PCASA: Probabilistic Control-Adjusted Selective Allocation for Shared Caches

Konstantinos Aisopos[†], Jaideep Moses[‡], Ramesh Illikkal[‡], Ravishankar Iyer[‡], Donald Newell[§]

[†]*Dept. of Electrical Engineering*
Princeton University
Princeton, NJ

[‡]*Intel Labs*
Intel Corporation
Hillsboro, OR

[§]*Server Product Group*
AMD
Austin, TX

***Abstract*—Chip Multi-Processors (CMPs) are designed with an increasing number of cores to enable multiple and potentially heterogeneous applications to run simultaneously on the same system. However, this results in increasing pressure on shared resources, such as shared caches. With multiple processor cores sharing the same caches, high-priority applications may end up contending with low-priority applications for cache space and suffer significant performance slow-down, hence affecting the Quality of Service (QoS).**

In datacenters, Service Level Agreements (SLAs) impose a reserved amount of computing resources and specific cache space per cloud customer. Thus, to meet SLAs, a deterministic capacity management solution is required to control the occupancy of all applications. In this paper, we propose a novel QoS architecture, based on Probabilistic Selective Allocation (PSA), for priority-aware caches. Further, we show that applying a control-theoretic approach (Proportional Integral controller) to dynamically adjust PSA provides accurate and fine-grained capacity management.

1. Introduction

Chip Multi-Processors (CMPs) have become mainstream in the marketplace and the trend is towards increasing the computational resources. On the other hand, cores still share platform resources, such as multiple levels of caching and memory bandwidth. In the past, important applications were given higher priority with priority-aware OS scheduling. In other words, a high-priority (HP) application was given more compute time than a low priority (LP) application. However, since multiple cores are now available, the OS may schedule LP and HP applications simultaneously. While this allows more compute time for both applications, it may not translate to higher performance, due to contention in the shared cache. The burden of performance differentiation between HP and LP priority applications now falls on the rest of the platform where contention for shared resources takes place.

In datacenters, Service Level Agreements (SLAs) impose a reserved amount of computing resources and specific cache space per cloud customer. Thus, datacenters need to provide cache space guarantees to each application and protect it from other streaming or thrashing applications concurrently running in the same system. To address this problem, researchers have proposed Quality of Service (QoS) techniques, where the amount of cache capacity occupied by each application is directly controlled based on counters [2,3,9] or restricted via way partitioning [7,11]. However, these approaches either add to the architecture complexity, or do very coarse-grained capacity management.

In this paper, we present PCASA, a low overhead QoS-aware architecture based on Probabilistic Selective Allocation (PSA). PSA takes advantage of the eviction mechanism to control the occupancy of applications, by probabilistically determining the rate at which lines become MRU (the rest of the lines are marked as LRU). PSA is a non-intrusive mechanism and easily implementable in real platforms, but exhibits a lack of determinism. To address this limitation, we utilize a control-theoretic approach, based on a Proportional Integral controller (PI controller), to dynamically adjust PSA at runtime and deterministically meet occupancy targets. Our approach achieves accurate and fine-grained control of shared resources not possible before without the use of expensive hardware.

This paper is organized as follows: Section 2 presents the related work on QoS enforcement techniques for capacity management. Section 3 introduces PCASA and details our PI controller. Section 4 discusses our simulation infrastructure and shows results through which we explain the trade-offs for our mechanism. Finally, Section 5 concludes this paper.

2. Related Work

Several capacity monitoring and enforcement mechanisms have been recently proposed to allocate a specific amount of cache resources based on application priorities, or other objectives (maximizing the overall throughput, achieving fairness among applications, etc). These mechanisms utilize the following allocation schemes to enforce an occupancy target to applications: way partitioning [7,11], capacity counters in cache [3,9] or set [2,9] granularity. In this section, we briefly describe these schemes and their trade-offs, and then distinguish how our approach is different and novel.

Way Partitioning [7,11] achieves very accurate but coarse-grained capacity management. Each application is assigned a number of ways to allocate to, thus the target occupancy for each application must be a multiple of a way's capacity. If there are few ways, it is not possible to provide sufficient granularity to support several priority levels. In addition, when applications are underutilizing the ways assigned to them, other applications cannot use these ways, which potentially results in hurting the overall performance.

