Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2019) 27: 4269 – 4283

doi:10.3906/elk-1903-183

A Fine-grain and scalable set-based cache partitioning through thread classification

İsa Ahmet GÜNEY*, **Gürhan KÜÇÜK**
Department of Computer Engineering, Faculty of Engineering, Yeditepe University, Istanbul, Turkey

Received: 29.03.2019 • **Accepted/Published Online:** 08.07.2019 • **Final Version:** 26.11.2019

Abstract: As contemporary processors utilize more and more cores, cache partitioning algorithms tend to preserve cache associativity with a finer-grain of control to achieve higher throughput and fairness goals. In this study, we propose a scalable set-based cache partitioning mechanism, which welds an allocation policy and an enforcement scheme together. We also propose a set-based classifier to better allocate partitions to more deserving threads, a fast set redirection logic to map accesses to dedicated cache sets, and a double access mechanism to overcome the performance penalty due to a repartitioning phase. We compare our work to the best line-grain cache partitioning scheme that is available in the literature. Our results show that set-based partitioning improves throughput and fairness by 5.6% and 4.8% on average, respectively. The maximum achievable gains are as high as 33% in terms of throughput and 23% in terms of fairness.

Key words: Cache memory, partitioning algorithms, multicore processing

1. Introduction

The benefits of shared cache partitioning in multicore systems are very well known to the architecture community. There are many approaches that have already been proposed for both how to allocate resources among cores [1–3] and how to enforce those allocation decisions [1, 4–6]. Cache monopolization and cache thrashing are the most prominent dangers in an uncontrolled shared-cache environment. Static partitioning, in which all cores receive an equal and constant amount of resources, is an option for preventing such problems. However, it may lead to resource underutilization, and both the throughput and fairness of the system significantly get worse. A good partitioning scheme must be able to detect the resource needs of all cores and apply appropriate allocation decisions to yield better results in terms of performance, energy, energy-delay, or fairness metrics.

As applications tend to change program phases at run-time, their cache behavior also changes. Both allocation and enforcement schemes must be adaptive enough to keep up with the changes on cache demands of cores. Some mechanisms gradually adapt to new allocation decisions [4–6], whereas some of them adapt instantly [1].

The interplay between scalability and granularity of partitioning mechanisms is another topic of great importance. Naturally, a mechanism that can allocate smaller cache regions can also support more partitions. Scalability is an important aspect not only because modern processors keep integrating more and more cores but also for enabling other performance enhancement mechanisms such as Talus [7], which requires the number of partitions to be doubled.

*Correspondence: iguney@cse.yeditepe.edu.tr

Although many novel cache partitioning schemes have been proposed in the past, partitioning a shared cache in terms of sets has been rarely focused. While it seems like a very simple and straightforward approach, it brings a few obstacles that suddenly render it almost inapplicable. The first, and a considerably large, obstacle becomes the set index computation problem. In traditional caches, the set index is computed by simply applying a fixed bit-mask to addresses, exploiting the fact that the number of sets in caches is always a power of two. With the set partitioning, however, the number of sets that are allocated to a core may not always be a power of two, and this introduces a problem of set index calculation in such cases. As allocation decisions change, size and location of a cache region mapped to a core change as well. This immediately triggers the second and third problems: What would be the performance impact of changing size and location of a cache space allocated to a core due to older data becoming inaccessible, and how would the system guarantee cache coherency?

Though it introduces such tough challenges, set-based partitioning is also very adVantageous on many recently-popular terms: First, it is much more scalable compared to a way-partitioning scheme, since the number of sets is always much greater than the number of ways. As a result, with set partitioning, we can generate a large number of cache partitions with no difficulty. Secondly, cache sets hold fewer cache lines compared to cache ways. As a result, a finer grain of allocation/deallocation scheme might be implemented by a set-partitioning scheme. Last but not least, a set-based partitioning scheme does not harm the associativity level of a cache. Highly associative caches are especially good at fighting conflict misses, and preserving their level of associativity has great importance for high cache performance.

Our contribution in this study is the introduction of a set-based partitioning mechanism, which proposes a classifier-based allocation and a strict allocation enforcement policy. The resulting mechanism preserves maximum possible cache associativity, high scalability, better fairness and performance results. We provide a set of possible solutions and analysis for major problems, which exist when a shared cache is partitioned in terms of sets (i.e. cache coherency, the latency of modulo operator for cache sizes that are not powers of two, and finally, pseudo cache flushes.) We also present our results by comparing against the current state of the art.

The rest of the paper is structured as follows: Section 2 gives an account of the related work. Section 3 presents our enforcement mechanism, set partitioning, and related submechanisms. Section 4 introduces our allocation policy called set classifier. Section 5 describes our experimental methodology for evaluation. Section 6 presents and discusses simulation results. Finally, Section 7 concludes.

2. Related work

Earlier work in cache partitioning focus on partitioning a shared cache among different reference streams of a single application, or among applications which run sequentially in a single processing core [8, 9]. With the increased availability of multicore processors, work in this field shifted towards partitioning the cache among multiple concurrently running applications.

Brock et al. showed that the optimal partitioning-only solution is the optimal partition-sharing solution, and the problem of sharing partitions among programs can be reduced to the problem of partitioning-only [10]. Partitioning a shared cache can be done either implicitly by altering the line replacement policy [4, 11, 12] or explicitly by restraining the number of cache lines an application can have [1, 5, 6, 13–15]. Explicit cache partitioning techniques require allocation policies [1, 3, 16, 17] to determine how much cache space each partition should have. Recently, El-Sayed et al. present a partition-sharing mechanism which works on a commodity multicore system with a way-partitioning scheme [18].

Most set partitioning techniques use a software approach known as page coloring [19–23], which is a technique of altering the way the operating system allocates memory pages so that pages from different partitions map to different cache sets. Similarly, our set partitioning scheme requires changes in how memory addresses are mapped to cache sets.

