Turkish Journal of Electrical Engineering & Computer Sciences

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

Research Article

Turk J Elec Eng & Comp Sci
(2018) 26: 220 – 233
© TÜBİTAK
doi:10.3906/elk-1610-336

Last level cache partitioning via multiverse thread classification

Burak Sezin OVANT*, **İsa Ahmet GÜNEY**, **Muhammed Emin SAVAŞ**, **Gürhan KÜÇÜK**
Department of Computer Engineering, Faculty of Engineering, Yeditepe University, İstanbul, Turkey

Received: 26.10.2016 • **Accepted/Published Online:** 13.11.2017 • **Final Version:** 26.01.2018

Abstract: Last level caches (LLCs) are part of the last line of defense against the famous memory wall problem. Today, almost all processors utilize a LLC for the same reason. This study extends our previous work, which proposed a cache-partitioning mechanism using thread classification. Here, we propose three enhancements to the existing system: 1) an adaptive traffic threshold mechanism for more portable classification hardware, 2) a new method for classifying way-hungry threads, and finally, 3) a thorough comparison of two design alternatives. Compared to the original way-partitioning mechanism, we show that the proposed mechanism's performance improved by around 6%, on average.

Key words: Cache partitioning, classification, multiverse

1. Introduction

In recent years, shared last-level cache (LLC) structures have gained great importance, since they hold an important position in defending the processor performance against the well-known memory wall problem. Compared to all other instructions in an instruction set architecture of a reduced instruction set computer, the memory instructions are completed with an unpredictable latency. For instance, when the address of a LOAD instruction overlaps with the address of an earlier STORE instruction in the load/store queue, accessing the first-level cache structure become unnecessary. In such a case, the data are forwarded from the data field of an earlier STORE instruction to the data field of a LOAD instruction within a single clock cycle. However, if such data forwarding is not possible, first-level cache access, which may take around 2 to 3 cycles hit latency in today's processors, is triggered. If there is a cache miss in the first-level cache, then the second-level cache (i.e. the LLC) is accessed. The main difference between the first cache level and the LLC is that the LLC is a shared resource, whereas a dedicated thread privately handles the first level. In such a configuration, many threads might compete for a small set of cache lines of the LLC. Unfortunately, this conflict scenario is quite common, since there is no conflict resolution scheme on caches other than a simple cache associativity mechanism.

Utility-based cache partitioning (UCP) is a well-known technique that generates thread utility curves with the help of a hardware utility monitor (UMON) for better cache utilization [1]. There are two major components of a cache-partitioning mechanism: an allocation decision policy to make smart moves for resource partitioning, and an enforcement policy to guarantee those allocation decisions. The allocation mechanism in UCP uses a few heuristics, which are built on top of the UMON hardware. Its enforcement mechanism, on the other hand, is a simple LRU-based replacement policy, which tries to enforce allocation decisions in an on-demand fashion. However, there are many other enforcement policies, which share the generic UCP allocation policy, in the literature [2,3].

*Correspondence: burakovant@gmail.com

In our earlier study, we replaced the UCP allocation decision policy with a simple policy based on thread classifiers, while keeping the original enforcement policy as it was. Our thread classification mechanism focuses on collecting cache traffic, miss, and steal rates for each running thread. What makes our proposed mechanism different from any other method is that it enables cache access on multiple universes. In its current version, each thread has its own present time and its parallel universe. The present time is represented by the original LLC structure. There is an allocation decision and each thread receives its allowed share of LLC. The parallel universe time, on the other hand, is represented by a new structure called parallel universe tag directory (PUTD), which is replicated and assigned to each running thread. Each PUTD allows its dedicated thread to receive more cache ways than it is actually allowed, reducing cache ways from the remaining threads. By collecting cache statistics from each thread's present time and parallel universe, the classifier mechanism receives an upper hand in accurately classifying each of the running threads. Our test results show that the classifier provides better scalability and has better performance and fairness results than the original UCP in various processor configurations [4,5].

The major contributions of this work are listed as follows:

1. We introduce an adaptive traffic rate threshold setting mechanism for a more portable thread classification process,
2. We extend our thread classifier to carefully classify some of the threads that require special attention,
3. We add a new section to address and discuss different design strategies of our partitioning mechanism, and finally,
4. We test an alternative replacement policy over the replacement policy of UCP, which we utilized in our previously published paper.

