Refining the Utility Metric for Utility-Based Cache Partitioning [*]

Xing Lin, Rajeev Balasubramonian
School of Computing, University of Utah

Abstract

It is expected that future high-performance processors will implement large L2 or L3 caches that will be shared by multiple cores. Allocating shared cache space among multiple programs is an important problem and has received much attention in recent years. Typically, the ways or sets of a cache are partitioned across multiple competing programs. Most recent work is built on the concept of marginal utility, i.e., a way is assigned to a program that is expected to benefit most by receiving that additional way. Benefit *is usually quantified as misses per kilo-instruction (MPKI). A cache partition is chosen such that overall MPKI for a workload is minimized. However, the ultimate performance metric of interest is overall IPC or throughput, not overall MPKI. MPKI is used as a proxy for IPC because it is much easier to compute – recent work has suggested that MPKI per additional way can be easily computed by maintaining a small shadow tag structure. In this paper, we first quantify how the use of MPKI instead of IPC can result in sub-optimal cache partitions. It is well-known that misses have varying impacts on IPC across programs because of varying levels of latency tolerance in each program. As a result, we discover a non-trivial number of cases where the use of MPKI is sub-optimal. This number increases as more programs share a given cache. We then propose a simple mechanism that uses two IPC samples and a miss rate curve to compute an IPC curve. This allows us to better quantify marginal utility in terms of IPC and arrive at performance-optimal cache partitions.*

1 Introduction

Future processors will likely implement deep cache hierarchies. Each L2 or L3 cache may be shared by multiple cores. The shared cache space will therefore have to be partitioned across multiple programs in a manner that optimizes overall performance. This is a problem that has received much attention in recent years.

A paper by Qureshi and Patt [17] proposes *Utility-based Cache Partitioning (UCP)* where the ways of a cache are assigned to programs such that the overall number of misses (misses per kilo instruction or MPKI) is minimized. Each program maintains a sampled shadow tag structure that tracks the tag contents for a few sets assuming that all ways of the cache are available. For each cache hit, the LRU position of the block is tracked; this allows estimation of whether the access would have been a hit or miss for different cache associativities. Thus, an approximate miss rate curve (MPKI as a function of ways) is computed for each program. At regular intervals, the miss rate curves are examined and ways are assigned to programs such that the overall MPKI is minimized.

However, the end goal of any cache partitioning mechanism is the optimization of overall performance, expressed by either a weighted-speedup metric (sum of relative IPCs) or a throughput metric (sum of IPCs). MPKI is used as a proxy for IPC because the former is easier to calculate with a shadow tag structure. The hope is that by optimizing MPKI, we will also optimize IPC. But, it is well-known that the IPC impact of a miss depends on the extent of latency tolerance within the program. The average cost for a miss in one program is usually not equivalent to that cast in another program. In some cases, throughput may be maximized by incurring a high MPKI for a latency-tolerant program while reducing the MPKI for another latency-intolerant program. The corresponding way partition need not necessarily have the lowest combined MPKI. This is an observation also made in prior work [9, 15, 22]. The goal of this paper is to quantify the inaccuracy in way partition decisions by focusing on the easier-to-compute MPKI metric. As a case study, we will focus most of this analysis on the approach proposed in the UCP paper [17], i.e., reserving a fixed number of ways for each program.

The results of this analysis will also have bearing on other state-of-the-art cache partitioning schemes. There exist many cache partitioning approaches in the literature [1, 3, 9–12, 18–21, 23–26], described in more detail in Section 4. Most of these bodies of work, whether they implicitly or explicitly partition the cache, make their final decisions based on miss counting and not on IPC impacts. With the exception of a few QoS papers, most do not track the IPC impact of their policy choices. Hence, the extent of inaccurate decision-making described in this paper may also manifest in other mechanisms that are based on miss rate estimation. We also show that this inaccuracy increases as more cores share a cache, further motivating

---

[*] This work was supported in parts by NSF grants CCF-0811249, CCF-0916436, NSF CAREER award CCF-0545959, HP, Intel, SRC grant 1847.001, and the University of Utah.

