Noname manuscript No.
(will be inserted by the editor)

RECU: Rochester Elastic Cache Utility

Unequal Cache Sharing is Good Economics

Chencheng Ye[†] · Jacob Brock · Chen Ding · Hai Jin

Received: date / Accepted: date

Abstract When renting computing power, fairness and overall performance are important for customers and service providers. However, strict fairness usually results in poor performance. In this paper, we study this trade-off. In our experiments, equal cache partitioning results in 131% higher miss ratios than optimal partitioning. In order to balance fairness and performance, we propose two elastic, or movable, cache allocation baselines: Elastic Miss Ratio Baseline (EMB) and Elastic Cache Space Baseline (ECB). Furthermore, we study optimal partitions for each baseline with different levels of elasticity, and show that EMB is more effective than ECB. We also classify programs from the SPEC 2006 benchmark suite based on how they benefit or suffer from the elastic baselines, and suggest essential information for customers and service provider to choose a baseline.

Keywords cache partition · fairness · elastic cloud

1 Introduction

Computing power is increasingly rented. Following Amazon's Elastic Computer Cloud (EC2), part of Amazon Web Services (AWS), many such commercial services are now available. Their adoption is accelerating, shown for

[†] Chencheng Ye is a student visiting University of Rochester, funded by the Chinese Scholarship Council.

The research is supported in part by the National Science Foundation (Contract No. CNS-1319617, CCF-1116104, CCF-0963759), IBM CAS Faculty Fellow program, the National Science Foundation of China (Contract No. 61328201) and a grant from Huawei.

Chencheng Ye(✉) · Hai Jin
Huazhong University of Science and Technology, Wuhan, China
E-mail: yechencheng@gmail.com

Chencheng Ye · Jacob Brock · Chen Ding
University of Rochester, Rochester NY 14627, USA

example by media reports on the rapid increase in usage and revenue of AWS. On such rented computers, applications from one customer may frequently run on the same physical computer with applications from other customers. The performance of an application depends on the behavior of peer applications and the management of shared resources.

One of most important resource is the shared cache. On a modern processor, the most expensive operation is a memory access, which may take hundreds of CPU cycles. In modern programs, most data accesses, 99% or more, happen in not memory but in cache. It is so important to maximize cache performance that most transistors on a processor chip are there to implement the cache, especially the last-level cache (LLC), which is up to 45MB and 96MB on recent Intel Haswell and IBM Power 8 architectures.

This paper studies the sharing of the last-level cache on cloud computers. Since cache is the most important resource affecting the computing speed, its allocation is critical to efficiency and customer satisfaction. Most online users are affected by rented computing services directly or indirectly. Since 2010, Amazon's web stores have been run on AWS. As the processor core count increases and the shift to cloud computing accelerates, the sharing problem takes on the significance of societal-scale resource allocation.

A dilemma of sharing is whether to maximize the overall performance or to regulate the quality of service. When running on the same type of hardware, should an application maintain the same level of performance or should it adapt depending on the co-run group? Amazon customers are encouraged to measure the performance of their applications before renting computer time. The tacit assumption is that the performance will be consistent. Otherwise, a customer may be puzzled if the performance measured in testing varies greatly or the performance of a rented run falls far below that of the test run.

Many studies have examined fair cache allocation among independent programs. Xie and Loh borrowed the labels of ecopolitics and called the equal cache partition *communist* allocation, free-for-all sharing *capitalist*, and the optimization *utilitarian* [14]. More formal conditions were later established using the concepts of the game theory including sharing intensive, envy freeness and Pareto efficiency [2, 15].

It is a well-defined optimization problem to allocate the cache to minimize the total miss ratio. Stone et al. solved it assuming that miss ratio curves are convex [8]. Our recent solution uses dynamic programming to optimize for all types of programs [1].

In this study, we define a new type of cache sharing policy called *elastic baseline*. A baseline is a lower-bound on cache allocation for an individual program. The baseline is elastic in that it can be adjusted up or down to encourage or curtail optimization.

We consider the following baseline and two types of elastic baseline.

- *The strict baseline*. The program runs with $1/p$ of the cache, where p is the number of programs sharing the cache.

- *Elastic miss ratio baseline (EMB).* The program's predicted miss ratio may increase by up to some percentage, e.g. 5%, compared to what it would be with the strict baseline.
- *Elastic cache space baseline (ECB).* The program may yield at most a given percentage of its cache space to peer programs.

Our study is a novel combination of elasticity and optimization. In comparison, previous work either does not provide a baseline guarantee, e.g. in optimal caching, or does not optimize, e.g. in capitalist allocation and Pareto efficiency.

The paper makes two main contributions:

1. *Formalism.* The definition and optimization of two types of elastic baseline.
2. *Experimentation.* The performance and resource gain and loss under baseline optimization, with different levels of elasticity.