978-3-9810801-8-6/DATE12/©2012 EDAA

[§]This work was done while the author was working for Intel

Table 1. Overhead and occupancy granularity comparison for QoS mechanisms

	granularity of occupancy target	area overhead (per cache line)		hardware to implement QoS enforcement logic (estimation)
		% lines	additional bits	
Way Partitioning	multiple of total capacity/ways	0%	no additional bits	partitioning logic, masks (3Kgates)
Capacity Counters	fine grained (any value)	100%	owner bits (=2)	LRU tuning logic (5Kgates)
PSA	fine grained (any value)	3%	owner bits (=2)	LRU tuning logic and PI (5Kgates)

Capacity Counters [2,3,9] keep track of applications' occupancy in a per-cache line/per set granularity, and control their occupancy by biasing the eviction mechanism to evict cache lines of the applications exceeding their quota. This requires all cache lines to be tagged with "owner" bits, for the eviction logic to identify the application each line belongs to and appropriately choose an over-quota victim. Occupancy enforcement based on Capacity Counters is fine grained, since the occupancy of each application can be restricted to any value by controlling the number of its evicted lines.

The uniqueness of PSA lies on the fact that it can achieve fine-grained occupancy control with an implementation overhead comparable to coarse-grained mechanisms (20x less than fine-grained mechanisms). PSA tunes the LRU bits whenever allocating/reading/writing a cache line; at that time the application the line belongs to is known, thus there is no need to tag it with "owner" bits, or incorporate any counter. PSA does though keep track of the *overall* occupancy of each application. Fortunately, this does not require additional bits to identify the owner of every single cache line (as in Capacity Counters), since low overhead monitoring mechanisms are available to monitor an application's *overall* occupancy by sampling a small percentage (3%) of the cache, as discussed in [12].

Table 1 summarizes the granularity and hardware overhead tradeoffs of the discussed QoS mechanisms, as estimated by Intel. We assume 4 applications (2 bits to indicate the "owner" application of a cache line). As shown in Table 1, PSA achieves fine-grained granularity (any occupancy value can be enforced), while its hardware overhead is 20x less than Capacity Counters, since only 3% of cache lines (sampled lines [12]) store the "owner" bits. The hardware to implement the enforcement logic is negligible for all enforcement mechanisms, since it is stateless combinational logic with a few control inputs. We note here that incorporating two additional bits in every cache line is considered a very significant overhead by industry architects and unlikely to be implemented in Intel's future products.

In this paper, we are not attempting a quantitative comparison to explore the performance differentiation that can be achieved with these techniques, since our primary target is deterministic occupancy control to meet an SLA. Instead, we show that PSA can achieve fine-grained and highly accurate capacity management with 20x less hardware overhead than any other fine-grained mechanism. Intel estimates a total of 0.5 to 0.7% area overhead in the Last Level Cache (LLC) die to implement PCASA in an actual product (including additional bits, logic, PI controllers, and wiring). In contrast, capacity counters incur a 20% area overhead, due to tagging each cache line with "owner" bits.

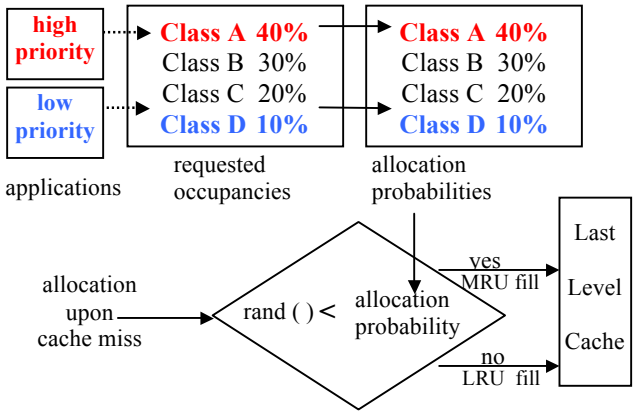

Figure 1. Baseline PCASA Architecture

3. PCASA: A Priority-Aware Architecture

This section presents the PCASA architecture and its components (Figure 1). This architecture introduces platform priority classes: each application belongs to a priority class and its priority class defines the percentage of the cache that it is allowed to occupy. In the example of Figure 1, applications are mapped to four priority classes, with each specifying a requested (target) occupancy in the cache.