2. Related work

A cache insertion policy focuses on keeping valuable data in the cache while evicting cache lines that do not have any positive effect on performance. We would like to cite three papers in this category. Qureshi and his team elaborate that some cache lines are not referenced until they are evicted in the LRU policy due to absence of temporal locality or reuse distances greater than cache associativity [6]. This outcome points to a very important problem: LLC resource waste. To overcome this problem, the study proposes an insertion policy called bipolar insertion policy (BIP), which inserts new cache lines into the most recently used (MRU) position on a cache set with a low probability. Otherwise, the new line is inserted into the LRU position. Then the paper shows that both BIP and LRU policies may perform better than each other in various scenarios. To adaptively select the better-performing policy, a dynamic insertion policy (DIP) is suggested. In DIP, a small portion of cache sets are dedicated to LRU and another portion are dedicated to BIP. With the use of saturated counters, these dueling sets determine which of these policies causes fewer cache misses. The remaining sets are governed by the policy selected by the set dueling mechanism.

Jaleel et al. suggest that thread behavior should be taken into account when determining whether BIP or LRU should be utilized [7], and propose the thread-aware dynamic insertion policy (TADIP), where each thread can use LRU or BIP, independently. TADIP essentially categorizes applications into *Harmful* and *Harmless*, in the context of the other applications they are running with. TADIP also uses set dueling. In half of the dueling sets dedicated to a core, the LRU policy is used, while the BIP policy is used for the other half. The rest of the cores use their current policy in these sets. With saturated counters, these dueling sets determine if a core is

Harmful to the workload in terms of cache misses. For the rest of the cache, each core uses its current policy determined by the dueling sets.

Duong et al. propose a replacement policy in which cache lines are prevented from being evicted for a number of accesses to their respective sets [8]. The authors define protective distance (PD) based on reuse distance, which determines the number of accesses cache lines that are protected. The remaining PD (RPD) of a line is reset to PD when it is accessed. If there are no unprotected lines in an inclusive cache, the line with highest RPD is evicted; in noninclusive caches, the line is bypassed. The study derives a function that approximates hit rates for a given protective distance in noninclusive caches. The mechanism searches through all possible values of PD to find the PD with the highest expected hit rate, E. For multicore systems, the study suggests an implicit partitioning by assigning different PD values to cores. The insight behind this system is that threads with higher PDs will tend to keep their lines longer in the cache, thus using a bigger portion of the cache. The mechanism selects the thread with the highest E and the corresponding PD. Then the remaining threads are examined in a descending order of E, where the PD values near peaks are tested and the one that works best is selected.

Xie and Loh utilize the UMON mechanism as their allocation policy but enforce the decision implicitly by arranging the insertion positions [2]. In this policy (PIPP), a core with a target of n cache ways inserts its new lines into a way position with nth lowest priority. When a cache line is accessed, it is promoted one step closer to the MRU position, with some probability. Additionally, the algorithm marks a core as running a stream-like application if that core experiences a number of cache misses greater than a certain threshold. Target cache way allocations for stream-like applications are set to the number of stream-like applications that are currently running. The study also proposes in-cache estimation monitors as an alternative to UMON, which dedicates a small portion of cache sets to track the utility of each core. In these dedicated sets, the core being tracked uses the LRU policy, where the remaining cores use PIPP with an upper limit.

Sanchez and Kozyrakis propose a partition enforcement mechanism called Vantage [3], which divides the cache into managed and unmanaged regions. On a cache miss, Vantage gives priority to the unmanaged region for cache line evictions. Meanwhile, cache lines from the managed region are demoted to the unmanaged region according to a coarse-grain time-stamp LRU policy. Allocation decisions are made by the UMON mechanism. Vantage enforces target allocations to be reached by demoting one cache line per cache miss on the average, instead of evicting exactly one cache line from a partition for every cache miss. This allows Vantage to enforce finer-grain allocations compared to other methods without degrading associativity.