This paper addresses cache allocation as a sub-problem of cloud computer resource sharing. The performance is measured in miss ratio, which makes the evaluation results direct (based only on the caching effect) and CPU independent. The miss ratio does not fully determine execution time. The effects of prefetching and memory-bandwidth contention are equally important. However, the miss ratio measures the effect of cache allocation directly. It provides a well defined problem for optimization. Current hardware does not fully support cache partitioning, so it is impossible to actually measure the execution time. The current hardware supports cache sharing. Our earlier empirical results have shown that the predicted miss ratio[1] (which we use in this paper) has a near-linear correlation with the co-run slowdown [10]. In contrast, the measured miss rate (misses per second) does not provide such correlation.

2 Elastic Cache Utility

2.1 Background and Motivation

Amazon EC2 lets a user rent different types of computer instances. Three of the types — burstable performance (T2), balanced (M3), compute optimized (C4) — have subtypes where a user rent 1, 2 or 4 vCPUs. There are multiple purchasing options including on-demand instances and spot instances. An on-demand instance has a fixed price and can be rented at any time. A spot instance is dynamically priced based on the system load. A user bids for a spot instance, and the bid is accepted when the spot price falls below the user bid. The spot price can be much lower than the on-demand price. Through these options, Amazon can pool and run applications from multiple users on the same multicore processor, sharing the last level cache. Through the spot

[1] As described below, this miss ratio is predicted based on the Higher Order Theory of locality.

market, there is an inexhaustible supply of user tasks when Amazon has spare cores, and it chooses to use them by running spot instances.

Another provider, Jelastic, sells resource by how much a customer uses per hour. The unit is called Cloudlet. It counts both CPU and memory usage. The CPU usage is the average number of CPUs used over the hour.

In these cases, the price implies that the same type CPU has the same cache resource whenever it is rented, because the price for it is the same. Jelastic charges a partial price if a CPU is only partially utilized. However, the price is still the same for the part of the time the CPU is used. There is still the implicit agreement is that the cache resource is the same for the period of time whenever the CPU is used.

Consistency clashes with optimization. Taking equal cache partition as the baseline, optimal cache partitioning reduces the total miss ratio of a co-run group but may increase the miss ratio of individual programs. In the experiments we will present in Section 3, we found that under optimal cache partitioning, the average miss-ratio reduction for all programs was 131%, but the worst-case increase in a single program was 32 times.

Uneven optimization also happens naturally when programs share the cache, as is the case on current cloud computers. Our experimental data show 102% average reduction for all but 69 times increase for one. It may be argued that a service provider should ensure that such slowdown does not happen without the customer knowledge and consent.

Next we present the new technique of baseline optimization. We will build on our recent research on locality analysis [10] and optimizing [1].

2.2 Rochester Elastic Cache Utility (RECU)

Rochester Elastic Cache Utility provides an algorithm to choose a cache partitioning for a set of programs in order to optimize the predicted cache performance, but with some fairness concessions based on a baseline of each program having an equal partition. With a miss-ratio baseline, the amount of cache a program can give up is limited by the miss-ratio penalty it is expected to incur due to this. With a cache-allocation baseline, each program can only give up a small fraction of its cache.

Slowdown-Free Miss Rate (SFMR) An ideal cache partitioning policy would minimize the overall miss rate for the whole group of programs (i.e. the sum of the programs' individual miss rates). However, while we can measure a program's access rate on a real machine, it is difficult to predict how the access rate will change with with different cache allocation and memory bandwidth (this would be equivalent to predicting its run-time). In order to have a well-defined optimization problem, we define the *slowdown-free miss rate* below. But first we must clarify the relationship between the miss rate, access rate and miss ratio.

Miss Rate and Group Miss Rate The *miss rate* of a program is defined as its average number of misses per time. As a function of the memory access rate AR and miss ratio mr, the miss rate is $MR = AR * mr$. For a group of programs, the *group miss rate* is the sum of individual miss rates:

$$MR_{group} = \sum_i MR_i. \tag{1}$$

Slowdown-Free Miss Rate (SFMR) The *slowdown-free miss rate* of a program is defined similarly to the miss rate[2], as a function of the original access rate AR and the predicted miss ratio mr_{pred}:

$$SFMR = AR * mr_{pred}. \tag{2}$$

The SFMR gives a way to compare the contribution of misses from each program to the group without knowing their slowdown by essentially assuming there is none. *The quantity we are minimizing is the sum of the SFMRs in a co-run group:*

$$SFMR_{group} = \sum_i SFMR_i. \tag{3}$$

Optimal Partition The Higher Order (or "Footprint") Theory of locality provides a way to predict the miss ratio curve of a program based on an online analysis of its memory-access trace [13]. We use an $O(PC^2)$ time and $O(PC)$ space dynamic programming algorithm to find the optimal partitioning for a set of P programs on a cache with C blocks that can be allocated to any program. The algorithm starts with one program, and adds the rest one-by-one. For each added program, it chooses the partition size that minimizes the total miss count for the whole group, computed using the optimal partitioning of the remaining cache to the rest of the programs. The algorithm is described by Brock et al. [1].