Decode/Issue	128-entry instruction window with no scheduling restrictions.
Pipeline	8-stage, 4-wide pipeline; A maximum of two loads and a maximum of one store can be issued every cycle.
Execution Units	All instructions have one-cycle latency except for cache misses.
branch predictor	perfect branch prediction
L1 Instruction Cache	32KB 4-way set associative cache with LRU replacement
L1 Data Cache	32KB 8-way set associative with LRU replacement
Unified L2 Cache	256KB, 8-way set-associative with LRU replacement; L2 cache hits are 10 cycles.
Last Level Cache(LLC)	16-way 1 MB cache with LRU replacement; L3 cache hits are 30 cycles.
Memory	memory requests have a 200-cycle latency.

Table 1. Baseline processor configuration.

a closer look at IPC-based metrics in future work.

We start by quantifying the differences in way partition decisions that are focused on IPC optimization and MPKI optimization. In this initial analysis, we assume that a miss rate curve and IPC curve (as a function of cache space) are magically known beforehand. We show that these optimization strategies often diverge, especially when more programs share a given cache. We then propose a simple mechanism that uses two execution samples and a miss rate curve to estimate an IPC curve. We use this estimated IPC curve to improve the decisions made by the UCP mechanism.

2 Comparing MPKI and IPC Optimization

2.1 Experimental Methodology

Simulator We use the simulation framework provided by the first JILP Workshop on Computer Architecture Competitions (JWAC-1, Cache Replacement Championship). It is based on the CMP$im simulator and models a simple out-of-order superscalar with a 128-entry instruction window processor. The baseline configuration is described in Table1.

Workload We use 23 SPEC CPU2006 benchmarks compiled for the x86-64 architecture. The reference input set is used. We fast forward past the first billion instructions and simulate the following two billion instructions in detail. For each benchmark, we vary the number of ways from 1 to 16, while keeping the number of sets fixed at 1024, and compute the MPKI and IPC for each configuration. To facilitate the analysis of all combinations of two, three, and four benchmarks, we wrote scripts in Perl to calculate the IPCs and MPKIs of all possible cache partitions for each combination. Some of the scripts used for our analysis can be obtained at the following link: `https://github.com/xinglin/mpki-cpi`

Metrics The UCP paper makes periodic partition decisions based on hit counters associated with each position of the LRU stack for each program's shadow tag structure. Hence, UCP estimates misses per kilo cycles (MPKC), not MPKI, and a partition is selected to optimize overall MPKC. Similar metrics are also commonly employed in other papers. However, overall MPKC is a metric that suffers from some problems. For example, consider a partition that favors program A, but that is highly unfavorable to program B. Compared to a baseline where program B receives the entire cache, B now suffers from many misses and executes at a much lower IPC. Compared to the baseline, B has a much higher miss rate, but because it executes much fewer instructions in a given time quantum, it also yields fewer misses. Therefore, even though program B is suffering (much higher MPKI than the baseline), it has a lower MPKC than the baseline. Therefore, as a baseline in this paper, we use MPKI instead of MPKC. We have verified that this change in metric does not impact the overall conclusions of the study. In hardware, it is easy to replace the MPKC metric with the more accurate MPKI metric with an additional counter that tracks committed instructions per epoch.

In our analysis, the UCP baseline tries to minimize overall MPKI, i.e., it tries to minimize the sum of MPKIs. Our proposed models attempt to pick cache partitions that maximize throughput. We consider two metrics for throughput. One is the sum of IPCs. The second is the weighted speedup, where we add the relative IPCs for each program. Relative IPC for a program is defined as the IPC of a program divided by its standalone IPC (where the program receives the entire cache).

2.2 Analysis of results

Our first goal is to understand if MPKI optimization strategies can indeed result in throughput optimization. Since we already have miss rate and IPC curves (as a function of ways), it is easy to estimate cache partitions that optimize for MPKI or IPCs – we use the term *divergence* to refer to situations where different cache partitions are selected for the two.

Table 2 shows how optimizing for MPKI and Weighted Speedup can diverge in various cases. Table 3 is a similar table that shows how optimizing for sum of IPCs and

MPKI can diverge. The trends are very similar in both tables; we observe that the divergence is greater when considering the IPC-sum metric.