3.1 Baseline PCASA

To restrict each priority class to the requested occupancy, we introduce a probabilistic approach to control the occupancy of its applications, by modifying their lines' position in the LRU-stack during fills. In a regular cache replacement policy, every cache fill marks the corresponding cache line as MRU. Instead, we propose a probabilistic replacement policy, called Probabilistic Selective Allocation (PSA), where we probabilistically determine whether the cache line will be marked as MRU or not in each fill. Each class is assigned an allocation probability of 0%-100%. This allocation probability corresponds to the chance the applications of this class have to set the touched line as MRU in each fill. The allocation probability each application is assigned has a direct effect on its occupancy in the shared cache. The higher the probability, the more lines will be set to MRU, the less lines will be evicted, hence the more cache space the application will occupy. As shown in Figure 1, this mechanism is implemented as follows: a random number (0-100) is generated in every fill (based on a linear feedback shift register for instance); if this number is lower than the allocation probability then the line is filled as MRU. Otherwise, when the application fails to set the touched line as MRU, the line is marked as LRU.

In our baseline PCASA, the allocation probabilities match the requested occupancy percentages. For example, if the requested occupancy of an application is 30%, this implies that its allocation probability will be 30%. However, filling 70% of its lines as LRU does not guarantee that these lines will be replaced soon and the application will be limited to 30% of cache space. If the application has a larger memory footprint compared to other applications concurrently running, more lines have to be marked as LRU so that more lines are evicted. The allocation probability that is required to meet a specific occupancy target is not deterministic, as it depends heavily on the access patterns and phases of all workloads running in the system, thus needs to be calculated on-the-fly. The solution we propose is to monitor each priority class' resulting occupancy at runtime and calculate the difference of requested and resulting occupancy. This difference is then used to dynamically adjust the allocation probabilities to higher or lower values and restrict the occupancy of each priority class on-the-fly to converge to the requested value. To achieve this, we incorporate a Proportional Integral controller to PCASA, which enables accurate and fine-grained capacity management.

3.2. PCASA Incorporating a PI Controller

Closed loop Proportional and Integral (PI) controllers are used in applications ranging from temperature control to sophisticated space robots. The controller takes some system behavior as its requirement, monitors the system and applies the necessary corrections in a closed-loop execution to achieve the required behavior. The control output (PI output) can be represented using the following equation:

$$\textbf{PI output} = Kp * error + Ki * \sum error$$

The error is the deviation of the monitored value from the desired setpoint. Based on the current error (proportional part) and the aggregate error of previous measurements (integral part), the control output is corrected to make the system output converge to PI input. Kp and Ki are constants determining the weights of the proportional and integral part.

Recently, PI or PID controllers have been used for shared resource management in the memory subsystem [11] and the NoC [10]. In our implementation of the PI controller we used cache occupancy as the desired setpoint (Figure 2). Our architecture monitors the occupancy obtained through PSA (with set sampling [12]) and derives the error as the deviation from the requested occupancy. This error is then used by the PI to compute the new allocation probabilities. This closed loop of error detection and corrected allocation probabilities will converge quickly to the optimal allocation probabilities that provide the desired occupancies. To achieve this, we use a PI controller per priority class, which performs the following actions every 15ms: (1) adds the current error to the sum of all previous error measurements ($\sum error$), and (2) re-calculates the probabilities using the equation $Kp*error+Ki*\sum error$. We observed little sensitivity when

