{llc})$. If we can estimate the contribution of LLC to CPI when a program is run alone($CPI_{llc}^{standAlone}$), then we can estimate its performance when run alone($CPI_{standAlone}$) as $((CPI_{shared} - CPI_{llc}) + CPI_{llc}^{standAlone})$. Shadow tags [14] can provide us with an estimate of increase in hits when we go from shared cache scenario to stand-alone scenario for each program. We use the estimate of benefits provided by shadow tags to scale the CPI_{llc} value linearly to arrive at $CPI_{llc}^{standAlone}$. The space allocated to each core is increased proportional to the slowdown experienced by it. We evaluated this algorithm in a quad-core machine and found that the predicted $CPI_{standAlone}$ was close to the actual performance for most of the benchmarks. Due to lack of space we do not show those results. For each core, the algorithm requires us to read the performance counters for CPI, instructions committed, cycles, shared and stand alone misses in the shadow tags and number of shadow tag accesses. Other than the shadow tag metrics, the others are standard performance counters. The number of computations required range between 28 for 4 cores to 224 for 32-cores.

N:Number of Cores;
$IntervalLength$:Length of the most recent interval in cycles;
$LLCStallCycles[1, ..., N]$: Number of cycles spent waiting for a long latency load to commit;
for $core \in 1toN$ **do**
  StandAloneLLCStallCycles[$core$] =
  LLCStallCycles[$core$] $\times$ (1 -
  (StandAloneHitsGained[$core$] $\div$ SharedMiss[$core$]));
  StandAloneCycles[$core$] = (IntervalLength -
  LLCStallCycles[$core$]) +
  StandAloneLLCStallCycles[$core$];
  Slowdown[$core$] = IntervalLength $\div$
  StandAloneCycles[$core$];
  $T_{core} = C_{core} \times Slowdown[core]$;
end
Normalize T_i. $T_i = T_i \div \sum T_i$;
Translate T_i to E_i using Equation 1;

Algorithm 2: Fairness

Quality of Service. Quality of Service (QOS) ensures specified minimum levels of performance for a given program [6]. We use maximum slowdown in Instructions Per Cycle (IPC) [18] to specify QOS targets. We specify QOS target for one of the cores and try to achieve hit-maximization for the remaining cores. The allocation policy to achieve QOS is shown in Algorithm 3. Without loss of generality, we show the algorithm to achieve a specified performance target for Core$_0$. The algorithm increases, decreases or maintains the cache occupancy(C_0) of Core$_0$ based on its performance relative to the specified target. The scaling is done according to the parameters α and β and we use the value 0.1 for both. Once the target occupancy T_0 for Core$_0$ is computed, we attempt to maximize the hits provided by the other cores that will share the remaining (1 - T_0) of cache space. This algorithm requires IPC which can be obtained from standard performance counters [4].

Input: N: Number of Cores
Input: $Target_{IPC}$: Minimum IPC to achieve for $Core_0$
Input: $Current_{IPC}[0]$: The current performance levels of core$_0$
$T_0 = C_0$;
if $Current_{IPC}[0] < Target_{IPC}$ **then**
    // Increase the space allocated as performance is below par.
    $T_0 = (1 + \alpha) \times C_0$;
end
else if $Current_{IPC}[0] > Target_{IPC}$ **then**
    /* Decrease the space allocated */
    $T_0 = (1 - \beta) \times C_0$;
end
/* Allocation is not changed if the performance target is being met */
Do hit maximization for cores 1, ..., N for using cache space $(1 - T_0)$;
Convert T_i to E_i using Equation 1;

Algorithm 3: QOS

3.4. Discussion

Below we highlight the key features of the proposed PriSM framework and where relevant we report how it is different from existing schemes :

- PriSM provides an ability to manage the shared cache occupancy in a fine-grained fashion. Both the desired target occupancy and achieved target occupancy for an interval can take any value between $[0, 1]$. Vantage [17] is the only other framework that supports fine grained cache management. But Vantage requires significant changes to the hardware to enable fine-grained partitioning.
- PriSM can be used with any replacement scheme. Solutions like PIPP [20] and TA-DIP [7] tightly integrate performance goals with cache management and hence do not have this flexibility. Interestingly UCP [14] is also restrictive as it can only work with replacement policies that exhibit the stack property.
- The actual hardware changes required to the cache controller are (i) space to store the eviction probabilities for each core, (ii) extend the replacement policy to include a core-selection step and (iii) a random number generator to be used by the core-selection step.