The first row in Table 2 shows that of all possible benchmark combinations ($^{23}C_2$ for the two program case), a large fraction is divergent. The percentage of divergent cases grows as more programs share the cache. The other rows of the table provide a breakdown of how often significant divergence (10%, 8%, ...) is observed in terms of weighted-speedup and combined MPKI. For the 4-program model, more than 26% of the cases show a weighted-speedup difference greater than 4%. It is worth noting that this difference is comparable to the average performance improvements shown in many cache optimization papers. The last four rows show an average in weighted-speedup and MPKI differences. When considering the average across all possible benchmark combinations, we see that only about 1% of performance is left on the table by optimizing for MPKI instead of the performance metric itself. While there are many cases where divergence is significant, the average difference (1%) is small because for more than half the workload combinations, cache partitions with the two optimization strategies aren't divergent. We see that in many cases, weighted-speedup is optimized by incurring an MPKI that is more than 50% higher than the alternative MPKI optimization strategy. This data drives home the point that simply focusing on miss counting can often lead to highly sub-optimal cache partitioning decisions.

The importance of IPC optimization is even more stark when examining the IPC-Sum metric (Table 3). In a number of cases, the MPKI-optimal strategy is off by more than 20% in terms of the IPC-Sum metric. On average across all benchmark combinations, the MPKI-optimal strategy is off by about 3.4%, thrice the difference seen for the weighted-speedup metric. Given that most cache policy papers report performance improvements under 10%, this difference is significant enough that it is worth considering seriously when designing cache optimizations.

Most of the divergence is because a cache miss, on average, has varying degrees of latency tolerance in each program. As an illustrative example, in Figure 1 we show the MPKI and CPI curves for a workload consisting of programs bzip2 and gcc. We can see that a comparable MPKI difference in the two programs results in varying CPI differences. Figure 2 shows the combined MPKI and weighted-speedup for this workload as 1 to 15 ways are assigned to the first of the two programs. Because of the varying latency tolerance in these programs, we see that the weighted-speedup and MPKI curves are optimized at partitions that are very different. MPKI is minimized when bzip2 receives three ways, but weighted speedup is maximized when bzip2 receives 15 ways.

3 Making Accurate CPI Predictions

The previous section shows that better cache partitions can be made by focusing on the IPC metric instead of on the MPKI metric and it quantifies the extent of divergence in the two optimization strategies. The MPKI metric has the nice feature that a simple sampled shadow tag structure can estimate the MPKI curve for all possible associativities at run time. In order to estimate an accurate IPC curve, we instead need to execute the program with varying ways for an epoch each. For a 16-way large LLC cache, this implies a large "exploratory" phase before decisions can be made. To put the analysis of the previous section to good practical use, we need to find low overhead mechanisms to estimate an IPC curve. While many possibilities may exist, we consider the efficacy of one simple mechanism in this work.

We first make the observation that the CPI curve often has a somewhat linear relationship with the MPKI curve. Depending upon the latency tolerance in the program, each additional cache miss has a roughly constant average impact on the increase in execution time. Of course, as a program gets more or less memory bound, the latency tolerance changes and the impact of each additional miss on execution time is no longer constant. However, we hypothesize that assuming a constant is a reasonable approximation. We can therefore express a program's CPI as a function of the number of allocated ways w as follows:

$$CPI(w) = c_1 + c_2 \times MPKI(w)$$

Given an MPKI curve and the values of c_1 and c_2 for each program, we can estimate the CPI curve for each program and thus compute better cache partitions. In order to estimate c_1 and c_2, we will need the CPIs for two way allocations. This significantly shortens the exploratory phase; we simply need to run our workload for two epochs with different way allocations before making any decisions.

The error in our CPI estimation is a function of the way allocations selected for our two samples. With perl scripts, we computed CPI estimation errors for every possible choice of two samples. We first considered a case where each benchmark was allowed to magically select the two sample points that yielded minimum error in the CPI estimation curve. In this model, we observed that the average error in CPI estimations across the entire benchmark suite was only 0.38%. The highest error is 2.04% for soplex. Only 2 out of 23 benchmarks had an error greater than 1% and 4 benchmarks had an error greater than 0.5%. Table 4 shows the best estimated value of c_2 for each program. This value represents the latency tolerance of each program and Table 4 confirms the high variance in this value across programs.