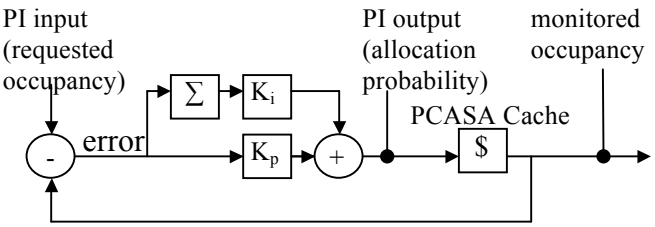

Figure 2. PI controller to tune allocation probabilities

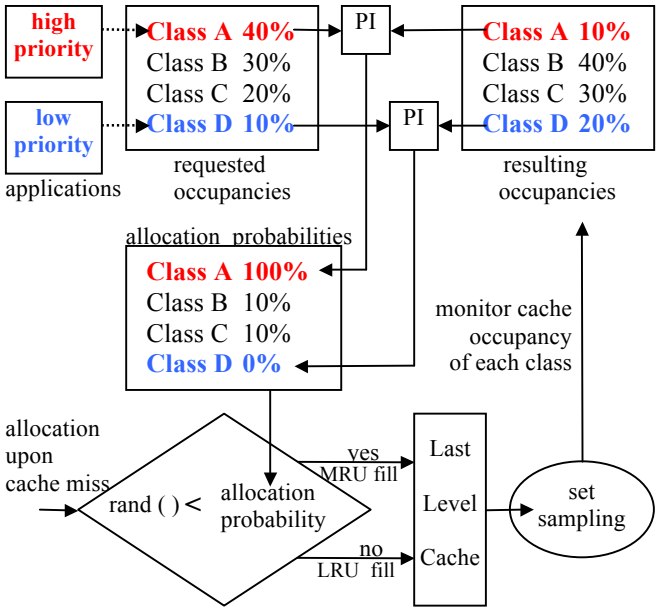

Figure 3. PCASA Architecture incorporating PI controllers

activating the PI every 1ms to 100ms, and thus chose 15ms since it is conveniently comparable to the OS time-quanta.

The hardware cost of a PI controller is minimal (2 registers, 3 FP 32bit adders, 2 FP 32bit multipliers). Its output is restricted to [0,1] and it constantly provides the allocation probability value to its corresponding class. Kp and Ki determine how quickly PI converges and if it stabilizes to the desired occupancy without oscillation. There are two approaches to calibrate a PI controller: empirical and theoretic. While theoretic approaches require process modeling and can be complex, many PI controllers even in the control theory domain are empirically tuned [13]. Our PI controller was empirically calibrated, by extensively studying the sensitivity of our system to Ki,Kp. We chose (Kp,Ki)=(0.6,0.2) for modest speed of convergence and high system stability. These values are the same across all workloads and nearly constant across platforms.

Figure 3 depicts how PCASA incorporates PI controllers. The occupancy of each priority class is monitored using set sampling [12] and the difference of the requested and resulting occupancy is used as feedback to the corresponding PI controller, which adjusts its allocation probability on-the-fly to converge to the requested occupancy value. The figure

demonstrates a scenario where LP applications have large working sets, and thus exceed their requested occupancies for baseline allocation probabilities (equal to the values of requested occupancies). The PI controller detects this and sets the highest allocation probability to HP class A (100%), while lowering the allocation probabilities of other classes to (10%,10%,0%), so that class A can utilize more cache space to meet its occupancy target (40%).

3.3 Variants of Probabilistic Selective Allocation

Although controlling the LRU stack of an application can significantly decrease its occupancy, there may be cases of a memory intensive LP application running concurrently with less memory intensive HP applications, where the LP cannot be restricted as much as we want. Once the PI sets the allocation probability of the LP to 0% (all newly allocated lines become LRU), the LP cannot be restricted further. In order to deal with cases where 0% is not enough to prevent the LP from utilizing cache space reserved for the HP applications, we consider the following variations of PSA:

- Drop Fills: Fills may bypass the cache, as studied in [2]; this can be accomplished only if the cache hierarchy is non-inclusive. However, many cache hierarchies (e.g. Intel's cache hierarchy) are strictly inclusive. Since this mechanism is less interesting industrially, we do not evaluate it.

- K0H (Keep 0 on Hits): When hits occur to cache lines, the LRU state also changes probabilistically, similar to fills. A random number is generated and if the random number is lower than the allocation probability, then the hit does not update the state of the line (the line does not become MRU).

***- 1WB (1-Way Buffer)*:** When PSA fills a cache line as LRU, it also restricts it to allocate to a limited number of ways (in our implementation one specific way). This technique is actually way partitioning on the top of PSA: lines that probabilistically fail to become MRU, are further restricted by a way mask to a specific way, in order to minimize their interference with lines that probabilistically succeed to become MRU. Consequently, once PI sets the allocation probability to 0% to minimize the occupancy of an application, in addition to all lines being filled as LRU, the application will be restricted to one way.