Zarandi et al. propose a mechanism in which different sets have different associativity [24]. Sanchez and Kozyrakis utilize hash functions and multilevel search in order to increase effective associativity [25]. Ranganathan et al. examine partitioning the cache based on sets in general terms [26]. Varadarajan et al. describe an architecture where caches consist of small, direct mapped caching units [27].

The closest match to our work is the work by Abousamra et al. [13]. They report a 15% speedup and 14% fairness improvement when compared to the traditional the least recently used (LRU) policy with no partitioning, on the average. Our work differs in the following points: we propose a mechanism for computing set index of incoming requests without implementing a lookup table and a scalable linear allocation policy which can be completed in a single step in contrast to their multiiteration algorithm. We also provide detailed evaluation analysis for each proposed mechanism separately. Finally, we evaluate our work against a more recent partitioning algorithm (Vantage) in a larger configuration with 32 cores (instead of 2 or 4 cores).

3. Set partitioning

Set partitioning is an enforcement mechanism, which partitions the cache by allocating blocks of sets to partitions. In this mechanism, associativity available to a partition is not hindered by other applications. Partitions work with cache regions, which have less number of sets but are completely dedicated to themselves.

At a glance, adVantages of set partitioning are better scalability, granularity, and performance. In set partitioning, the smallest unit which can be allocated to a partition is a set. Caches usually consist of hundreds or thousands of sets, which makes set partitioning a fine-grained mechanism. This also makes set partitioning scalable since there are a large number of sets available for partitioning; thus, set partitioning can support a large number of partitions.

Partitioning the cache in terms of sets allows set partitioning to preserve associativity provided by the hardware. For example, way partitioning halves associativity for each partition in a two-partition configuration, whereas set partitioning does not degrade associativity. Actually, associativity stays constant as the number of partitions increase in set partitioning. Set partitioning also raises some issues as cache coherency, cache flush effect, and the latency of the modulo operator. These issues and our proposed solutions are discussed in the following subsections.

3.1. Set index computation

In traditional caches, the set index of an access is determined by applying a set mask to the memory address after eliminating offset bits. This is equivalent to computing the modulo as shown in Section 3.1. However, since each partition owns a reduced number of sets in set partitioning, the set index is computed using the number of sets allocated to a partition as shown in Section 3.1.

$$set\ index = (memory\ address)\ mod\ (number\ of\ cache\ sets) \quad (1)$$

$$set\ index = beginning\ set_i + (memory\ address)\ mod\ (number\ of\ cache\ sets_i) \quad (2)$$

Beginning set of a partition starts at the end of previous partition's dedicated cache region and can be computed iteratively using Equation (3):

$$Beginning\ set_i = \begin{cases} 0, & \text{if } i = 0 \\ beginning\ set_{i-1} + size_{i-1}, & \text{otherwise} \end{cases}. \tag{3}$$

In traditional caches, lines with different set bits are directed into different sets, and lines which have identical set bits are distinguished by their tag bits. However, when set partitioning is applied, cores may have less number of sets than there is in the cache, causing some lines which have identical tags but different set bits mapping to the same set. To help the cache distinguish different addresses which are mapped to the same set, tag bits are extended to include set bits in our scheme. The additional area requirement due to this extension is somewhat negligible, the inclusion of additional hardware required by set partitioning causes roughly a 2% increase in cache space. However, this increase in area cannot be directly traded away for more cache storage since without a set-based partitioning number of sets should always be selected as powers of two. Similarly, for a 16-way cache, adding one way to the cache means increasing cache area by more than 6%.

3.2. Double access mechanism

Changing a mapping function is equivalent to an immediate cache flush in terms of throughput. Lines may remain in the cache, but most of them, if not all, become effectively inaccessible. In order to mitigate the throughput loss caused by these cache flushes, we propose a mechanism called double access (DA).

In DA, partitions remember their size and beginning set for the previous period in addition to the current one. This allows partitions to be able to locate their valuable cache lines inserted in earlier periods. When a cache access occurs, a partition initially searches a cache line within a set indexed by the current mapping function. If the line is valid and if there is a tag match, it is called a primary hit, otherwise a primary miss. In case of a primary miss, the partition searches the requested line within a set computed by the mapping function of the previous period. If the line is valid and if there is a tag match at the end of this second search, it is called a secondary hit, otherwise a secondary miss. Primary and secondary accesses are carried out sequentially; therefore, access latency of a primary hit is equal to the latency of the cache, whereas the access latency of a secondary hit is equal to twice the latency of the cache. In case of a secondary miss, data is retrieved from a lower level memory structure, as usual.

Secondary accesses allow partitions to address cache lines from other partitions. Partitions are free to evict any line from their dedicated space whenever it is necessary, but there is no need to prevent others from accessing these lines as long as they stay healthy in the cache, especially if others can benefit from them. Indeed, results discussed in Section 6.2 show that there is a possible throughput gain by allowing this via the DA mechanism.

In our current design, to prevent partitions from evicting lines from other partitions and violating allocation decisions, the secondary access only allows accessing cache lines without updating their timestamps, and all insertions are applied into a partition's own space (i.e. using the most up-to-date mapping function). Another interesting design choice might be to update timestamp information of cache lines that give a secondary hit. In such a case, the new owner of those cache lines may get a really hard time evicting them, since they can be instantly moved to the most recently used (MRU) position in a cache set, in each secondary access. Here we prefer giving secondary access as a best-effort service without impeding the performance of the new owner of a partition.

3.3. Ensuring correct computation

When a partition's beginning set or size changes, its mapping function for computing set indices also changes. This may cause lines which are inserted two periods ago to be inaccessible (unless the mapping function happens to be the same with one of the last two repartitioning periods). If not managed properly, such lines will become "lost", i.e. they will still be in the cache but become inaccessible.

Lost lines cause two types of problems which may cause the program to enter a faulty state. If the lost line is a dirty line, the cache will be unable to locate the requested line, and bring the line with an obsolete value from a lower level memory structure. The other case is when the lost line is clean: a clean line may be retained in the cache long enough so that it becomes accessible again after a number of repartition periods, but data in that line may be obsolete.