Baseline-Optimal Partition The above algorithm was previously modified to include a guarantee to each program that it will be given enough cache so that its miss ratio does not exceed a certain amount. We define two options that allow for relaxation of that constraint, and one new constraint:

1. Elastic Miss-Ratio Baseline (EMB) With a miss-ratio baseline (i.e. lower-bound), the algorithm only accepts partitions for which each program has a predicted miss ratio of no more than it would have with an C/P of the cache (call this miss ratio mr_{equal}). By relaxing this requirement to allow a program's miss ratio to be some percentage higher than mr_{equal}, we can improve the overall miss ratio and get closer to the optimal partitioning

[2] It is also similar to the *common-logical time miss ratio* defined by [10], which specifies each co-run program's miss ratio scaled by the interleaved memory accesses from other programs.

while maintaining a reasonable minimum predicted cache performance for each program. If the miss ratio is allowed to increase by x% over mr_{equal}, we call this x%-EMB (e.g. 5%-EMB).

2. Elastic Cache-Allocation Baseline (ECB) Another option is to set a baseline in cache allocation. For x%-ECB, every program is required to have at least $(100 - x)\%$ of C/P of the cache. For example, 0%-ECB is the same as an equal partitioning, but 5%-ECB allows any program to be given up to 5% less of the cache in order to optimize cache performance for the group. 100%-ECB is the same as an optimal partitioning.

Note that EMB is in a sense a more restrictive policy. The miss ratio can increase by any percentage (a 50% miss ratio can increase by 100%, a 10% miss ratio can increase by 1000%, and so on). ECB is less restrictive in that there is a maximum cache-space concession of 100%, so the ratio of allowed degradation to possible degradation is necessarily smaller than for most cases in EMB.

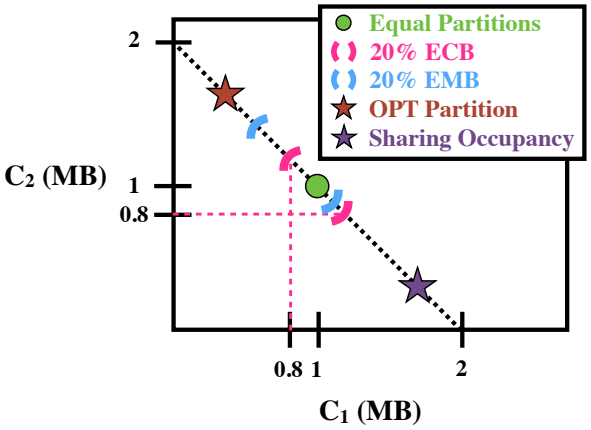


Fig. 1: A toy example of the allowed partition sizes C_1 and C_2 for a pair of programs sharing a 2 MB cache. The black dashed line $(C_1 + C_2 = 2)$indicates the possible partitions. The green dot indicates the equal partitioning, and the parentheses around it indicate the solution spaces for a 20% Elastic Cache-Allocation Baseline (pink) and a 20% Elastic Miss-Ratio Baseline (blue). The red star indicates the optimal partitioning, and the purple star the cache occupancies when the cache is shared.

Fig. 1 shows a toy example to demonstrate how our search spaces compares to the search space of uninhibited partitioning. The dashed line shows the possible partition sizes for each C_1 and C_2. For 20%-ECB (20%-EMB), the search space is limited to the region encased by the pink (blue) parentheses. If program 1 has a larger working set, or accesses cache more frequently, it

may hog the cache but not benefit much from that cache space (purple star). Therefore, the optimal partitioning would be to give program 2 more of the cache (red star). But this may be unfair to program 1. EMB and ECB allow us to find the optimal partition among a smaller set of options with the "baseline" fairness constraint.

3 Evaluation

This section evaluates the benefits of RECU and compares it with alternative methods.

Methodology We predict co-run performance for 16 programs (arbitrarily chosen) from SPEC 2006: *perlbench, bzip2, mcf, zeusmp, namd, dealII, soplex, povray, hmmer, sjeng, h264ref, tonto, lbm,omnetpp, wrf, sphinx3*. We use the first reference input provided by SPEC. For co-run groups, we enumerate all 4-program subsets, which gives us 1820 groups. We model their co-run miss ratio with 8MB of shared cache. The shared cache is partitioned at the granularity of 1024 128-block (8KB) units.

For each program, we measure the average footprint [12] and the access rate (number of accesses per second in a solo execution). We then use the higher-order theory of locality (HOTL) to compute the miss ratio of any program group in any size cache, including the total and individual miss ratio in shared and in partitioned cache [13].