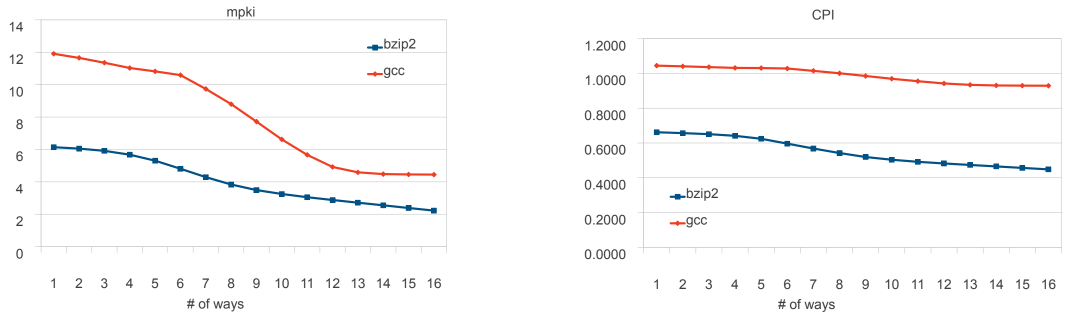
We also considered a more practical case where every workload selected the same two sample points for their estimation of c_1 and c_2. Based on our exhaustive analysis,

Metric	2 Programs	3 Programs	4 Programs
Divergent Cases	84/253 (33.20%)	828/1771 (46.75%)	4827/8855 (54.51%)
Wt-Spdup $\geq$ 10%	3/84 (3.57%)	4/828 (0.48%)	0/4827 (0.0%)
Wt-Spdup $\geq$ 8%	4/84 (4.76%)	39/828 (4.71%)	1/4827 (0.02%)
Wt-Spdup $\geq$ 6%	8/84 (9.52%)	99/828 (11.96%)	312/4827 (6.46%)
Wt-Spdup $\geq$ 4%	12/84 (14.29%)	151/828 (18.24%)	1268/4827(26.27%)
Wt-Spdup $\geq$ 2%	24/84 (28.57%)	262/828 (31.64%)	1793/4827 (37.15%)
MPKI $\geq$ 50%	4/84 (4.76%)	27/828 (3.26%)	154/4827 (3.19%)
MPKI $\geq$ 40%	4/84 (4.76%)	31/828 (3.74%)	189/4827 (3.92%)
MPKI $\geq$ 30%	6/84 (7.14%)	72/828 (8.70%)	290/4827 (6.01%)
MPKI $\geq$ 20%	8/84 (9.52%)	92/828 (11.11%)	574/4827 (11.89%)
MPKI $\geq$ 10%	11/84 (13.10%)	134/828 (16.18%)	1000/4827 (20.72%)
MPKI $\geq$ 5%	18/84 (21.43%)	257/828 (31.04%)	1666/4827 (34.51%)
Wt-Spdup avg (all)	0.59%	0.95%	1.13%
Wt-Spdup avg (divergent cases)	1.79%	2.03%	2.08%
MPKI avg (all)	2.54%	3.89%	4.42%
MPKI avg (divergent cases)	7.66%	8.31%	8.12%

Table 2. Extent of divergence when optimizing for MPKI and Weighted Speedup. Cache partitioning optimized for MPKI is used as the baseline. For all possible workloads, the first row shows the number of cases where MPKI optimization and Weighted Speedup optimization arrived at different cache partitions. The next five rows show the magnitude of divergence, i.e., in how many cases did the weighted speedups of the two optimization strategies differ by 10%, 8%, ... The next six rows show the magnitude of divergence in terms of MPKI for the two optimization strategies. The last four rows show the average change in Weighted Speedup and MPKI (across all cases and across just the divergent cases) for the two optimization strategies.