To prevent such problems, lost cache lines must be avoided at all costs. This can be realized by using an extra one-bit counter per line, called the stale bit. This bit represents whether a cache line will become lost in the next repartitioning period or not. If a cache line is just inserted or is accessed by primary access, it means that it will also be accessible during the next period. Therefore, when a line is inserted or accessed via primary access, the stale bit of that line is reset.

At the end of each period, all lines which will be lost during the next period must be evicted. Similar to a traditional eviction, if the line to be evicted is clean, simply resetting the valid bit is sufficient to evict that line. On the other hand, if the line is dirty, contents of that line must be sent to the lower level unit in the memory hierarchy.

Here, we propose an invalidation/eviction process after each repartitioning period. During this process, all cache lines are sequentially checked, and clean stale lines are invalidated, and dirty stale lines are written back to the main memory. The process ends when all lines are checked, and all write-back operations are completed. All cache accesses are blocked during this period. The impact of this invalidation/eviction process on performance is discussed in Section 6.4.

3.4. Fast set redirection

The modulo operator is an ideal operator for directing incoming cache accesses towards cache sets if addresses are assumed to be uniformly distributed. Since computation of the set index is on the critical path of cache access, the number of sets in caches are always selected in powers of two to allow fast bit masking instead of executing a costly modulo operation. However, in set partitioning, partitions may have sizes that are not powers of two. The modulo operator is significantly slower than the bit masking operation, and it is an unfeasible option for fast set index computation. Considering that cache access latency has a big impact on the processor throughput, the mechanism which replaces modulo operator should be as simple and as fast as possible.

As speed is of utmost importance, we propose utilizing bit masks in order to harness their speed. Two simple approaches, which use bit masks with nonpower of two sizes, exist: rounding a given size up or down to a power of two. If partition sizes are rounded down, some sets would never be indexed and therefore never would be accessed. This approach would require additional mechanisms for deciding which accesses would be directed to that extra space and how accesses would be indexed within that region. On the other hand, if partition sizes are rounded up, some accesses would be indexed to sets which are beyond the extent of a partition and would have to be redirected inside partition boundaries. We propose a simple and fast method called fast set redirection (FSR) for computing set indexes by rounding partition sizes up to a power of two as shown in Algorithm 1.

Algorithm 1 Fast set redirection

set index = (*address*) *mod* (*rounded up partition size*)
if *set index* ≥ *partition size* **then**
  set index = *set index* − *partition size*
end if
set index = *set index* + *beginning set of partition*

Here the proposed set index computation method is not ideal and does not uniformly distribute accesses. Therefore, some sets in a partition are used less frequently than others. The effect of this suboptimal distribution, which is directly caused by our method, on throughput is examined in Section 6.3.

FSR computation requires a one-bit mask, one comparison, one subtraction, and one addition operations. However, computation can be implemented in a much simpler way. First, the comparison can be ignored since comparing two operands is carried out by applying a subtraction and checking the sign bit. Therefore, FSR can apply subtraction operation immediately, and use this value if the sign bit is appropriate. The addition and subtraction operations can be handled in a single move with the 3-in-1 structures described by Vassiliadis et al. [28]. Overall, FSR computation can be carried out with a bit mask operation and two parallel 3-in-1 operations. Given that bit masking and multiplexing are fast operations, we assume that the critical path of FSR (bit masking, 3-in-1 addition, and bit selection) does not necessarily increase cycle time.

4. Set classifier

Cache partitioning mechanisms consist of two major components: an allocation policy, which decides the amount of cache resources each core can receive, and an enforcement policy, which realizes these decisions as faithful as possible. When these policies work in harmony, near-optimal performance gains can become attainable. Since the set partitioning is a set-based enforcement policy, we also need a complementary set-based allocation policy. Here we propose a classification-based policy, which is similar to our previous work that is solely based on way-partitioning [2]. To make allocation decisions about reference streams, our set classifier (SC) classifies them into four predefined classes: null (threads which are not interested in this level of the cache), very harmful (threads which have destructive cache behavior), harmful (threads which are mildly destructive, but also benefit from the cache), and harmless (threads that greatly benefit from this level of the cache). This classification is done periodically in epochs by utilizing run-time statistics. These statistics are the traffic rate, miss rate, and cache decay rate.[1]

Traffic rates of threads are determined by dividing the number of cache accesses in one epoch to the duration of the epoch. Miss rate is determined by dividing the number of cache misses to the number of cache accesses in one epoch. Finally, the cache decay rate is determined by dividing the number of cache decays in one epoch to the number of maximum possible cache decays in one epoch.

The well-known lookahead allocation policy exploits the stack property of LRU to determine the exact contribution of each allocated way in terms of cache hits that are gained [1]. However, the set-based approach lacks such property, and observing the contribution of extra sets on cache hits gained for a particular thread is not as simple as it is done in way-partitioning mechanisms. Here we propose the use of cache decay statistics to access a similar type of information. Cache decay is a metric used to determine whether a given cache set

[1] In our work, we assume a fixed one thread per core configuration. Hence, we refer to reference streams as threads in this section. However, the same allocation policy can be easily applied to reference streams with multiple threads by using their cumulative statistics.

(or line) has been idle for a certain amount of time, and it is mainly used for reducing the energy consumption of caches [29].

To track for how long each set has been idle, per-set decay counters are implemented. Since updating these counters on a per-cycle basis would cause higher complexity and energy dissipation, decay counters are usually implemented in a more coarse-grain fashion. In our setup, 3-bit counters per set are implemented. Whenever a set is accessed its decay counter is reset to a decay reset value. All counters are decremented by one at the end of a smaller period called decay period. In this scheme, a set with a decay value of zero is marked as decayed, and the set preserves its decay status until an access occurs to that set. Maximum possible cache decay represents the case where all sets of a thread are decayed throughout the epoch. This value is calculated using Equation (4).

$$Maximum\ possible\ decay_i\ =\ Number\ of\ sets_i * \frac{Epoch\ duration}{Decay\ period\ length}. \tag{4}$$

To keep future calculations simple, we discretize the collected statistics into 0, 1, and 2 by comparing these values with low and high thresholds and assigning them to traffic (T), miss (M), and decay (D) classes.