4. Simulation methodology

We use the M5 simulator [1] to evaluate our proposed framework. The simulator is extended to support a detailed DRAM memory model and to support multiple memory controllers. On top of this we implemented PriSM, UCP, PIPP, Vantage and other schemes studied in this paper. We simulate 4, 8, 16 and 32 core machines. The machine configurations are given in Table 2. We use a set of multi-programmed workloads to extensively evaluate our proposed PriSM framework. We use a total of 71 workloads: 21 quad core workloads, 16 eight-core workloads, 20 sixteen-core workloads and 14 thirtytwo-core workloads. The workloads are detailed in the technical report [12]. Each benchmark in the workload is fast-forwarded for 10B instructions and detailed simulation is carried out until all the programs execute 500M instructions. In the case of 16 and 32 cores we simulate 200M instructions in detail. Statistics are reported only for the first 500M/200M instructions for each program under detailed simulation. The experimental methodology and the number of instructions simulated are in line with earlier works [14, 17, 20].

Table 2. System parameters

Front-End/Issue/Commit	4 Wide
Clock speed	4 GHz
ROB/IssueQueue/LQ/SQ	96/32/32/32
L1 D/I Cache	64KB, 64B blocks, 2 Way
L2 Cache (LLC)	4MB/8MB/16MB, 64B blocks, 16/32/64 Way
Number of Cores	4/8/16/32
Memory Controllers	1/2/4/8

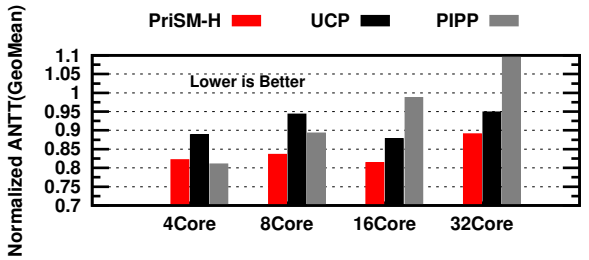
We use Average Normalized Turnaround Time (ANTT) [3] to summarize performance. ANTT is defined as $\sum(IPC_i^{SP}/IPC_i^{MP})/n$, where IPC_i^{SP} is the stand-alone IPC of program in $core_i$ and IPC_i^{MP} is its IPC when run as a part of the workload. ANTT is a metric of interest for the user and is a lower-is-better metric. Fairness is summarized by observing the relative gap between the minimum and maximum slowdown [3]. $Fairness = Min_{i,j}\left(\left(IPC_i^{MP}/IPC_i^{SP}\right) / \left(IPC_j^{MP}/IPC_j^{SP}\right)\right)$. Fairness is a higher-is-better metric with values in the range of 0 to 1.

In all our experiments, L2 is the last level shared cache. The default LLC replacement policy is LRU (unless specified otherwise). We use a 4MB 16-way associative L2 for 4 and 8 core machines. The 16-core machine uses a 8MB 32-way associative L2 and the 32-core machine uses a 16MB 64-way associative cache. We use shadow tags in 1/32 of the total number of sets. The allocation policies recompute the probabilities after the shared cache sees the same number of misses as number of cache blocks. As found in earlier work [18], the overheads of even a complex control theoretic allocation policy is negligible, less than 0.15% of the execution time. As our allocation policies are much simpler, we expect them to have practically no impact on the performance. Even for alternative proposals with which we compare, we assume that the computations performed by allocation policies does not add to the execution time.

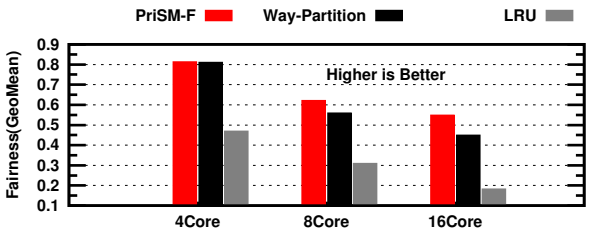
5. Results

Figure 2 shows the performance of PriSM-H and PriSM-F with increasing core counts. The performance of PriSM-

H is normalized to the performance of LRU cache. For PriSM-F we use the geomean of the fairness values. PriSM-H and PriSM-F consistently provide gains with increasing core count. It can be seen that PriSM-H achieves an average gain of 17.9%, 16.5%, 18.7% and 12.7% over LRU in the case of 4, 8, 16 and 32 cores respectively. Also our scheme outperforms UCP and PIPP at higher core counts (8,16 and 32) and performs as well as them in the quad core scenario. Similar trend is observed for fairness too where PriSM-F consistently provides better performance compared to LRU and the way-partitioning based solution [9].