4. Evaluation and Results

4.1 Performance Evaluation Framework

In this section, we briefly describe the CMPSched$im [5] simulation framework we used to evaluate PCASA. CMPSched$im is an extension of CMP$im [4], a parallel multi-core performance simulator. CMP$im utilizes the PIN [6] binary instrumentation system to evaluate the performance of single-threaded, multi-threaded, and multi-programmed workloads on a single- or multi-core processor. CMP$im models a simple processor pipeline and uses PIN to dynamically feed instructions and memory addresses from workloads executing in actual processor cores to simulated cores. This way, CMP$im realistically simulates concurrently executing workloads and avoids the I/O overheads associated with large address trace files. CMPSched$im has the added benefit of enabling the execution of multiple applications per core, while the scheduling is accurately captured by a real Linux scheduler, Linsched [1]. In our experiments, we simulated Intel's i3-530 cache hierarchy without L1 caching (256KB 4-way private L2, 4MB 16-way shared L3), since L1 caching only slightly affected L3 contention (where we studied the effectiveness of PCASA), while negatively impacting our simulation speed.

4.2. Results and Analysis

4.2.1 Efficacy of PCASA in Achieving Occupancy Targets

In this section, we present a case study where we evaluate PSA, its variants, and the PI controller using CMPSched$im. We start with a breadth exploration, where we simulate a variety of SPEC benchmark pairs to demonstrate the effectiveness of PCASA in enforcing a fixed "common case" occupancy target. Then, we delve into a challenging benchmark pair, to explore its sensitivity to meeting various occupancy targets and its dynamic occupancy behavior.

To get a good representation of the SPEC workload spectrum, we have simulated over 25 pairs, but we demonstrate 5 combinations (due to space restrictions), which capture diversity in characteristics such as memory intensiveness, memory sensitivity, and streaming nature (Table 2). These pairs are a representative sample of our experiments. In each pair, the first application is the HP that is allowed to use all of the cache (100% allocation probability) and the second application is the LP (restricted application). For our breadth exploration, we have a fixed occupancy target of 25% for the restricted application.

In Figure 4, we plot the monitored occupancy of the restricted application for these five pairs. For each pair, 8 bars are shown, corresponding to 8 different allocation schemes. The first 4 are: Probabilistic Selective Allocation (PSA), PSA with Keep Zero on Hits (K0H), PSA with 1-Way Buffer (1WB), and PSA with both K0H and 1WB. The following 4 bars correspond to the same allocation schemes when integrating a PI controller.

Table 2: Benchmark combinations

(art, mcf) = (sensitive, sensitive)
(applu, wupwise) = (insensitive, sensitive)
(equake, twolf) = (streaming, insensitive)
(lucas, galgel) = (streaming, sensitive)
(swim, mgrid) = (streaming, streaming)