The purpose of our set classifier is to classify threads according to their cache behavior: threads with high miss rate and/or high decay rate should be classified as harmful or very harmful, whereas threads with good cache behavior should be classified as harmless, and threads with no interest in cache should be classified as null. After T, M, and D classes are determined, each thread is assigned ThreadValues (TV) using the formula in Equation (5), in order to determine how bad their cache behavior are.

$$TV = T * (1 + M + 2 * D). \tag{5}$$

The traffic class is multiplied by other statistics since the traffic rate of a thread determines how big the impact of the miss rate and the decay rate is. A thread with a high miss rate will be much more destructive if it also has a high traffic rate.

Similarly, it is much worse if a thread has a high decay rate even though it has a high number of cache accesses (i.e. high traffic rate). One may expect traffic rate and decay rate to have a negative correlation, but this may not be the case with a nonuniform access pattern. If a thread has both high traffic and decay rate, it means that the thread is only accessing a subset of cache sets very frequently, while accessing the rest of its sets rarely. A thread with a high decay rate is a better candidate for giving away cache space compared to a thread with a high miss rate. Therefore, we multiply the decay class of threads by 2 in order to punish threads with high decay rates.

If a thread has a low traffic rate, then miss rate and decay rate of that thread become insignificant for the CPU performance. Such threads are classified as null (i.e. no interest in any cache resource), and their TV becomes 0. However, null threads and harmless threads with good cache behavior (M = 0, D = 0) should be distinguished. Hence, a constant value of 1 is added to TV computation. This way, threads with high traffic but low miss/decay rates will have a different score from threads with no cache interest.

Once the TV of a thread is determined, ThreadClass (TC) computation determines which category the thread should fall into (Table 1). Reflecting the theory behind Equation (5), threads with higher TVs are classified into classes with worse cache behavior.

In our set classifier, threads that are classified as null or very harmful are not given any cache resources since null threads do not utilize the cache and very harmful threads do not gain any performance, similar to our

Table 1: ThreadClass of a thread based on its ThreadValue.

	TV = 0	0 <TV <4	4 ≤ TV <6	6 ≤ TV
ThreadClass	Null	Harmless	Harmful	Very Harmful

previous work by Ovant et al. [2]. As a result, the entire last level cache (LLC) is partitioned among harmless and harmful threads. Although it is possible to differentiate the amount of cache space allocated to threads of the same class, it would require dividing these classes into additional subclasses. Thus, we allocate equal cache space to all threads of the same class in order to keep the mechanism simple enough to be implemented in hardware. Since harmless threads are expected to utilize cache resources more efficiently, they are provided with proportionally (twice, in our study) more cache resources than Harmful threads. How much cache resource a thread should receive is calculated according to Equations 6–8.

$$Total\ Weight = 2 * Number\ of\ Harmless\ threads + \ Number\ of\ Harmful\ threads \tag{6}$$

$$U = \frac{Number\ of\ sets\ in\ cache}{Total\ Weight} \tag{7}$$

$$Allocation = \begin{cases} floor(2 * U), & \text{if } thread\ is\ Harmless \\ floor(U), & \text{if } thread\ is\ Harmful \end{cases} \tag{8}$$

Note that the floor function is used in Equation (8), and there might be some leftover sets after partitioning, evidently. Leftovers are given to arbitrary threads (these sets are always given to the last in our tests). In case there are no harmless or harmful threads, all cache resources are equally partitioned among very harmful threads.

5. Experimental methodology

We test our cache partitioner in MacSim simulation environment[2]. Thirty 32-core workloads are created, twenty-five workloads are constructed by randomly selecting four benchmarks and repeating each benchmark eight times, and five workloads are constructed by randomly selecting eight benchmarks and repeating each benchmark four times. Benchmarks are selected from the SPEC2006 benchmark suite [30]. List of workloads used are given in Table 2[3]. All benchmarks are used except perlbench, gcc, and cactusADM, which we had trouble running them on the MacSim environment. When we run each application on a dedicated core, each of them starts running from its own region of interest.

In our configuration, each processor core has private L1 instruction and data caches, and all cores share a single L2 cache. As L1 caches are private, no partitioning is needed there. Our study focuses only on a shared LLC. Simulated processor configuration is shown in Table 3, and partitioning algorithm parameters are given in Table 4. Simulations are carried out for 100M cycles where the warmup duration is 10M cycles and classifier period length is 5M cycles for evaluating individual mechanisms (Sections 6.1 to 6.3). Simulations for obtaining

[2]HPArch research group (2012). MacSim: A CPU-GPU Heterogeneous Simulation Framework [online]. Website https://github.com/gthparch/macsim [accessed 11 June 2019]

[3]Benchmark abbreviations are: astar(A), bwaves(B), bzip2(BZ), calculix(C), deal(D), games(G), gems(E), gobmk(K), gromacs(R), hmmer(H), href264(F), lbm(L), leslie3d(L3), libquantum(Q), mcf(M), milc(I), namd(N), omnetpp(O), povray(P), sphinx(S), sjeng(J), soplex(SO), tonto(T), wrf(W), xalanc(X), and zeusmp(Z).

Table 2: Workloads.

1	T-X-O-W	7	N-J-G-S	13	C-M-F-P	19	A-H-T-D	25	C-M-B-SO
2	T-P-N-X	8	D-R-O-C	14	K-P-G-S	20	SO-H-D-P	26	C-H-M-L3-B-X-SO-K
3	T-K-H-M	9	H-B-X-D	15	T-L3-K-	21	O-M-D-W	27	F-E-BZ-S-A-K-D-L
4	C-X-W-SO	10	F-BZ-K-L	16	B-R-Z-L3	22	N-M-SO-BZ	28	D-K-T-R-H-O-J-C
5	W-S-L-O	11	H-E-R-C	17	D-R-L-G	23	C-T-Z-L3	29	Q-B-BZ-R-BZ-Z-W-L3
6	P-I-C-D	12	C-E-P-L	18	J-T-G-W	24	D-R-X-M	30	W-T-Z-P-I-N-SO-X

Table 3: Processor configuration.