(a) PriSM-H



(b) PriSM-F

Figure 2. PriSM performance summary

5.1. Comparison with UCP and PIPP

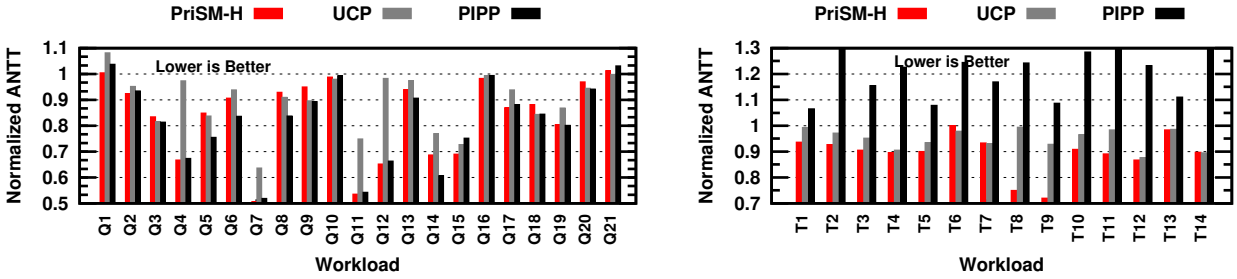
In this section we compare the performance of PriSM-H with UCP [14] and PIPP [20]. Figures 3(a) and 3(b) show the performance of the quad-core and 32-core workloads respectively. A good number of programs show more than 20% gain over LRU with Q7 showing as much as 50% gain over LRU. PIPP is able to provide more ways to cache friendly benchmarks like 179.art, 300.twolf and 471.omnetpp in quad-core workloads Q5, Q6, Q8 and Q14. In these workloads PIPP retains as many lines as possible brought in by the cache friendly benchmarks by inserting everything else closer to LRU. While this strategy helps in a few quad-core workloads, it ends up harming PIPP at higher core counts, as can be seen with most of the 32-core workloads. This behavior of PIPP has been observed in earlier works too [2, 11] and is primarily due to too many cores inserting lines closer to the LRU position and the lines being replaced early due to the increased cache contention.

It can be seen that PriSM-H works better than UCP in all the 32-core workloads and a large majority of the quad core workloads. To understand the quad core performance gap between UCP and PriSM better, we studied the occupancy of different programs in the various workloads under both schemes. Figure 4 shows the cache occupancy of individual cores when they finish 500M instructions in UCP and PriSM. As different programs finish at different point of time, the cumulative sum of occupancies need not equal 1 (whole cache). As expected, the cache space allocated to different programs by the two schemes differs. We discuss below the performance trends observed in a representative subset of workloads. In workload Q1, PriSM allocates more space to the relatively memory intensive benchamrk 168.wupwise and hence performs better than UCP. In workload Q4, PriSM allocates more space to benchmarks 175.vpr and 471.omnetpp compared to UCP. This is achieved at the expense of taking space allocated to 410.bwaves and 470.lbm. In workloads Q7, Q11 and Q12, PriSM-H benefits by providing more space to memory intensive benchmarks like 179.art and 471.omnetpp. On the otherhand, in workloads like Q3 and Q9, UCP was able to provide marginally more space to memory intensive benchmarks 179.art and 471.omnetpp. Hence in those workloads, UCP showed minor improvements over PriSM.

5.2. Benefits of fine-grained partitioning

To compare the benefits provided by fine-grained partitioning of PriSM over coarse-grained way-partitioning, we study the performance of using the same allocation policy (from Algo 1) with both the partitioning mechanisms. We adapt the allocation policy for way-partitioning by rounding off the outcome of the algorithm to the nearest integral number of ways. Figure 5 shows the relative performance in terms of ANTT (normalized to that of LRU) for sixteen core workloads. PriSM outperforms way-partitioning in all the sixteen core workloads. Similar results were observed for 4, 8 and 32 cores in the case of hit-maximization and also for fairness.