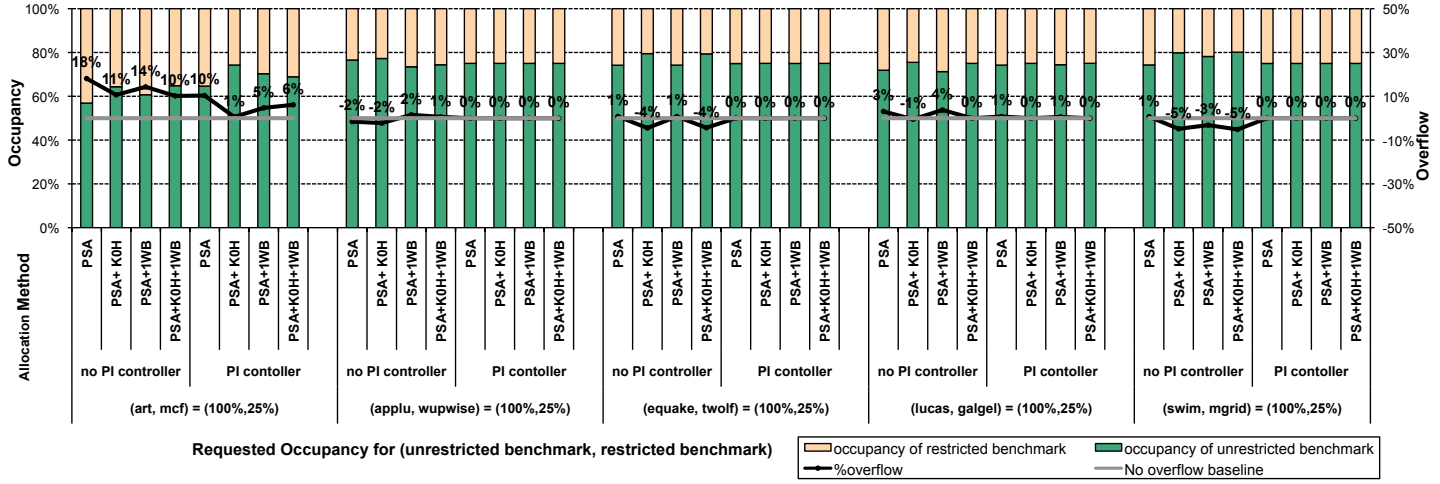

Figure 4. Effectiveness of PSA for a variety of SPEC workloads with LP requested occupancy of 25%

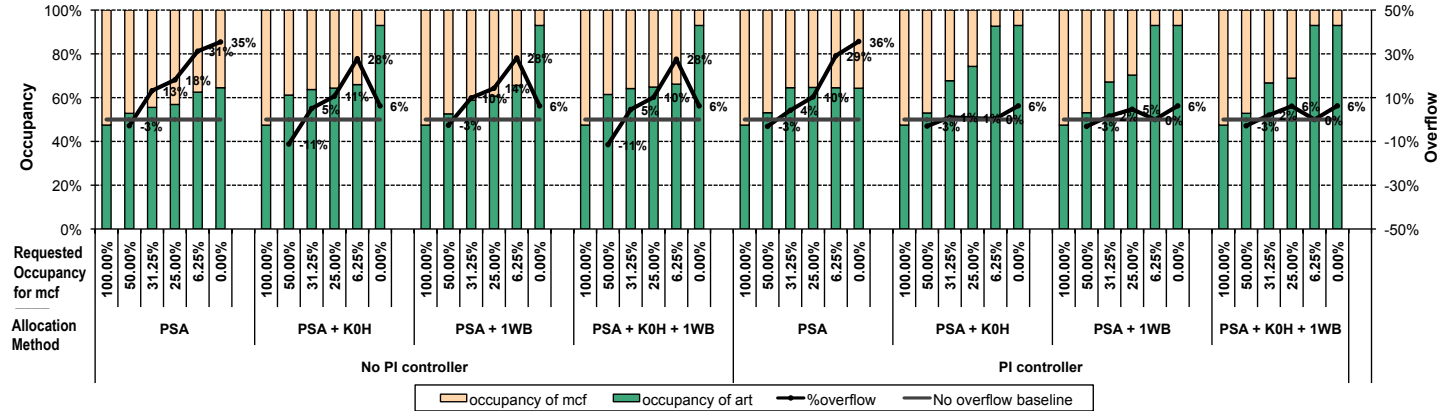

Figure 5. Effectiveness of PSA for (art, mcf) with various requested occupancy

The very first bar of (*art*, *mcf*) shows the occupancy of *mcf*, when using PSA to restrict its occupancy to 25%. The resulting occupancy is 43%. The secondary axis shows by how much the resulting occupancy of the restricted application was off (in terms of % occupancy). Since *mcf* had a resulting occupancy of 43%, it had an overflow of 18% (43% - 25%) and hence this mechanism was not as effective for this pair. Also, PSA+K0H and PSA+1WB are not that effective in containing *mcf*, resulting in an overflow of 10% or more. Once the PI mechanism is used in conjunction with PSA and its variants, we are able to achieve a higher accuracy. The 4 next bars show how the same variants of PSA perform when integrating a PI controller. With PI+K0H or PI+1WB, a high degree of accuracy is achieved containing the capacity overflow within 6%.