Number of cores	32	Processor core	8-wide, 256 entry ROB, 1 thread per core
Simulation duration	100M (500M) cycles	Private L1 icache and dcache	64 sets, 4 ways, 64 byte line size, 3 cycle latency
Warmup duration	10M (50M) cycles	Shared L2 cache	2048 sets, 16 ways, 64 byte line size, 23 cycle latency, noninclusive-nonexclusive
Main memory	4-wide bus, 9 columns, 90 cycle activation, 90 cycle precharge, 16 banks, 8 channels, 2048 byte rowbuffer size, FRFCFS scheduling policy		

the results presented in Section 6.4 are carried out for 500M cycles where warmup duration and classifier period length are 50M cycles each.

Our design is evaluated based on two metrics: throughput and fairness. Throughput represents cumulative instructions per cycle value of all cores, whereas fairness represents the harmonic mean of all applications' rate of throughput compared to their standalone performance [31].

Lookahead is used as the allocation policy when evaluating our work against way partitioning and Vantage partitioning, as well as evaluating SC [1]. As our configuration has 16 cache ways, a 16-way utility monitor (UMON) is utilized. However, for Vantage partitioning and SC evaluation, 16-way UMON data is interpolated to 256 data points for Vantage partitioning as proposed by Sanchez and Kozyrakis [5].

6. Results and discussion

Our proposed set partitioning has three major improvements over naive set-based partitioning: SC, DA, and FSR mechanisms. The former two aim to improve the performance of set partitioning, whereas the latter one is required for making set partitioning feasible. In this section, we first analyze the individual contribution of each improvement by removing one while keeping the rest. Then, we present an empirical analysis of FSR's ability to distribute accesses in addition to the simulation results. Finally, to show overall performance gains and to compare our work against the other work proposed in the literature, we compare set partitioning against Vantage partitioning scheme.

Table 4: Parameters of the partitioning algorithm.

Classifier period length	5M (50M) cycles	Traffic rate lower threshold	0.001	Traffic rate upper threshold	0.006
Miss rate lower threshold	0.5	Miss rate upper threshold	0.75	Decay rate lower threshold	0.5
Decay rate upper threshold	0.75	Decay reset value	4	Decay period length	5000 cycles

6.1. Set classifier

Throughput and fairness gains of set classifier SC compared to interpolated lookahead are shown in Figure 1a. SC performs better in terms of throughput compared to interpolated lookahead on 18 out of 25 workloads, achieving a throughput gain of 8.1% on the average and peaking at 33.3%. In terms of fairness, average and peak gains reach 5.6% and 53.9%, respectively.

The SC works better than interpolated lookahead since it is a tailor-made design targeting set partitioning. UMON structures in lookahead algorithm exploits the fact that LRU policy obeys stack property [32]. That allows lookahead to find out how many cache hits an application will gain when additional cache ways are allocated. However, the same rule does not apply to cache sets, and lookahead cannot determine cache hit gain per allocated set as accurately as it does for cache ways. Besides, in addition to cache access and miss information, the SC also leverages decay information, which provides hints on what portion of a cache area is being actively used. In other words, decay information gives the classifier hints on when it is safe to take cache sets away from cores. The number of cache accesses shows how intensively an application uses the cache but does not provide information on how diverse these accesses are. Decay information, on the other hand, is affected by the distribution of accesses among sets. For example, two applications may heavily access the cache, but accesses of one application may be concentrated on a handful of sets whereas accesses of the other may be distributed more evenly. In such a case, while taking sets away from the first application may be considered safe (since it is not using all of its sets anyway), taking sets away from the second application may severely degrade overall system performance. Decay information allows our classifier to distinguish application behavior in such cases.

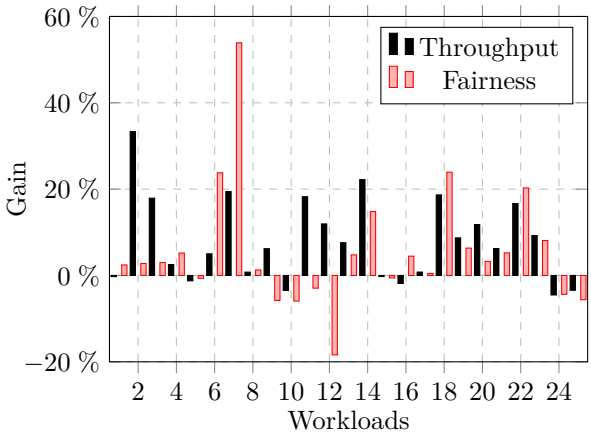
6.2. Double access mechanism

Dynamic partitioning of caches requires partition sizes (or their equivalent terms, such as the insertion position in PIPP [4]) to be periodically changed. With the set partitioning, this means changing the number of sets dedicated to a partition is quite common. Therefore, whenever partition sizes are changed, the mapping function also changes, and all data suddenly become inaccessible to all partitions. The only exception is when a thread's both beginning set and partition size remain unchanged.

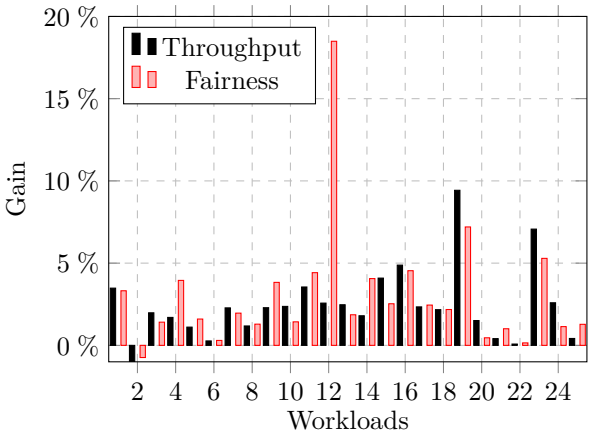
DA mechanism allows partitions to retain access to their old data until they are evicted by the new owner of their respective positions. DA initiates a secondary access if a line cannot be found upon the initial access. Naturally, if a line is found during secondary access, its latency becomes twice the regular latency of the cache.