As we partition the cache space rather than the associativity, our scheme can work well even when the number of cores is equal to the number of ways. We demonstrate this with a 8MB 16-Way L2 as shared LLC for a 16-core machine. For way-partitioning, the smallest unit that can be allocated is 1 way or 512KB. So in this case, if each program has to receive some cache space, the partitioning solution becomes a trivial one of giving one way per core (hence we do not evaluate it). But for PriSM, the smallest allocatable unit is a cache block or 64Bytes. Figure 6 shows the performance of PriSM-H in terms of ANTT normalized to that of baseline LRU cache with 8MB-16 way configuration. It can be seen that for all the workloads, PriSM-H improves



(a) 4-Core (b) 32-Core

Figure 3. Hit-Maximization in Quad and 32-Core

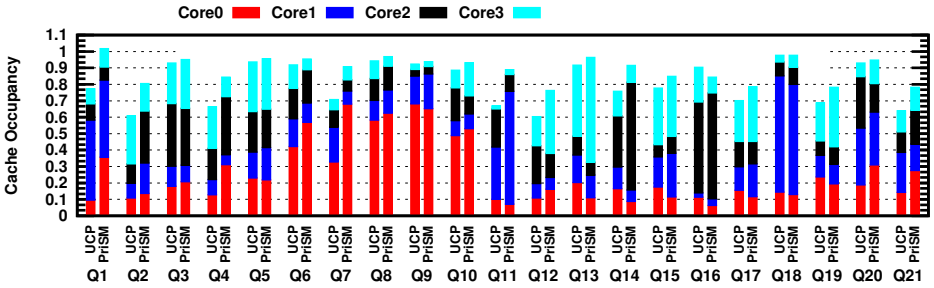


Figure 4. Cache occupancy of PriSM-H and UCP in quad-cores

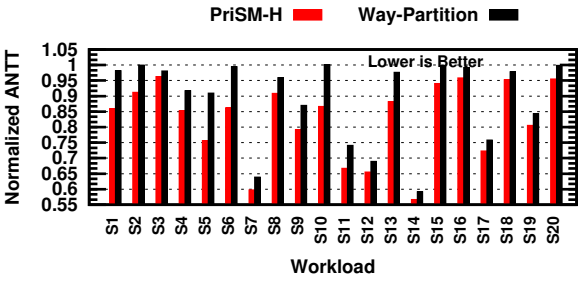


Figure 5. Hit-Maximization: Gains provided by PriSM over way-partitioning

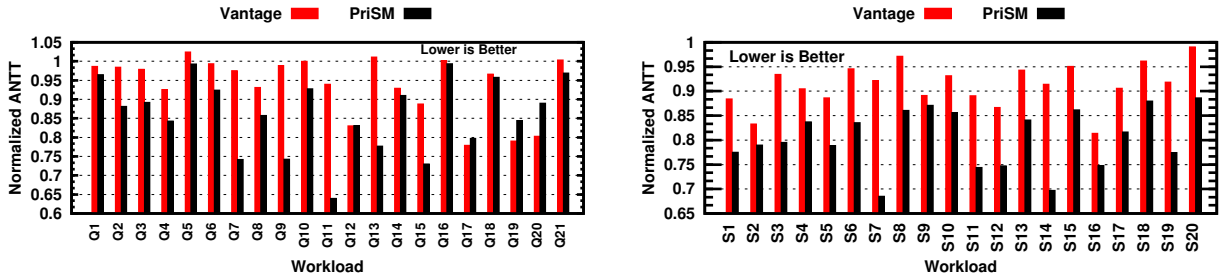
Figure 6. PriSM-H in a sixteen Core machine with 8MB 16-Way cache

the performance over LRU. The fine-grained partitioning allows us to share the cache space efficiently across the various programs resulting in an average gain of 14.8% over LRU.