For each set of 4 programs, the total number of partitions is $\binom{1027}{3}$, or over 180 million. We use an optimization algorithm by Brock et al. [1] to choose among all valid partition choices the one that minimizes the sum of the slowdown-free miss rates.

Current technology can support cache partitioning. IBM processors provide hardware support to partition resources including the cache. Intel has recently added cache-partitioning support in Haswell processors, named as Cache Allocation Technology. However, the information about such support is limited. In this study, we do not experiment on real hardware.

We do not simulate system performance for several reasons. First, our purpose and expertise both lie in program analysis and optimization, not hardware cache design. Second, our goal is not to maximize performance, which depends on more factors than we can rigorously study in one paper, but to minimize an indicator of poor performance. By focusing on cache and the slowdown-free miss rate, the achieved optimality is actually CPU independent, i.e. optimal regardless of the CPU used.

A simulator, especially for a multicore system, has many parameters. We are not confident that we can produce the same accuracy that we can with locality modeling. Finally, simulation is slow. Most computer architecture studies simulate a small fraction of a program. For example, Hsu et al. used a cache simulator called CASPER to measure the miss ratio curves from cache sizes 16KB to 1024KB in increments of 16KB [3]. They noted that "miss rate

errors would have been unacceptably high with larger cache sizes" because they "were limited by the lengths of some of the traces." In our analysis, we consider the whole trace of a program execution.

Validation Optimization must be built on theory, which in this case is the higher-order theory of locality (HOTL) [13]. Three recent studies provide independent validation. Wang et al. tested the analysis on program execution traces for CPU cache [10], Hu et al. on key-value access traces for Memcached [4], and Wires et al. on disk access traces for server cache [11]. The three studies re-implemented the footprint analysis independently and reported high accuracy through extensive testing. Hu et al. tested the speed of convergence, i.e. how quickly the memory allocation stabilizes under a steadystate workload, and found that optimal partition converges 4 times faster than free-for-all sharing [4]. Recent work also explored HOTL like statistics in memory allocation (object liveness instead of object locality) [5]. Finally, Wang et al. showed strong correlation (coefficient 0.938) between the predicted miss ratio and measured co-run speed [10]. The correlation means that if we minimize the miss ratio in shared cache using RECU, we minimize the execution time of co-run programs.

3.1 Overall Comparison

From a provider's perspective, RECU is designed to improve efficiency while guaranteeing a baseline performance. The baseline is the lower bound performance specified by the upper bound on the worst-case degradation compared to equal partitioning. Table 1 shows the overall and individual program performance when using different baselines.

RECU improves performance without the risk of pathological worst cases incurred by other techniques. If we look at the QoS results for individual programs in Table 1, we see that RECU keeps the bounds on the performance loss. The worst case loss is bounded by the given threshold, from 0% to 100%, for all programs. When the threshold is 0% worst-case loss, we call it strict RECU. Indeed, the table shows that no single program incurs more misses than equal partitioning. In contrast, the two alternatives, optimal caching and free-for-all sharing, have much worse worst cases. The largest miss ratio increase is 316 times and 687 times in optimal and free-for-all sharing. The largest cache-space loss is 99% in both cases.

Table 1 is arranged with two sides. The left hand side shows the average and median overall miss ratio reduction for co-run groups under each policy. This represents the efficiency of a policy (that is, how close it is to achieving the optimal miss ratio), and should be of interest to service providers. The right hand side shows the fraction of programs that suffer worse than 0, 5 and 10% loss, miss ratio increases for EMB and cache space decrease for ECB, and the worst single loss. This data should be of interest to consumers who want

Table 1: Comparison of RECU methods, optimal caching, and free-for-all cache sharing. The average miss ratio reduction (overall improvement) versus the individual miss ratio increase (individual degradation) are relative to equal partitioning. Two RECU methods are implicit in ECB: 0% cache space baseline is the same as strict, and 100% is the same as optimal.

Methods of cache allocation	Overall performance improvement		Individual loss			
			Percent prog. degraded by			Worst loss
	Avg	Median	$> 0\%$	$\geq 5\%$	$\geq 10\%$	
RECU with elastic miss ratio baseline (EMB)						
0%	5.67%	0%	0%	0%	0%	0%
5%	34.5%	6.22%	61.2%	0%	0%	4.99%
10%	42.6%	8.53%	58.6%	42.4%	0%	9.99%
20%	48.5%	11.1%	58.8%	43.7%	37.2%	19.9%
50%	51.9%	13.7%	59.1%	52.3%	46.0%	49.9%
100%	53.4%	18.3%	59.3%	53.3%	47.0%	99.7%
RECU with elastic cache space baseline (ECB)						
5%	2.20%	1.09%	69.1%	0%	0%	4.68%
10%	6.31%	2.69%	70.8%	68.7%	0%	9.76%
20%	16.4%	5.02%	65.9%	65.2%	65.0%	19.9%
50%	46.2%	8.40%	60.7%	57.4%	55.7%	50.0%
alternatives to RECU (2 types of losses: miss ratio and cache space)						
optimal caching	56.7%	30.4%	59.1%	53.2%	45.7%	31623%
			59.0%	57.7%	56.8%	98.8%
free-for-all sharing	50.5%	17.0%	63.9%	58.3%	52.9%	68735%
			64.0%	62.9%	61.6%	98.9%

quality of service guarantees. The percentiles on group performance are shown later, in Table 2.