Metric	2 Programs	3 Programs	4 Programs
Divergent Cases	110/253 (43.48%)	1088/1771 (61.43%)	6548/8855 (77.50%)
IPC-Sum $\geq$ 20%	5/110 (4.55%)	26/1088 (2.39%)	8/6548 (0.12%)
IPC-Sum $\geq$ 15%	10/110 (9.09%)	77/1088 (7.08%)	140/6548 (2.14%)
IPC-Sum $\geq$ 10%	16/110 (14.55%)	187/1088 (17.19%)	959/6548 (14.65%)
IPC-Sum $\geq$ 5%	29/110 (26.36%)	352/1088 (32.35%)	2426/6548 (37.05%)
MPKI $\geq$ 50%	12/110 (10.91%)	96/1088 (8.82%)	412/6548 (6.29%)
MPKI $\geq$ 40%	15/110 (13.64%)	128/1088 (11.76%)	507/6548 (7.74%)
MPKI $\geq$ 30%	18/110 (16.36%)	207/1088 (19.03%)	859/6548 (13.12%)
MPKI $\geq$ 20%	19/110 (17.27%)	252/1088 (23.16%)	1454/6548 (22.21%)
MPKI $\geq$ 10%	25/110 (22.73%)	331/1088 (30.42%)	2384/6548 (36.41%)
MPKI $\geq$ 5%	42/110 (38.18%)	565/1088 (51.93%)	3580/6548 (54.67%)
IPC-Sum avg (all)	1.85%	2.90%	3.40%
IPC-Sum avg (divergent cases)	4.26%	4.72%	4.60%
MPKI avg (all)	6.97%	9.84%	10.01%
MPKI avg (divergent cases)	16.02%	16.02%	13.54%

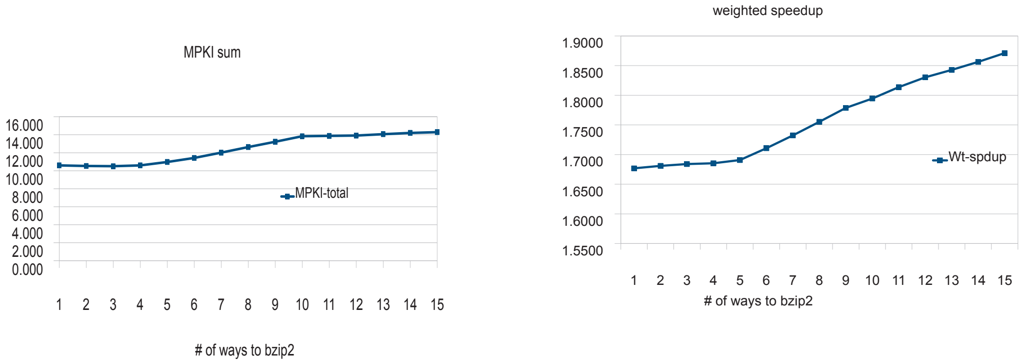
Table 3. Extent of divergence when optimizing for MPKI and IPC-Sum. Cache partitioning optimized for MPKI is used as the baseline. For all possible workloads, the first row shows the number of cases where MPKI optimization and IPC-Sum optimization arrived at different cache partitions. The next four rows show the magnitude of divergence, i.e., in how many cases did the IPC-Sum of the two optimization strategies diff by 20%, 15%, ... The next six rows show the magnitude of divergence in terms of MPKI for the two optimization strategies. The last four rows show the average change in IPC-Sum and MPKI (across all cases and across just the divergent cases) for the two optimization strategies.



(a) MPKI for bzip2 and gcc as a function of allocated ways.

(b) CPI for bzip2 and gcc as a function of allocated ways.

Figure 1. MPKI and CPI curves for bzip2 and gcc.



(a) Combined MPKI for bzip2 and gcc as a function of ways assigned to bzip2.

(b) Weighted speedup for bzip2 and gcc as a function of ways assigned to bzip2.

Figure 2. Combined MPKI and weighted speedup for bzip2 and gcc.

gromacs	0.1367	hmmer	0.0812
gamess	0.0630	namd	0.1625
calculix	0.1116	astar	0.1731
mcf	0.0539	cactusADM	0.0627
lbm	0.1630	bwaves	0.0097
h264ref	0.1187	libquantum	0.0366
leslie3d	0.0449	milc	0.1725
soplex	0.0786	zeusmp	0.0520
sphinx3	0.1029	povray	0.1416
sjeng	0.1327	omnetpp	0.0328
bzip2	0.0557	tonto	0.0714
gcc	0.0157		

Table 4. Best estimate of c2 for each benchmark.

we selected sample points that allocated 4 and 15 ways as they had the least average error, 0.53%. The highest error is 2.53% for soplex. Only 2 out of 23 benchmarks had an error greater than 2% and 3 of them had error greater than 1%.