5.3. Comparison with Vantage

In this section we compare the performance of PriSM with Vantage [17] in the context of set-associative caches. For a fair comparison, all the schemes use a timestamp based LRU [16,17] as the baseline replacement mechanism. We report results with both Vantage and PriSM using the extended UCP allocation policy that has been shown to work well with Vantage [17]. For all the parameters like width of timestamp, frequency of increment etc we use the values in the original vantage paper [17] . In short, this allows us to directly compare the efficiency of the partitioning mechanism of Vantage and PriSMFigure 7 shows the performance in terms of ANTT of vantage and PriSM, normalized to that of baseline set-associative cache managed using time-stamp LRU for 4 and 16 core machines. In quad-core workloads, it can be seen that in a majority of the workloads (all but Q12, Q17, Q19 and Q20), PriSM outperforms vantage. In the case of 16-core machine, PriSM outperforms Vantage in all the workloads. On average(geomean), PriSM performs better than vantage by 7.8% on quad-core workloads and 11.8% on sixteen core workloads.

To get a better understanding of the performance differences in the quad-core scenario, we plot (Figure 8) the normalized misses incurred by the individual benchmarks under PriSM. The misses are normalized to the misses ex-



(a) Quad-Core (b) Sixteen-Core

Figure 7. Performance of Vantage and PriSM in 4 and 16 core machines

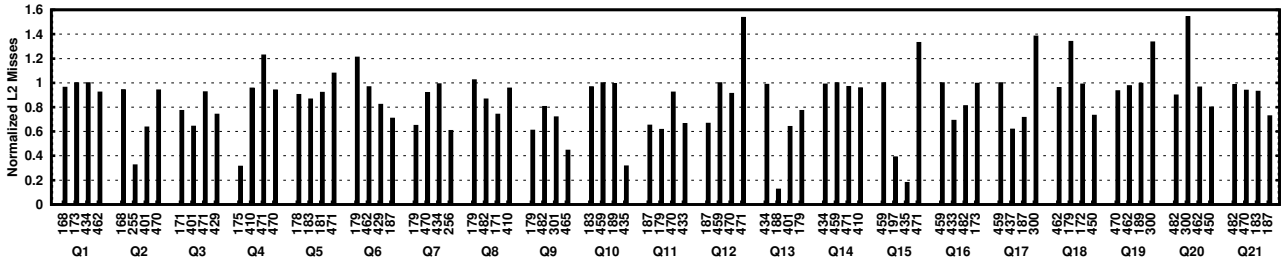


Figure 8. Misses incurred by PriSM normalized to Vantage

perienced by the benchmark under vantage. As can be seen PriSM reduces the misses incurred by at least three of the four benchmarks in all the quad-core workloads. In fact it improves the performance of all four benchmarks in 12 out of the 21 workloads. Even though misses go up with PriSM for one of the benchmarks in workloads Q4, Q5, Q6 and Q15, the significant decrease in the misses experienced by the other benchmarks helps PriSM to perform better than vantage in those workloads. In Q19 and Q20 there is an increase in misses for one benchmark (300.twolf). In these workloads, none of the other programs benefit to a great extent. Hence Vantage performs better than PriSM in those workloads. In the case of 16-core workloads, PriSM consistently performs better than Vantage.

5.4. Fairness

This section compares the performance of PriSM-F with a way-partitioning solution to achieve fairness [9]. PriSM improves fairness by 1.4%, 13.1% and 23.3% over way-partitioning in 4, 8 and 16-core workloads respectively. Figure 9 shows the absolute fairness of LRU, PriSM-F and way-partitioning in a 16-core machine with 8MB32Way shared L2 as LLC. A higher value indicates better fairness. As seen with hit-maximization, in a larger machine with more contention and sharing, fine-grained partitioning provided by PriSM gains in importance. This is reflected by the fact that the fairness improves with PriSM-F for all the workloads. Improving fairness at the expense of overall performance is undesirable. We observed that improved fairness was accompanied by an improvement in performance in all the workloads under PriSM-F. On an average(geomean), PriSM-F improves performance by 19% compared to LRU.

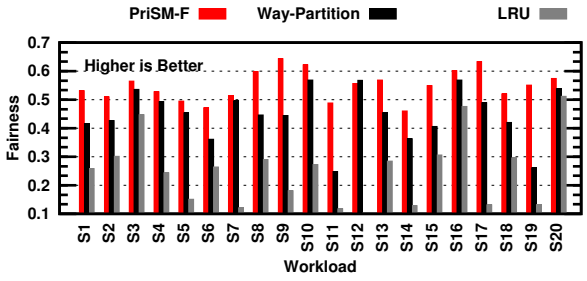


Figure 9. PriSM-F in sixteen core. Fairness of LRU, Way-partitioning and PriSM-F

5.5. Achieving quality-of-service with PriSM-Q

We define QOS as achieving a specified IPC target. We tested the QOS scheme by fixing the target IPC value for the program running in Core0 as 80% of its Stand-Alone IPC. Figure 10 shows the slowdown achieved by Core0 compared to Stand-Alone run for 16-core workloads. In a majority of the cases the slowdown is very close to the 80% target specified. For cache-insensitive programs the slowdown happens to be higher than 80% and we verified (by

setting the eviction probability to 1) that it is the maximum slowdown that the program can experience.