RECU's gain in efficiency is significant, as shown on the left side of Table 1. Strict RECU improves the average efficiency by 6% on average. The improvement increases as the bound increases. For the same percentage-bound, the miss ratio concession leads to a greater improvement than the cache space concession. The average gain is much greater than the median gain, showing that some programs have very large gains. For a service provider, the average gain is important since it corresponds to the system throughput.

The increase in efficiency is not linearly proportional to the degree of concession. For EMB, the average performance increases by 29% when moving from no concession to 5% concession but only 1.5% from 50% concession to 100% concession. For ECB, the average performance increases by 30% when moving from 20% to 50% concession but only 2.2% from no concession to 5% concession.

RECU techniques maintain strict monotonicity in both efficiency gain and worst-case loss — the larger the concession is, the greater the overall performance and the worse the possible individual loss. As Table 1 shows, every increase in the amount of concession leads to a greater gain in efficiency, and more programs see degradation (over the baseline). Such monotonicity does not come naturally; If we compare the two alternatives, optimal caching has

Table 2: The reduction in group miss ratio for all tested groups. The reduction is computed as $(base - opt)/base$. All methods reduce the group miss ratio for all groups (100%) except for "sharing", which improves 88% of all groups but degrades the remaining 12%, by -34% in the worst case.

allocation methods	Pct. of co-run groups whose group miss ratio is reduced by					
	≥5%	≥10%	≥20%	≥30%	≥50%	≥ 70%
RECU miss ratio bound						
0%	14.6%	10.3%	5.82%	4.84%	1.65%	0%
5%	54.4%	40.7%	22.6%	17.9%	8.63%	3.79%
10%	60.4%	46.8%	27.3%	22.4%	12.7%	5.93%
20%	66.9%	52.3%	34.0%	28.6%	14.2%	7.14%
50%	76.1%	60.1%	40.8%	35.1%	15.2%	7.64%
100%	80.3%	66.5%	47.3%	41.1%	15.3%	7.91%
RECU cache space bound						
5%	6.75%	3.07%	0.65%	0%	0%	0%
10%	32.1%	21.0%	8.24%	0.65%	0%	0%
20%	50.1%	28.6%	22.4%	15.1%	5.66%	0%
50%	60.5%	46.3%	26.4%	23.2%	13.7%	7.42%
alternatives to RECU						
optimal	87.0%	74.4%	56.8%	50.1%	15.9%	8.30%
sharing	74.1%	60.8%	46.1%	39.8%	11.1%	7.47%

a greater efficiency overall, but its worst individual losses are slightly better than those of free-for-all sharing.

This monotonicity — more concession means higher performance and greater worst individual loss — should be expected and welcomed by the provider and the customer of an online service. It is the third reason that RECU is valuable, after the first two we just saw: the maximal overall gain and the bound on the individual loss. All three benefits are the result of the optimization by RECU.

3.2 Group Performance Improvement

The group performance is improved for most of the co-run groups. Table 2 shows the percentage of co-run groups whose slowdown-free miss rate is reduced by RECU, as compared to equal partitioning. At the 5% miss ratio bound, over 54% of groups are improved by 5% or more and 8.6% by 50% or more.

The miss-ratio bound enables greater optimization than the cache-space bound does. The 10% miss ratio bound enables a similar improvement as the 50% cache space bound.

Baseline optimization can obtain the improvement of free-for-all sharing, which is shown at the bottom of Table 2. The 50% miss-ratio bound shows a similar improvement as free-for-all cache sharing. The difference is that with RECU, no program will lose performance beyond the baseline, but free-for-all sharing provides no such guarantee. In Table 1, we have shown that the worst-

Table 3: Access rate, miss rate and slowdown-free miss rate for each program. Combined with Fig. 2 and Fig. 3, winners are colored **green**, losers are colored red, and others are colored black. The programs are sorted by slowdown-free miss rate because that is the metric that directly affects speed.