DA reclaims some of the throughput loss caused by the artificial flushes which take place at the end of each period. Figure 1b shows the throughput and fairness gains of Set Partitioning when DA is utilized. Here 24 out of 25 workloads perform better when DA is utilized, and set partitioning performs 2.4% better with DA, on the average, peaking around 9.4%. DA also provides an extra 3% fairness on average and a peak fairness gain of 18.5%.

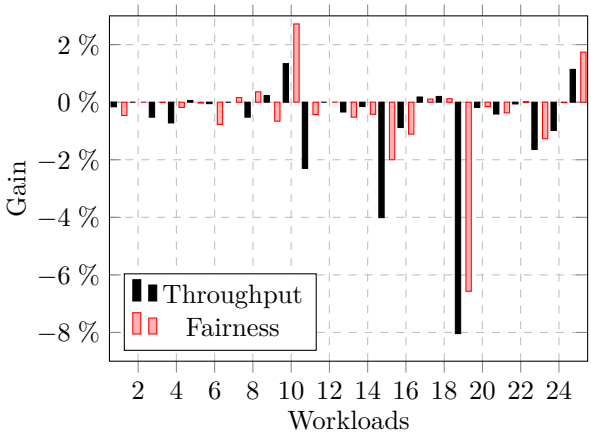
The effect of extra latency induced by the secondary access can be examined by comparing the DA mechanism with an idealistic mechanism called DA-parallel, which simultaneously initiates both primary and secondary accesses without any penalties. As expected, DA-parallel performs better than DA, but only 0.4% better in terms of throughput and 0.1% in terms of fairness, on the average. However, since DA-parallel initiates two accesses at the same time, it would require the cache to have twice as many ports to carry out these simultaneous accesses. Hence, DA-parallel becomes an infeasible option when its greater complexity and very limited gains are considered.



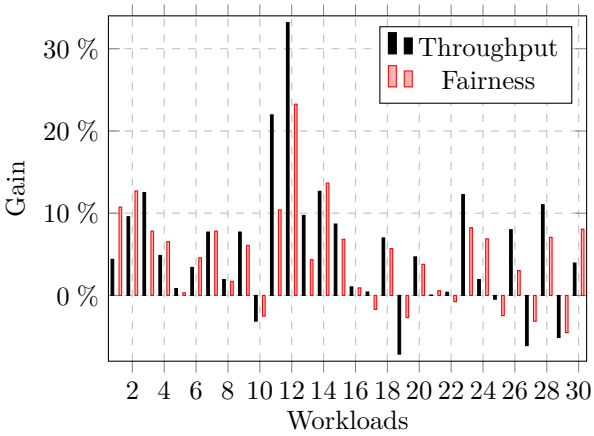
(a) Throughput and fairness gain of set partitioning with SC compared to interpolated lookahead.



(b) Throughput and fairness gain of set partitioning with DA over without DA.



(c) Throughput and fairness gain of set partitioning with FSR over without FSR.



(d) Throughput and fairness gains of set partitioning over Vantage partitioning scheme.

Figure 1: Performance gain of set partitioning with various submechanisms.

To examine the effect of cache lines becoming inaccessible due to repartitioning, we also investigate another idealistic mechanism which copies all data in the cache to a buffer and reinserts all lines into the cache using new access functions with zero time penalty. This mechanism provides a 2.8% throughput gain. The throughput gain obtained by DA, which is 2.4%, shows that DA is able to reclaim most of the throughput loss caused by changing the access function at each repartitioning period. Interestingly, this reinsertion mechanism provides a 2.3% gain in fairness compared to the 3% gain provided by DA. This is caused by the nature of both mechanisms: the reinsertion mechanism may evict some data without immediate need if the space allocated to a partition shrinks after repartitioning, while DA helps partitions to virtually have greater cache space by allowing them to access data from other partitions until these lines are gradually evicted.

6.3. Fast set redirection

The proposed FSR mechanism computes set index by rounding the partition size up to the closest power of two and applying a bit mask. Resulting values, which are greater than or equal to the partition size, are directed

back inside to the partition by subtracting the partition size from them. This causes first power–size sets to be accessed twice as much compared to the rest due to accesses to the interval [size,power] being directed to [0,power–size], under the assumption of a uniform distribution of addresses.

The set partitioning preserves associativity by partitioning the cache by sets. The number of additional conflicts caused by both decreasing number of sets and FSR is usually not high enough to cause a significant throughput loss. Figure 1c shows the throughput and fairness gains of Set Partitioning with FSR. Note that both SC and DA are utilized in these tests. According to these results, FSR causes a throughput loss of 0.7% and fairness loss of 0.4%, on the average. Out of 25 workloads, 16 perform worse with FSR. As expected, throughput loss caused by FSR is small and certainly worth making such a sacrifice for the throughput gain obtained by utilizing a set-based partitioner. It is worth noting that six workloads perform better when FSR is utilized. FSR can lead to throughput gains by either a) redirecting accesses to different sets where evicted lines will be less critical than where modulo operation would direct to, or b) causing allocation decisions to change because some applications perform worse in previous epochs. Although both cases are possible as seen in Figure 1c, these phenomena are purely coincidental and are not to be relied upon.

6.4. Set partitioning

For the workloads tested, Set Partitioning performs better than Vantage partitioning in 25 out of 30 workloads in terms of throughput, and 23 out of 30 workloads in terms of fairness, as presented in Figure 1d. Throughput gain of set partitioning is 5.6% compared to Vantage partitioning on the average, and reaches up to 33.2%, whereas fairness gain is 4.8% on the average with the peak gain of 23.3%.