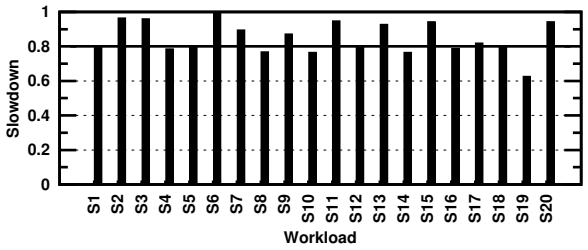


Figure 10. Achieving QOS using PriSM-Q. Performance guarantee of 80% StandAloneIPC for core0 in 16-core workloads

5.6. Analysis and sensitivity study

Eviction Probabilities. Figure 11 shows the mean and standard deviation of the eviction probabilities observed under PriSM-H. The values are shown for each individual benchmark in the various quad-core workloads. Each bar shows the mean value of the eviction probability while the error bar indicates the magnitude of standard deviation. The error bar is from (mean−std.dev) to (mean+std.dev). The probabilities are recomputed anywhere between 199 (Q2) and 1175 (Q5) times. It can be seen that the measured standard deviation in the eviction probabilities is low indicating that the probability values remain fairly stable. Similar stable behaviour was observed in the case of 8, 16 and 32 core workloads too.

Bits required for Eviction-probability. The probability values have to be stored in the hardware and need to be communicated to the cache controller by the allocation policies. To reduce the hardware overhead, we propose to store the probability values as 'K=6,8,10,12' bit integers. Figure 12 shows the performance of the K-bit variants of PriSM-H normalized to that of baseline PriSM-H which uses probabilities in the range $[0, 1]$. It can be seen that the performance with 6, 8, 10 and 12 bits is very similar to that of using floating point to represent probability. We observed similar behaviour in the case of 16-core workloads too with 6 and 8 bit precision. Hence it is enough to use 6 or 8 bits to represent eviction probability in PriSM.

Victim core selection. PriSM relies on the fact that given a sufficiently large interval, the actual evictions will match the eviction probability distribution. As mentioned earlier, we can face scenarios where no replacement candidate belonging to a core identified by the core-selection step is found

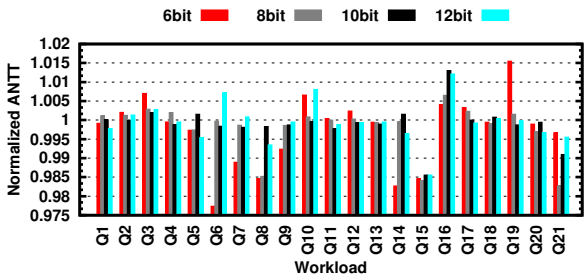


Figure 12. Performance of schemes using N-bits to represent probabilities compared to floating point representation

in the set. Figure 13 shows the impact of interval length on the fraction of such instances for quad-core workloads with PriSM-H. As expected, as the interval length increases from 32K misses to 128K misses, the fraction of cases where a desired replacement candidate was not found falls steadily from 3.8% of replacements on an average with 32K interval to 2.5% of replacements with 128K interval (3.1% at interval length of 64K misses). Similar trend was observed in the case of sixteen core workloads too.

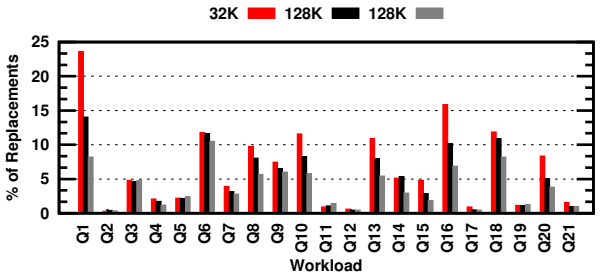


Figure 13. Fraction of replacements for which victim block belonging to desired core is not found

Changing the Replacement Policy. PriSM is agnostic to the underlying replacement mechanism and can work with any replacement policy with just an extension to include the core-selection step. We studied the performance of PriSM with DIP [13] as the underlying replacement policy. DIP does not exhibit stack-property and hence makes for an interesting choice. In the quad-core scenario, PriSM-H on a cache with DIP replacement improves performance over the baseline cache with DIP by 8.9%. TA-DIP [7] performed similar to that of the baseline cache with DIP. Both DIP and TA-DIP yeld better performance than LRU. Due to lack of space we do not show the detailed results for this case.