Benchmark	Access rate (million)	Miss ratio	Slowdown-free miss rate(million)
lbm	12,727	6.32%	804
zeusmp	10,758	1.15%	124
sphinx3	2,754	4.24%	116
mcf	856	11.0%	94.2
wrf	5,292	1.35%	71.2
soplex	449	7.44%	33.4
dealII	3,029	0.66%	19.8
omnetpp	592	3.24%	19.2
hmmer	2,640	0.11%	3.01
tonto	3,778	0.08%	2.86
sjeng	1,497	0.11%	1.61
bzip2	367	0.32%	1.18
namd	1,915	0.04%	0.74
perlbench	56	0.90%	0.51
h264ref	427	0.08%	0.32
povray	3,249	0.00005%	0.001

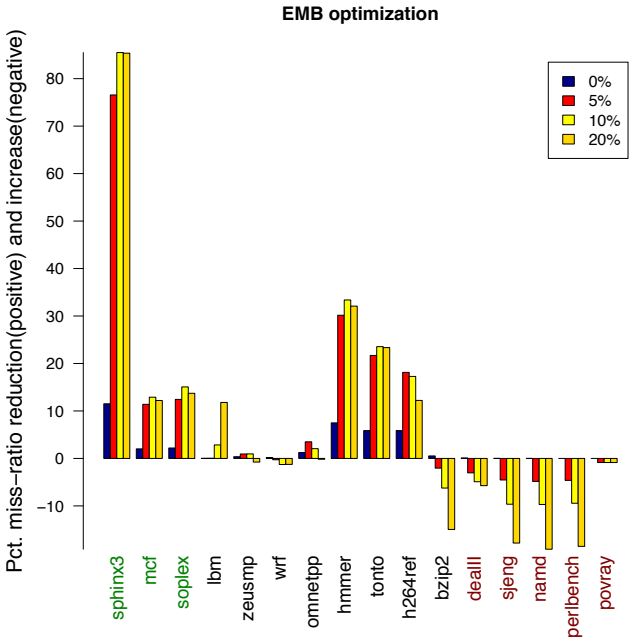
case degradation is 49.9% for 50% EMB but about 69,000% for free-for-all sharing. Hence, baseline optimization is strictly better.

3.3 Individual Comparison

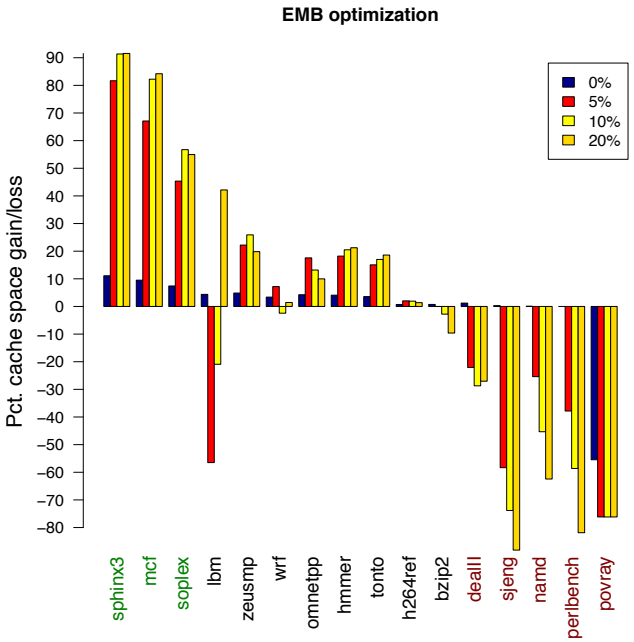
Table 3 shows a breakdown of programs into three categories: winners, losers, and others. The categories are based on the average results across every co-run group, as shown in Fig. 2 and Fig. 3, and are described in detail below.

Winners Winners gain cache and have their miss ratio decreased in every concession scenario, and generally have high equal-partition miss ratios. Notably, *lbm* also has a high equal-partition miss ratio, but is not a winner. This is a result of its stair-step shaped miss ratio curve: since it is usually nearly constant, *lbm* typically has low cache utility. This is also showed by Fig. 2, *lbm* gains little in 0%-EMB, then loses cache in 5% or 10%-EMB until 20%-EMB.

Losers Losers lose cache and have their miss ratio increased in every concession scenario, and generally have a low slowdown-free miss rate. There are three exceptions to that generalization: *bzip2* and *dealII, h264ref*. *bzip2* gains in 5%-EMB, but loses in all other cases. Furthermore, it only gains less than 3%. Thus *bzip2* behaves in general like a loser. *dealII* is a loser despite its large SFMR. It loses cache due to its nearly constant miss ratio, which means it has lower cache utility. In other words, it is not sensitive to cache size, so it tends to be robbed of cache. *h264ref* is cache sensitive, thus it gains improvement in