Vantage achieves target partitioning by demoting one cache line per miss on the average and adjusting demotion probabilities for applications according to their actual and target sizes. However, this does not guarantee that the number of eviction candidates will be equal to the associativity of cache, as opposed to set partitioning. In some cases where no demoted cache lines are found, data from an application, whose actual size is smaller than its target size, may be evicted. Set partitioning, on the other hand, guarantees explicit partitioning where data of an application is safe from eviction by other applications.

The purpose of partitioning techniques is to enforce a cache allocation decision over all cores. It goes against this purpose when a core, which has less actual size than its target size, loses cache lines. Let unfair evictions denote such evictions where the evicted line belongs to a core which is under its target size. Unfair evictions include evictions caused by both other cores and the core itself. A high number of unfair evictions imply that lines belong to a core are continuously evicted although allocation decision tries to allocate more resources to that core. Thus, a high number of unfair evictions is a negative indicator of how well a partitioning technique fulfills the allocation decision.

A case study reveals that although there are other factors such as the ability of an allocation policy to make good decisions, cache behavior of the application, and the cache usage pattern of concurrently running applications, performance loss of Vantage compared to set partitioning is mostly related to such unfair evictions. Vantage does not take into consideration whether an evicted line belongs to a core under or over its target size, and it simply evicts one of the demoted lines, which is expected to enforce allocation decision on the average. For the workloads examined, for almost all cores where Vantage performs worse than the set partitioning, the ratio of unfair evictions to the total number of evictions is much higher than the cores where Vantage performs better. For example, in workload 1, average unfair evictions to total evictions rate has an average of 57% for the 14 cores that set partitioning improves performance and the average rate for the remaining 18 cores where

Vantage performs better is only 1%. Unfair eviction rate varies in a large interval between 40% and 75% in our case studies.

Performance results presented in Figure 1d include the cost of invalidations and evictions carried out at the end of each repartitioning period. During this process, every cache line is sequentially checked for invalidation/eviction. We assume one cycle latency for each check, and the process completes when all dirty stale lines are written back to the main memory.

To mitigate the effects of evictions, we keep the length of repartitioning period relatively high. In our configuration, in the worst case scenario (where every line in the cache is written back to the main memory), the time required for invalidation and eviction process takes only 3% of the repartitioning period. The set partitioning remains viable even in such a worst-case scenario: throughput and fairness gain of it over Vantage becomes 3.1% and 4.5% (compared to 5.6% and 4.8%), on the average, respectively.

In our studied workloads, the average number of cache lines which are forcefully evicted at the end of repartitioning periods amount to 2% of the cache, whereas the maximum value reaches 35%. The cost of invalidations are negligible, average throughput and fairness of the set partitioning would be only 0.05% higher if invalidations and evictions caused no latency, and the workload which suffered the highest amount of dirty stale lines lost only 0.2% throughput and fairness.

The set partitioning requires incoming memory addresses to be remapped to the owner's partition. In order to carry out the remapping, we introduce FSR. FSR requires an additional 12-bit addition and a layer of parallel 2-to-1 multiplexers. We evaluate our work with the assumption that the additional computation required for FSR does not increase cycle time or access latency. However, we also evaluate our work for the case where FSR increases access latency from 23 cycles to 25 cycles. The additional two cycles introduced to access latency causes set partitioning mechanism to perform 0.27% and 0.22% worse on the average, in terms of throughput and fairness respectively.

7. Conclusions and future work

We present a set-based cache partitioning scheme for guaranteeing associativity. We discuss the major problems of set-based partitioning including cache coherency, the latency of the modulo operator for nonpowers of two and pseudo cache flushes. We propose solutions to these problems as well as a classification-based allocation policy tailored for set partitioning. Finally, we analyze the effects of these solutions on throughput and fairness.

We evaluate our work by comparing it against a well-known line-based partitioning technique, Vantage. On the average, set partitioning outperforms Vantage partitioning scheme by more than 5% in terms of both throughput and fairness. It also improves performance in both metrics for almost all workloads that are tested.

Possible future work includes perfecting set partitioning by improving orthogonal techniques proposed in this study. We believe that there is still room for improvement in our set classifier by introducing new statistics, refining the thread class equation and defining new classes and/or thresholds, such as special case miss threshold proposed by Ovant et al. [2].

Another interesting research direction is to examine the usability of a highly scalable cache partitioning mechanism, such as our proposed set partitioning, on processors with a massive number of cores. One such configuration in use today is general-purpose graphics processing units (GPGPUs), where the processors typically have hundreds of processing cores. However, these processing cores are single instruction multiple data cores, and the tasks these processors are tailored for can be significantly different from the applications that are run in CPUs. Although set partitioning offers great promise in partitioning caches in such configurations due to

its high scalability, it requires further investigation to identify whether GPGPU can significantly benefit from cache partitioning or not.

Finally, this manuscript focuses on improving the throughput and fairness of the system. However, power is also a very important aspect of architectural design. We plan to investigate possible power savings related to set partitioning in a follow-up study.

Acknowledgment

This study was funded by the Scientific and Technological Research Council of Turkey (TÜBİTAK) under grant no. 114E119. The authors would like to thank M. E. Savaş for his help in running the experiments.

References

[1] Qureshi MK, Patt YN. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In: MICRO 39 Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture; Orlando, FL, USA; 2006. pp. 423-432.

[2] Ovant BS, Güney İAG, Savaş ME, Küçük GK. Allocation of last level cache partitions through thread classification with parallel universes. In: 2016 International Conference on High Performance Computing Simulation; Inssbruck, Austria; 2016. pp. 204-212.

[3] Güney İA, Yıldız A, Bayındır İU, Serdaroğlu KÇ, Bayık U et al. A machine learning approach for a scalable, energy-efficient utility-based cache partitioning. In: High Performance Computing: 30th International Conference, ISC High Performance; Frankfurt, Germany; 2015. pp. 409-421.

[4] Xie Y, Loh GH. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. ACM SIGARCH Computer Architecture News 2009; 37 (3): 174-183. doi: 10.1145/1555815.1555778

[5] Sanchez D, Kozyrakis C. Scalable and efficient fine-grained cache partitioning with Vantage. IEEE Micro 2012; 32 (3): 26-37. doi: 10.1109/MM.2012.19

[6] Wang R, Chen L. Futility scaling: High-associativity cache partitioning. In: MICRO-47 Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture; Cambridge, UK; 2014. pp. 256-367.