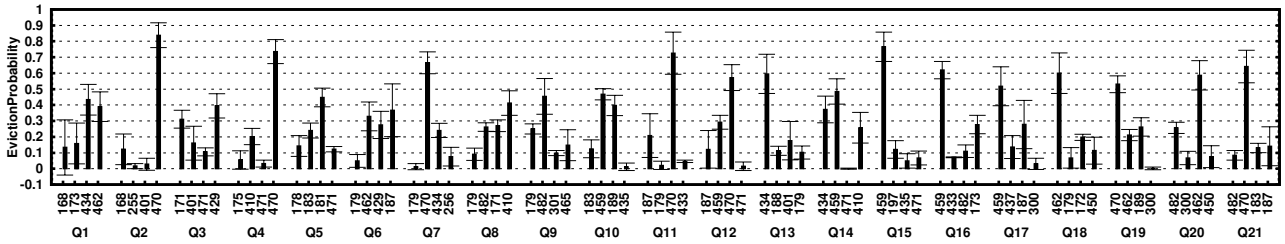


Figure 11. Quad-Core – Mean and Std.Dev of replacement probability

6. Related Work

Cache sharing is a well studied problem [5–7, 9, 10, 14, 15, 18, 20] and a variety of solutions have been proposed to achieve different performance and/or fairness goals. Broadly these solutions have two components: (i) a cache partitioning scheme to enforce partition decisions and (ii) allocation policies that build on the partitioning support and decide how much cache space each core should receive to achieve the desired performance target.

Way-partitioning. The most widely used solution is to partition the associativity (Way-partitioning) of the cache [6, 9, 14, 15, 18]. Other solutions include partitioning the sets of the cache [19] or using page-coloring techniques [10]. Unlike way-partitioning, these solutions suffer from limited support for reconfigurability when the workload characteristic changes. Enabling OS support to manage shared caches and convey the partitioning information computed by allocation policies has also been proposed to manage cache sharing [15, 18].

Vantage. Vantage [17] is the only other partitioning scheme that enables fine-grained partitioning. The key advantages of PriSM over Vantage are given below. PriSM is built from scratch to work with any standard cache while keeping the hardware overhead low. PriSM achieves the desired sharing goals by taking into account the incoming rate, the current cache occupancy and by computing the eviction rate/probability required to arrive at the desired target occupancy. This can be layered on top of any simple but efficient existing and well known cache. Vantage on the other hand logically partitions the cache into 'managed' and 'unmanaged' regions and achieves the desired target occupancy in the 'managed' portion of the cache by borrowing space from the unmanaged region. This is a fundamental change to the cache organization. Also vantage requires the replacement policies, including commonly used LRU, to be implemented in a Vantage-friendly fashion.

A cache hit in vantage needs to be region aware and promote the block to managed region if it is in unmanaged region. Whereas, hits in PriSM behave exactly like the baseline cache. On a miss, PriSM follows a two-step replacement process involving a core-selection and victim identification steps as explained earlier. Vantage on the other hand requires demotion of multiple blocks to the unmanaged region during each miss and various book keeping activities like recomputing the the SetPointTimeStamp and Aperture periodically as part of replacement in addition to identifying the victim block [17].

Allocation Policies. UCP [14] is an allocation policy built on top of way-partitioning to maximize the number of cache hits (Hit-Maximization). It allocated ways to different cores based on the concept of marginal utility.

In [9], it was observed that rate of increase in misses from stand-alone to shared cache scenario for any given program correlates well with the increase in execution time. The allocation policy proposed in [9] builds on top of way-partitioning and provides more ways to the cores that are doing poorly in terms of the miss-increase criterion while taking away ways from cores that are doing well so as to achieve similar levels of slowdown for all cores. CQos [6] tries to use way-partitioning and selective insertion/bypass into the cache to achieve QOS guarantees. These allocation policies are designed to work with way-partitioning and cannot take advantage of fine-grained partitioning abilities without modification.