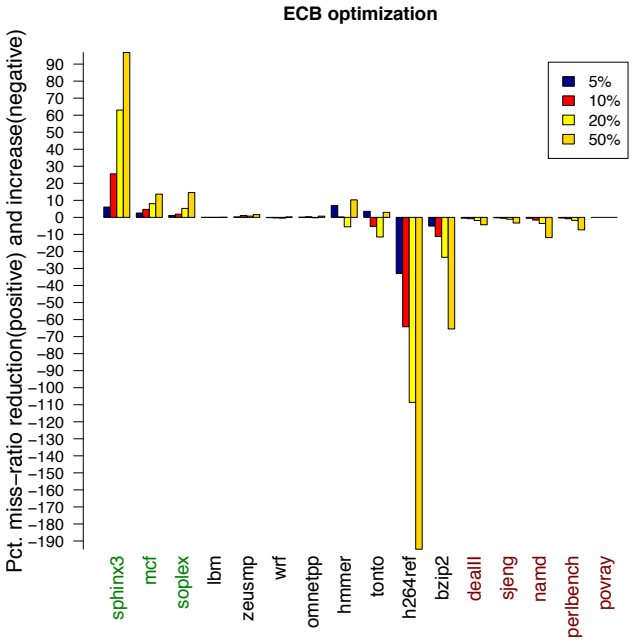


(a) Miss ratio reduction (positive) vs. increase (negative)

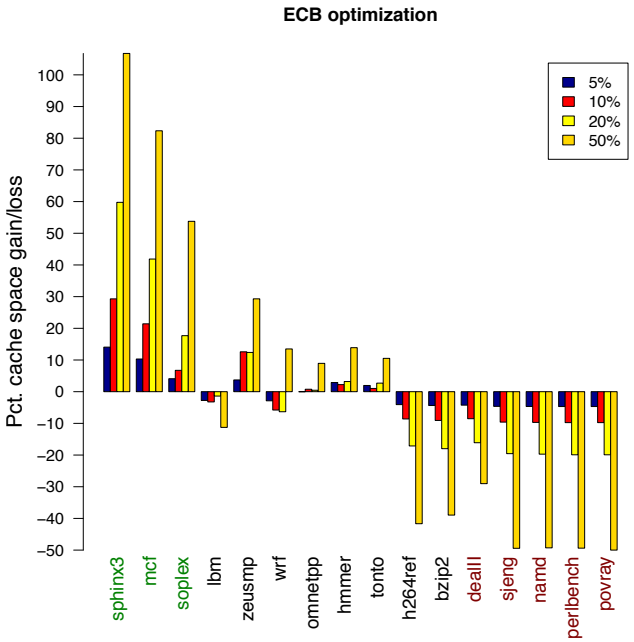


(b) Cache space gain vs. loss

Fig. 2: Effect of elastic miss ratio baseline optimization (a,b) in each of the 16 test programs, measured by the miss ratio reduction/increase (a) and cache space gain/loss (b) for each program, averaged across all its $\binom{15}{3}$ co-run appearances. The programs are ordered by the decreasing y-axis value. They are ranked by performance gain in (a) and cache space gain in (b).



(a) Miss ratio reduction (positive) vs. increase (negative)



(b) Cache space gain vs. loss

Fig. 3: Effect of elastic cache space baseline optimization (a,b) in each of the 16 test programs, measured by the miss ratio reduction/increase (a) and cache space gain/loss (b) for each program, averaged across all its $\binom{15}{3}$ co-run appearances. The programs are ordered by the decreasing y-axis value. They are ranked by performance gain in (a) and cache space gain in (b).

some cases. However, it has less much potential due to low miss SFMR. Thus when other programs can benefit from more cache, its cache will be robbed.

Others Others do not fit into either of the above neat categories. However, all of the *others* in our experiments eventually gain more cache with more relaxed constraints. For example, the optimization allocates more to *tonto* with 10% and 20%-ECB, but less in all cases of EMB. Our experiments also show, paradoxically, that more cache space on average does not lead to miss ratio reduction on average. For example, *lbm* loses cache in 10%-ECB, but has reduction on miss ratio. *zeusmp* gains more cache but increases miss ratio. The cause is related to the stair-steps in its miss ratio curve. *lbm* either loses cache and pays a small penalty in misses, or gains cache and has its miss ratio significantly reduced. In most cases of 10%-ECB, there is not enough cache for *lbm*. *zeusmp* is similar but in a reverse situation.

3.4 Analysis Overhead

Online Locality Analysis Recent research has made it possible for *in vivo* analysis with several techniques. First, a metric called footprint measures the average working-set size [12]. The footprint measurement has linear-time complexity and can be performed online by sampling. Second, the footprint is used to compute the miss ratio curve [13].

We wrote the dynamic programming algorithm in C++, and the scripts in Ruby. We tested the speed of analysis on a 1.7GHz Intel Core i5-3317U (11 inch MacBook Air, with the power cord unplugged). On average, footprint sampling can be done in 0.09 second per program using a technique called *adaptive bursty footprint (ABF) sampling* [10]. Based on that, we computed the optimal cache partitioning for each group of four in 0.21 second on average.