As shown in Figure 4, other benchmark pairs meet their occupancy targets without the need for a PI controller (overflow or underflow contained in 5%). *Art* and *mcf* seems to be the most challenging pair, since we observe the highest differentiation between the requested and resulting values in occupancy. PSA by itself, PSA with PI, or PSA with variants without the PI, do not perform effectively for this pair. That is because of the high degree of disparity in working sets between the two applications. This problem occurs when a restricted application (like *mcf*) has a very large working set (or very high request rate) into the cache compared to the HP application. Thus, the HP application does not operate at a fast enough rate to evict the LP application's lines, although these are filled as LRU. In our experiments, we did observe a few more pairs with similar behavior to (art, mcf), although these are not the common case.

In Figure 5, we delve deeper into the (*art*, *mcf*) example (in depth exploration). We keep *art* as HP (no capacity restrictions) and restrict the requested occupancy of *mcf* to variable values (X-axis). Y-axis shows *mcf*'s actual resulting occupancy in the cache. Note that 100% requested occupancy represents the default behavior without QoS (both applications set the touched line as MRU with 100% probability). In this case, *mcf* and *art* end up occupying 53% and 47% of the Last Level Cache (LLC) respectively.

In the same figure, we observe that the more we attempt to restrict the occupancy of *mcf*, the more inaccuracy we have. Without using a dynamic PI based mechanism to adjust the allocation probabilities at runtime, there's a potential occupancy inaccuracy up to 38% (for low occupancy target values).

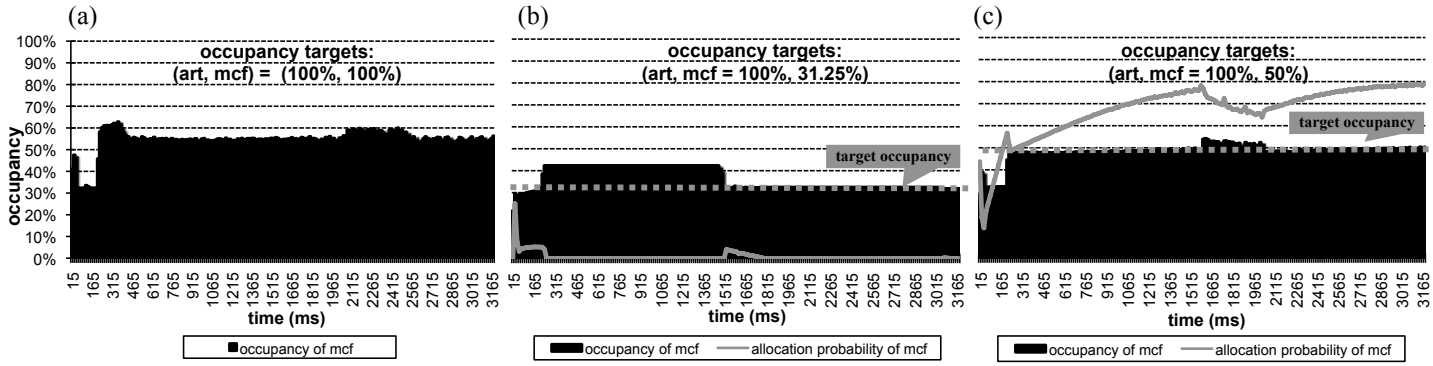

Figure 6. Occupancy dynamics for art-mcf running in a 4MB LLC (PSA+K0H+PI)

When a PI controller is incorporated with PSA+K0H, the overflow and underflow are monitored dynamically and the allocation probabilities are adjusted to achieve an accurate occupancy within 1%. The 0% requested occupancy however results in a 6% overflow, because of the underlying limitation that we don't have a mechanism to bypass the cache and some lines do remain in cache though they are marked LRU.

The last two allocation methods (PSA+1WB and PSA+K0H+1WB) are also effective in containing the overflow / underflow within 6%. However, they do require the 1WB as an additional mechanism, which is unnecessary since PSA+K0H with the PI controller already provides the desired accuracy. We observed similar behavior for the rest of the few challenging pairs we came across (not shown here), thus we believe that PSA+K0H+PI is by itself a very practical and effective mechanism for shared cache management. Next, we show the dynamic occupancies over time for *mcf* and *art* (for PSA+K0H+PI).