[7] Beckmann N, Sanchez D. Talus: A simple way to remove cliffs in cache performance. In: IEEE International Symposium on High Performance Computer Architecture (HPCA); Burlingame, CA, USA; 2015. pp. 64-75.

[8] Suh GE, Devadas S, Rudolph L. Analytical cache models with applications to cache partitioning. In: ICS '01 Proceedings of the 15th International Conference on Supercomputing; Sorrento, Italy; 2001. pp. 1-12.

[9] Stone HS, Turek J, Wolf JL. Optimal partitioning of cache memory. IEEE Transactions on Computers 1992; 41 (9): 1054-1068. doi: 10.1109/12.165388

[10] Brock J, Ye C, Ding C, Li Y, Wang X et al. Optimal cache partition-sharing. In: ICPP '15 Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP); Beijing, China; 2015. pp. 749-758.

[11] Duong N, Zhao D, Kim T, Cammarota R, Valero M et al. Improving cache management policies using dynamic reuse distances. In: Annual IEEE/ACM International Symposium on Microarchitecture; Vancouver, BC, Canada; 2012. pp. 389-400.

[12] Li L, Lu J, Cheng X. Block value based insertion policy for high performance last-level caches. In: ICS '14 Proceedings of the 28th ACM International Conference on Supercomputing; Munich, Germany; 2014. pp. 63-72.

[13] Abousamra S, El-Mahdy A, Selim S. Fair and adaptive online set-based cache partitioning. In: The 2011 International Conference on Computer Engineering & Systems; Cairo, Egypt; 2011. pp. 9-16.

[14] Chang J, Sohi GS. Cooperative cache partitioning for chip multiprocessors. In: ACM International Conference on Supercomputing 25th Anniversary Volume; Munich, Germany; 2014. pp. 402-412.

[15] Suh GE, Rudolph L, Devadas S. Dynamic partitioning of shared cache memory. The Journal of Supercomputing 2004; 28 (1): 7-26. doi: 10.1023/B:SUPE.0000014800.27383.8f

[16] Beckmann N, Sanchez D. Jigsaw: Scalable software-defined caches. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques; Edinburgh, UK; 2013. pp. 213-224.

[17] Kim S, Chandrra D, Solihin Y. Fair cache sharing and partitioning in a chip multiprocessor architecture. In: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques; Antibes, Juan-les-Pins, France; 2004. pp. 111-122.

[18] El-Sayed B, Mukkara A, Tsai O, Kasture H, Ma X et al. KPart: A hybrid cache partitioning-sharing technique for commodity multicores. In: 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA); Vienna, Austria; 2018. pp. 104-117.

[19] Wang X, Chen S, Setter J, Martinez JF. SWAP: Effective fine-grain management of shared last-level caches with minimum hardware support. In: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA); Austin, TX, USA; 2017. pp. 121-132.

[20] Ye Y, West R, Cheng Z, Li Y. COLORIS: A dynamic cache partitioning system using page coloring., In: International Conference on Parallel Architecture and Compilation Techniques (PACT); Edmonton, AB; 2014. pp. 381-392.

[21] Sherwood T, Calder B, Emer JS. Reducing cache misses using hardware and software page placement. In: ICS '99 Proceedings of the 13th International Conference on Supercomputing; Rhodes, Greece; 1999. pp. 155-164.

[22] Tam DK, Azimi R, Soares L, Stumm M. Managing shared L2 caches on multicore systems in software. In: Workshop on the Interaction Between Operating Systems and Computer Architecture, Citeseer, 2007. pp. 26-33.

[23] Zhang L, Liu Y, Wanr G, Qian D. Lightweight dynamic partitioning for last-level cache of multicore processor on real system. The Journal of Supercomputing 2014; 69 (2): 547-560. doi: 10.1007/s11227-014-1092-2

[24] Zarandi HR, Sarbazi-Azad H. Hierarchical binary set partitioning in cache memories. The Journal of Supercomputing 2005; 31 (2): 185-202. doi: 10.1007/s11227-005-0106-5

[25] Sanchez D, Kozyrakis C. The ZCache: decoupling ways and associativity. In: MICRO '43 Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture; Atlanta, GA, USA; 2010. pp. 187-198.

[26] Ranganathan P, Adve S, Jouppi NP. Reconfigurable caches and their application to media processing. In: ISCA '00 Proceedings of the 27th Annual International Symposium on Computer Architecture; Vancouver, BC, Canada; 2000. pp. 214-224.

[27] Varadarajan K, Nandy SK, Sharda V, Bharadwaj A, Iyer R et al. Molecular caches: A caching structure for dynamic creation of application-specific Heterogeneous cache regions. In: MICRO 39 Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture; Orlando, FL, USA; 2006. pp. 433-442.

[28] Vassiliadis S, Phillips J, Blaner B. Interlock collapsing ALU's. IEEE Transactions on Computers 1993; 42 (7): 825-839. doi: 10.1109/12.237723

[29] Kaxiras S, Hu Z, Martonosi M. Cache decay: Exploiting generational behavior to reduce cache leakage power. In: Proceedings 28th Annual International Symposium on Computer Architecture; Goteborg, Sweden; 2001. pp. 240-251.

[30] Henning JL. SPEC CPU2006 benchmark descriptions.ACM SIGARCH Computer Architecture News 2006; 34 (4): 1-17. doi: 10.1145/1186736.1186737

[31] Vandierendonck H, Seznec A. Fairness metrics for multi-threaded processors. IEEE Computer Architecture Letters 2011; 10 (1): 4-7. doi: 10.1109/L-CA.2011.1

[32] Mattson RL, Gecsei J, Slutz DR, Traiger IL. Evaluation techniques for storage hierarchies. IBM Systems Journal 1970; 9 (2): 78-117. doi: 10.1147/sj.92.0078