4 Related Work

Optimal Cache Partitioning Cache partitioning is an intuitive way to improve fairness as well as performance. Stone et al. [8] proposed a greedy algorithm for N programs, which is optimal for programs with convex miss ratio curves. They start with each program having only one cache unit, then allocate cache greedily. Suh et al. [9] relaxed the convexity assumption by dividing miss ratio curves between non-convex points.

Our study employs a dynamic programming algorithm developed by Brock et al. [1], whose optimality does not require miss ratios being convex. The optimization handles general types of baselines or their mixtures. For example, it can optimize cache partition when one customer uses 5%-EMB, and another uses 10%-ECB.

Fair Cache Partitioning There have been a plethora of work to partition the cache among multiple threads and programs. A recent technique is cache rationing, which protects the cache ration of each program and at the same time finds unused ration to share among the co-run programs [7]. Our technical report contains references to many earlier, heuristics-based solutions [6]. More formal conditions have been established using the concepts of the game theory including sharing intensive, envy freeness and Pareto efficiency [2,15]. Our approach uses the elastic baseline for fairness, which is straightforward from the customer's perspective, while maximizing the overall performance for the service provider.

5 Conclusion

In this paper, we have described two elastic baselines for fair and efficient cache allocation: *elastic miss ratio baseline* and *elastic cache space baseline* to increase average performance by sacrificing a small amount of fairness. We employed a dynamic programming algorithm to minimize overall slow-down miss rate for each baseline. We have evaluated equal and optimal partitions, 6 levels of EMB, and 4 levels of ECB.

The results show that both ECB and EMB can significantly improve the total performance of a group while limiting the worst loss for every individual. For the same percentage of allowed concession, EMB is more effective than ECB. The effect of RECU is montone — larger concession means greater performance for the group and worse possible loss for the individual.

Lastly, we classified individual programs into three categories. Our results show that lower slowdown-free miss rate usually leads to performance degradation and higher miss ratio usually results in improvement. Furthermore, we noted the seeming-paradox that programs which on average sacrifice cache can also gain performance on average.

References

1. Brock, J., Ye, C., Ding, C., Li, Y., Wang, X., Luo, Y.: Optimal cache partition-sharing. In: Proceedings of ICPP (2015)
2. Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., Stoica, I.: Dominant resource fairness: Fair allocation of multiple resource types. In: Proceedings of NSDI (2011). URL https://www.usenix.org/conference/nsdi11/dominant-resource-fairness-fair-allocation-multiple-resource-types
3. Hsu, L.R., Reinhardt, S.K., Iyer, R.R., Makineni, S.: Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In: PACT, pp. 13–22 (2006)
4. Hu, X., Wang, X., Li, Y., Zhou, L., Luo, Y., Ding, C., Jiang, S., Wang, Z.: LAMA: Optimized locality-aware memory allocation for key-value cache. In: Proceedings of USENIX ATC (2015)
5. Li, P., Ding, C., Luo, H.: Modeling heap data growth using average liveness. In: Proceedings of ISMM (2014)
6. Parihar, R., Brock, J., Ding, C., Huang, M.C.: Protection, utilization and collaboration in shared cache through rationing. URL http://www.cs.rochester.edu/u/cding/Documents/Publications/tr-ration.pdf

7. Parihar, R., Brock, J., Ding, C., Huang, M.C.: Protection and utilization in shared cache through rationing. In: Proceedings of PACT, pp. 487–488 (2014). DOI 10.1145/2628071.2628120. URL http://doi.acm.org/10.1145/2628071.2628120. *short paper*
8. Stone, H.S., Turek, J., Wolf, J.L.: Optimal partitioning of cache memory. IEEE Transactions on Computers **41**(9), 1054–1068 (1992). DOI http://dx.doi.org/10.1109/12.165388
9. Suh, G.E., Rudolph, L., Devadas, S.: Dynamic partitioning of shared cache memory. The Journal of Supercomputing **28**(1), 7–26 (2004)
10. Wang et al.: Optimal program symbiosis in shared cache. In: Proceedings of CCGrid (2015)
11. Wires, J., Ingram, S., Drudi, Z., Harvey, N.J., Warfield, A., Data, C.: Characterizing storage workloads with counter stacks. In: Proceedings of OSDI, pp. 335–349. USENIX Association (2014)
12. Xiang, X., Bao, B., Ding, C., Gao, Y.: Linear-time modeling of program working set in shared cache. In: Proceedings of PACT, pp. 350–360 (2011)
13. Xiang, X., Ding, C., Luo, H., Bao, B.: HOTL: a higher order theory of locality. In: Proceedings of ASPLOS, pp. 343–356 (2013)
14. Xie, Y., Loh, G.H.: Dynamic classication of program memory behaviors in CMPs. In: CMP-MSI Workshop (2008)
15. Zahedi, S.M., Lee, B.C.: REF: resource elasticity fairness with sharing incentives for multiprocessors. In: Proceedings of ASPLOS, pp. 145–160 (2014)