Having discussed the accuracy of our CPI estimation scheme, we next employ this scheme for practical IPC-based cache partitioning. Specifically, we use the same two IPC sample points (4 ways and 15 ways) for all benchmarks, combine these with an MPKI curve to generate an IPC curve, and use this curve to optimize either weighted speedup or sum of IPCs. Table 5 repeates the analysis of Table 2, but uses the estimated IPC curve instead of a magically known precise IPC curve. The divergence is very similar to that seen before.

We believe it is worthwhile to explore other low-overhead strategies to estimate IPC curves. For example, MLP metrics [16] can be used to modify MPKI curves to arrive at IPC curves. A similar approach was considered

Metric	2 Programs	3 Programs	4 Programs
Divergent Cases	89/253 (35.18%)	844/1771 (47.66%)	5038/8855 (56.89%)
Wt-Spdup $\geq$ 10%	3/89 (3.37%)	4/844 (0.47%)	0/5038 (0.00%)
Wt-Spdup $\geq$ 8%	4/89 (4.49%)	39/844 (4.62%)	1/5038 (0.02%)
Wt-Spdup $\geq$ 6%	8/89 (8.99%)	99/844 (11.73%)	303/5038 (6.01%)
Wt-Spdup $\geq$ 4%	12/89 (13.48%)	151/844 (17.89%)	1253/5038 (24.87%)
Wt-Spdup $\geq$ 2%	23/89 (25.84%)	262/844 (31.04%)	1779/5038 (35.31%)
IPC-Sum $\geq$ 20%	4/89 (4.49%)	24/844 (2.84%)	8/5038 (0.16%)
IPC-Sum $\geq$ 15%	7/89 (7.87%)	61/844 (7.23%)	117/5038 (2.32%)
IPC-Sum $\geq$ 10%	12/89 (13.48%)	140/844 (16.59%)	731/5038 (14.51%)
IPC-Sum $\geq$ 5%	25/89 (28.09%)	265/844 (31.40%)	1783/5038 (35.39%)
MPKI $\geq$ 50%	4/89 (4.49%)	25/844 (2.96%)	135/5038 (2.68%)
MPKI $\geq$ 40%	4/89 (4.49%)	29/844 (3.44%)	155/5038 (3.08%)
MPKI $\geq$ 30%	6/89 (6.74%)	70/844 (8.29%)	251/5038 (4.98%)
MPKI $\geq$ 20%	7/89 (7.87%)	100/844 (11.85%)	538/5038 (10.68%)
MPKI $\geq$ 10%	10/89 (11.24%)	147/844 (17.42%)	981/5038 (19.47%)
MPKI $\geq$ 5%	17/89 (19.10%)	262/844 (31.04%)	1703/5038 (33.80%)
Wt-Spdup avg (all)	0.58%	0.93%	1.11%
Wt-Spdup avg (divergent cases)	1.66%	1.96%	1.96%
IPC-Sum avg (all)	1.44%	2.23%	2.56%
IPC-Sum avg (divergent cases)	4.09%	4.68%	4.50%
MPKI avg (all)	2.42%	3.90%	4.34%
MPKI avg (divergent cases)	6.89%	8.19%	7.63%

Table 5. Extent of divergence when optimizing for MPKI and Weighted Speedup, based on MPKI (former) or fixed-way predicted CPIs (latter). Cache partitioning based on the MPKI is used as the baseline. The first row shows the number of divergent cases, the next five rows show the extent of weighted-speedup divergence, the next four rows show the extent of IPC-sum divergence (note that we are optimizing for weighted-speedup), and the next six rows show the extent of MPKI divergence. The last six rows show the average difference (across all combinations and across only divergent combinations) in these three metrics for the two optimization strategies.

by Moreto et al. [15].