4.2.2 Occupancy Dynamics

Figure 6(a) shows the occupancy dynamics for the first 3 seconds of the execution of *art* and *mcf* without QoS (100% allocation probability for both applications). X-axis indicates the time in 15ms granularity, while Y-axis depicts the corresponding cache occupancy of *mcf* (in terms of percentage). We note that *art* and *mcf* occupy 47% and 53% of the cache (in respect) most of the time, corresponding to the first bar (100%) in Figure 5. The occasional spikes in occupancy (e.g. 2.1-2.5 seconds) are due to phase changes.

In Figures 6(b) and 6(c), we observe how the occupancy curve changes when enforcing an occupancy target with the proposed PI controller (configuration is PSA+K0H+PI). In Figure 6(b) the target occupancy for *mcf* is 31.25%, while in 6(c) it is 50% (dashed lines in subfigures). The continuous line indicates the allocation probability that PI enforces in order to achieve the desired occupancy. In 6(c), we observe that PI raises allocation probability over 50% most of the time to enable *mcf* to occupy more space than baseline PSA, and achieve the 50% occupancy target. We also observe that whenever the *mcf*'s occupancy exceeds 50% (e.g. 1.6-2.1 seconds), the allocation probability drops to further restrict *mcf*. In contrast, in 6(b), where the target occupancy is lower (31.25%), we observe that the allocation probability is limited to less than 10% most of the time. Note that in baseline PSA this value would be constantly 31.25%.

5. Conclusions

Differentiated services and QoS are gaining importance as Moore's law increases the number of processing elements in multicore systems. In this paper, we introduced PCASA, a novel low-overhead QoS architecture using Probabilistic Selective Allocation (PSA) for shared cache management in multicore systems. We also demonstrated that when integrating a control-theoretic Proportional Integral (PI) controller into the system, PSA achieves highly accurate occupancy enforcement to applications. At only 0.5-0.7% area overhead, our solution can be utilized in datacenters to satisfy Service Level Agreements (SLAs) and provide cache space guarantees to cloud customer applications.

References

[1] J. Calandrino, D. Baumberger, T. Li, J. Young, S. Hahn, "Linsched-The Linux Scheduler Simulator", PDCCS, May 2008, New Orleans, Louisiana.
[2] R. Iyer, "CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms," ICS, June 2004, Malo, France.
[3] R. Iyer, L. Zhao, F. Guo, R. Illikkal, D. Newell, Y. Solihin, L. Hsu and S. Reinhardt, "QoS Policies and Architecture for Cache/Memory in CMP Platforms", ACM Sigmetrics, June 2007, San Diego, CA.
[4] A. Jaleel, R. S. Cohn, C.-K. Luk, B. Jacob, "CMP$im- A Pin-Based On-The-Fly Multi-Core Cache Simulator", MoBS, June 2008, Beijing, China.
[5] J. Moses et al, "CMPSched$im: Evaluating OS/CMP Interaction on Shared Cache Management", ISPASS, April 2009, Boston, MA.
[6] Pin - A Dynamic Binary Instrumentation Tool, rogue.colorado.edu/pin
[7] M. K. Qureshi, Yale N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches", MICRO, December 2006, Orlando, FL.
[8] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. C. Steely Jr, and J. Emer, "Adaptive Insertion Policies for Managing Shared Caches on CMPs", PACT, October 2008, Toronto, Canada.
[9] N. Rafique, W.-T. Lim, and M. Thottethodi, "Architectural Support for Operating System-Driven CMP Cache Management", PACT, September 2006, New York, NY.
[10] A. Sharifi, H. Zhao, M. Kandemir, "Feedback Control for Providing QoS in NoC Based Multicores," DATE, March 2010, Dresden, Germany.
[11] S. Srikantaiah, M. Kandemir, and Q. Wang, "SHARP control: Controlled Shared Cache Management in Chip Multi-Processors," MICRO, December 2009, New York, NY.
[12] L. Zhao et al., "CacheScouts: Fine-Grain Monitoring of Shared Caches in CMP Platforms", PACT, September 2007, Brasov, Romania.
[13] K. Kiyong, R. Schaefer, "Tuning a PID Controller for a Digital Excitation Control System," IEEE Trans. Industry Applications, March 2005.