Other schemes. There are a variety of other schemes of interest which do not rely on way-partitioning. The most effective and well-known among these are the hit maximization schemes TA-DIP [7] and PIPP [20]. These schemes do not rely on explicit partitioning but build a monolithic and complex replacement scheme that is a combination of the cache partitioning support and allocation policy. TADIP and PIPP work by adjusting the insertion and/or promotion mechanisms in the LRU replacement policy. Though the schemes are effective to achieve hit-maximization they cannot support other performance goals in their current form and hence lack the flexibility provided by the decoupling of partitioning support from allocation policies.

Other proposals like RRIP [8], PE-LIFO [2] and NUcache [11] are replacement policies proposed to manage shared caches. These schemes are primarily targeted towards hit maximization. However it is tough to extend these

replacement policies to achieve other performance goals like fairness or QOS.

7. Conclusions

In this paper, we proposed Probabilistic Shared Cache Management (PriSM), a scalable framework to manage shared cache occupancy in a fine-grained fashion. PriSM controls the occupancy of the different cores by controlling the eviction probabilities – the rate at which lines brought in by any given core are evicted. From a hardware perspective, PriSM requires minimal extensions to the underlying cache replacement policy. We demonstrate the flexibility of PriSM by implementing three allocation policies to achieve Hit-Maximization (PriSM-H), Fairness (PriSM-F) and QOS (PriSM-Q) in our proposed framework. We demonstrate the scalable nature of our solution by studying its performance from low core count (4-cores) to high core count (32-cores).

Acknowledgments

The first author was supported by a Microsoft Research India PhD fellowship during his PhD. The authors would like to thank Prof Matthew Jacob, Prof Mainak Chaudhuri, Ashwin Prasad, members of HPC lab and the anonymous reviewers for their useful feedback on earlier drafts of this paper.

References

[1] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26:52–60, 2006.

[2] M. Chaudhuri. Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches. In *MICRO 42*, pages 401–412, New York, NY, USA, 2009. ACM.

[3] S. Eyerman and L. Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *Micro, IEEE*, 28(3):42 –53, may-june 2008.

[4] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate CPI components. In *ASPLOS-XII*, pages 175–184, New York, NY, USA, 2006. ACM.

[5] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *PACT '06*, pages 13–22, New York, NY, USA, 2006. ACM.

[6] R. Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *ICS '04*, pages 257–266, New York, NY, USA, 2004. ACM.

[7] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *PACT '08*, pages 208–219, New York, NY, USA, 2008. ACM.

[8] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *ISCA '10*, pages 60–71, New York, NY, USA, 2010. ACM.

[9] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT '04*, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society.

[10] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA*, pages 367–378, 2008.

[11] R. Manikantan, K. Rajan, and R. Govindarajan. NUcache: An efficient multicore cache organization based on Next-Use distance. In *HPCA '11*, pages 243–253, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

[12] R. Manikantan, K. Rajan, and R. Govindarajan. Probabilistic shared cache management(PriSM). Technical report, Lab for High Performance Computing, IISc, 2012.

[13] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA '07*, pages 381–391, New York, NY, USA, 2007. ACM.

[14] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *MICRO 39*, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[15] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. In *PACT '06*, pages 2–12, New York, NY, USA, 2006. ACM.

[16] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *MICRO '43*, pages 187–198, Washington, DC, USA, 2010. IEEE Computer Society.

[17] D. Sanchez and C. Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *ISCA '11*, pages 57–68, New York, NY, USA, 2011. ACM.

[18] S. Srikantaiah, M. Kandemir, and Q. Wang. SHARP control: controlled shared cache management in chip multiprocessors. In *MICRO 42*, pages 517–528, New York, NY, USA, 2009. ACM.

[19] K. Varadarajan, S. K. Nandy, V. Sharda, A. Bharadwaj, R. Iyer, S. Makineni, and D. Newell. Molecular Caches: A caching structure for dynamic creation of application-specific heterogeneous cache regions. In *MICRO 39*, pages 433–442, Washington, DC, USA, 2006. IEEE Computer Society.

[20] Y. Xie and G. H. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ISCA '09*, pages 174–183, New York, NY, USA, 2009. ACM.