4 Related Work

A large number of papers use cache miss rates as a metric when selecting a configuration or policy. Since cache misses have varying impacts on IPC, its use as a metric can lead to inaccurate decisions when comparing misses for different programs. This effect shows up most prominently when a shared cache is being partitioned across multiple programs for throughput or QoS. This effect has been previously mentioned in other papers [9, 15, 16, 22], but its impact on cache partition decisions has not been quantified. Jaleel et al. [9] use policies that are based on miss rate estimates, but point out that accuracy can be improved by using metrics that more closely approximate CPI. Qureshi et al. [16] take MLP (a measure of latency tolerance) into account in their replacement policy. Suo et al. [22] propose a modified version of UCP that attempts to optimize IPC instead of miss rates. However, that work uses an equation based on cache access latencies to convert MPKI to CPI and focuses on an algorithm to efficiently compute the optimal partition. Moreto et al. [15] propose an algorithm that takes an application's MLP into account when estimating cache partitions. Our work focuses on an analysis to understand the error introduced by simpler MPKI metrics; we then propose the use of a simple 2-sample exploration to convert MPKI to CPI.

The following bodies of work focus on cache partitioning for throughput. Suh et al. [21] were the first to use marginal utility for cache partitioning and use a large number of counters to estimate miss rate curves. The work of Qureshi and Patt [17] shows that low-complexity mechanisms can be designed to achieve coarse-grained (one way at a time) cache partitioning. Yeh and Reinman use a shadow tag structure to estimate miss rate curves and implement cache partitioning in a D-NUCA cache [25]. PIPP [24] and TADIP [9] are implicit cache partitioning schemes that determine insertion points based on miss rate curves or miss rates for competing policies. The work of Liu and Yeung [14] picks a victim selection policy for implicit cache partitioning based on the IPC impact of different policies. Chang and Sohi [3] cycle through unfair partitions where one core receives most of the available cache space. Adaptive Set Pinning [19] allocates sets among competing applications by measuring hits and misses to a set from different applications. Tam et al. show how miss rate curves can be computed at run-time on modern processors and use this information to implement cache partitions with page coloring [23]. The same authors also suggest the use of cache stall-rate curves that can be estimated with performance counters [2]. The works of Cho and Jin [4], Lin et al. [12, 13], and Awasthi et al. [1] also implement set-based cache partitioning with page coloring that is primarily based on miss rate estimations. Zhuravlev et al. [26] compute approximate miss rates of programs and assign programs to a collection of shared caches such that overall miss rates of each shared cache are equalized. Jiang et al. [10] allocate heterogeneous private caches across many programs in a workload based on miss rate curve estimations.

The following bodies of work fall under the umbrella of QoS policies. Rafique et al. [18] focus most of their cache partitioning study on cache miss metrics, but also employ policies that use IPC metrics to ensure that program slowdowns are in proportion to program priorities. Guo et al. [6] point out that it is easier to achieve cache space targets than IPC or miss rate targets when providing QoS. Iyer et al. [7, 8] also focus on cache space targets for QoS enforcement. Srikantaiah et al. [20] express equations so miss rates can be translated into IPC when providing QoS. Kim et al. [11] effect incremental cache allocation adjustments every epoch to cause uniform miss rate degradations in all applications.

Apart from the few exceptions mentioned above, almost all related work on cache partitioning focuses on miss rates to guide their policies. QoS policies are better at being IPC-aware because many QoS policies ultimately try to cap IPC slowdown. In other related work, Dropsho et al. [5] estimate miss rate curves with per-way counters to reduce energy by selectively disabling cache ways.

5 Conclusions

It is well known that misses have varying impacts on IPC across programs. Even though IPC is the ultimate metric of interest, several cache optimization policies base their decisions on miss rate estimates because they are easier to compute. Little is known about the possible error introduced by using miss rate as a proxy for IPC. This work uses utility-based cache partitioning (UCP) as a case study for examining this error.

Our findings show that the error is non-trivial and grows as more programs share a given cache. While the difference in the optimization strategies is fairly small (1%-3%) when taking an average across all workloads, there are many instances where the difference is significant. When 4 programs share a cache, MPKI-based decisions are sub-optimal 55% of the time. Cache partitioning based on accurate IPCs can improve the weighted speedup metric by more than 4% in 1268 of the 4827 possible workloads, and cause an average increase of 1% in weighted speedup across all possible workloads. When considering sum of IPCs as the performance metric, an average 3% performance increase over MPKI-based UCP can be achieved, across all possible workload combinations. The IPC-based optimization strategy can improve the IPC-Sum metric by more than 5% in 37% of all 4-program workloads. This argues for the use of IPC-based metrics in any cache optimization mechanism, especially when multiple programs are sharing a cache.

We suggest a simple IPC estimation mechanism that is based on a short exploratory phase and the expected linear relationship between MPKI and CPI. With this mechanism, we are able to make cache partition decisions that are sub-optimal in few cases, causing an average performance increase comparable to cache partitioning based on accurate CPIs. As future work, we believe that it is worthwhile to explore other IPC estimation mechanisms, especially those that do not involve an exploratory phase. For example, MLP may be used to estimate latency tolerance and generate IPC curves out of MPKI curves.

References

[1] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter. Dynamic Hardware-Assisted Software-Controlled Page Placement to Manage Capacity Allocation and Sharing within Large Caches. In *Proceedings of HPCA*, 2009.

[2] R. Azimi, D. Tam, L. Soares, and M. Stumm. Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring. *ACM SIGOPS OSR*, April 2009.

[3] J. Chang and G. Sohi. Co-Operative Cache Partitioning for Chip Multiprocessors. In *Proceedings of ICS*, 2007.

[4] S. Cho and L. Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *Proceedings of MICRO*, 2006.

[5] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. L. Scott. Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 141–152, September 2002.

[6] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A Framework for Providing Quality of Service in Chip Multi-Processors. In *Proceedings of MICRO*, 2007.

[7] R. Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *Proceedings of ICS*, 2004.

[8] 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. In *Proceedings of SIGMETRICS*, 2007.

[9] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, J. S. Steely, and J. Emer. Adaptive Insertion Policies For Managing Shared Caches. In *Proceedings of PACT*, 2008.

[10] X. Jiang, A. Mishra, L. Zhao, R. Iyer, Z. Fang, S. Srinivasan, S. Makineni, P. Brett, and C. Das. ACCESS: Smart Scheduling for Asymmetric Cache CMPs. In *Proceedings of HPCA*, 2011.

[11] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings PACT*, 2004.

[12] 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 *Proceedings of HPCA*, 2008.

[13] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Enabling Software Management for Multicore Caches with a Lightweight Hardware Support. In *Proceedings of Supercomputing*, 2009.

[14] W. Liu and D. Yeung. Using Aggressor Thread Information to Improve Shared Cache Management for CMPs. In *Proceedings of PACT*, 2009.

[15] M. Moreto, F. Cazorla, A. Ramirez, and M. Valero. MLP-Aware Dynamic Cache Partitioning. In *Proceedings of HiPEAC*, 2008.

[16] M. Qureshi, D. Lynch, O. Mutlu, and Y. Patt. A Case for MLP-Aware Cache Replacement. In *Proceedings of ISCA*, 2006.

[17] M. Qureshi and Y. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of MICRO*, 2006.

[18] N. Rafique, W. Lim, and M. Thottethodi. Architectural Support for Operating System Driven CMP Cache Management. In *Proceedings of PACT*, 2006.

[19] S. Srikantaiah, M. Kandemir, and M. Irwin. Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors. In *Proceedings of ASPLOS*, 2008.

[20] S. Srikantaiah, M. Kandemir, and Q. Wang. SHARP Control: Controlled Shared Cache Management in Chip Multiprocessors. In *Proceedings of MICRO*, 2009.

[21] G. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *J. Supercomput.*, 28(1), 2004.

[22] G. Suo, X. Yang, G. Liu, J. Wu, K. Zeng, B. Zhang, and Y. Lin. IPC-Based Cache Partitioning: An IPC-Oriented Dynamic Shared Cache Partitioning Mechanism. In *Proceedings of ICHIT*, 2008.

[23] D. Tam, R. Azimi, L. Soares, and M. Stumm. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. In *Proceedings of ASPLOS*, 2009.

[24] Y. Xie and G. Loh. PIPP: Promotion/Insertion Pseudo-Partitioning ofMulti-Core Shared Caches. In *Proceedings of ISCA*, 2009.

[25] T. Yeh and G. Reinman. Fast and Fair: Data-Stream Quality of Service. In *Proceedings of CASES*, 2005.

[26] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of ASPLOS*, 2010.