Qureshi et al. show that there are cliffs in the relation graphs between the number of cache misses and the cache space in [1], and some applications do not immediately benefit from extra cache space until a working set fits into the cache. Recently, Beckmann and Sanchez proposed Talus, which removes these performance cliffs [9]. Talus behaves as if the cache space allocated to a thread is distributed into two partitions. The access stream is also distributed into these two partitions. The distribution rate and partition sizes are calculated by Talus according to the beginning and the end of the cliff and the target size desired.

Manikantan et al. proposed a framework that computes eviction probabilities for each core and replaces the cache lines according to these probabilities in order to achieve a finer granularity at line level [10]. Probabilistic shared cache management (PriSM) collects the augmented cache hit information using shadow tags and obtains target size. Using the target sizes, PriSM suggests a formula to compute eviction probabilities for each core. At the end of each interval, by subtracting shared cache hits of the core from the stand-alone hits of the core, PriSM obtains potential gains for each core and assigns target sizes. The authors also propose two other algorithms for improving fairness and QoS.

Li et al. devised a mechanism called value-based insertion policy (VIP) that, contrary to most previous work, takes hit benefits into consideration in addition to the miss penalties [11]. The penalty of a cache miss is determined by time spent when it is the only pending cache miss. Hit benefit is computed by assuming that the cache access is a hypothetical miss, and subtracting the miss latency by number of cycles spent where the hypothetical miss is the only pending one. The value of a cache line is equal to the sum of its miss penalty and hit benefit. VIP then utilizes two tables in order to learn value relation between incoming and evicted lines. If an incoming line has a lower predicted value than the evicted line, its eviction bit is set to 1. During a cache miss, VIP prioritizes lines whose eviction bit is set and uses the baseline replacement policy if no such candidates exist.

Wang and Chen argue that strict partition enforcement schemes, which restrict the eviction candidates to lines belonging to partitions who exceed their target sizes, hurt associativity by degrading the ability to find useless lines, especially when the number of partitions is large [12], and propose futility scaling. Futility is defined as the uselessness of a given line, which can be determined by various methods, such as LRU, LFU, or OPT. Futility scaling evicts the line with highest futility, after the futility of all candidates is multiplied with the owning partition's scaling factor. By changing these scaling factors, futility scaling can adjust the eviction rate of partitions, and therefore shrink or expand the sizes of partitions in order to meet their target sizes. The study then proposes a low-overhead futility scaling implementation based on coarse time-stamp LRU.

Guney et al. propose an alternative to the Lookahead partitioning algorithm named Lookup [4]. The Lookup algorithm utilizes a linear function, which periodically calculates a score for each core. In the offline phase of the algorithm, coefficients are computed by using machine-learning techniques for utility values and partitioning decisions made by UCP mechanism. In the online phase, scores for each core are calculated by finding the weighted sum of the first four utility values collected from the UMON and corresponding coefficients. Consequently, the cores are given a fraction of the cache space equal to the ratio of their individual score to the overall score.

Wang and Martinez allocate resources that include LLC space among multiple cores with a market-based approach, where each core tries to obtain maximum utility with a given budget [13]. Resource prices are determined by total demand on the resource, and cores iteratively update their bids according to the changes in prices until the system converges to a balance. This approach merges allocation of different types of resources among cores instead of applying independent allocation policies for different resource types, which can be harmful to performance.

Our work is orthogonal to PIPP, Vantage, futility scaling, and the UCP. These methods focus on how to enforce given target sizes among cores rather than determining the target sizes themselves. DIP and TADIP aim at improving performance by changing the insertion policy and do not utilize any target sizes. Although TADIP somehow classifies running threads, this is a bimodal classification whose insertion policy causes fewer cache misses and is not directly comparable to an allocation policy. Finally, the Lookup mechanism focuses on complexity and power reduction rather than improving processor performance. The mechanism is trained for only four auxiliary tag directory (ATD) structures. Although the authors report that the Lookup performs well in small cache configurations, its performance might quickly deteriorate as the cache associativity and the number of cores increase.

3. Thread classification-based cache partitioning

Our way-based cache partitioning mechanisms focus on the classification of running threads. Figure 1 [5] shows this classification and allocation process in detail. First, we periodically collect cache statistics from the multiverse (i.e. each thread's present time and parallel universe). Next, at the end of a period (epoch), the *Cache Allocator* is run with these statistics to change the cache allocation decisions for each thread. In the following subsections, we describe the details of each step shown in Figure 1 [5].

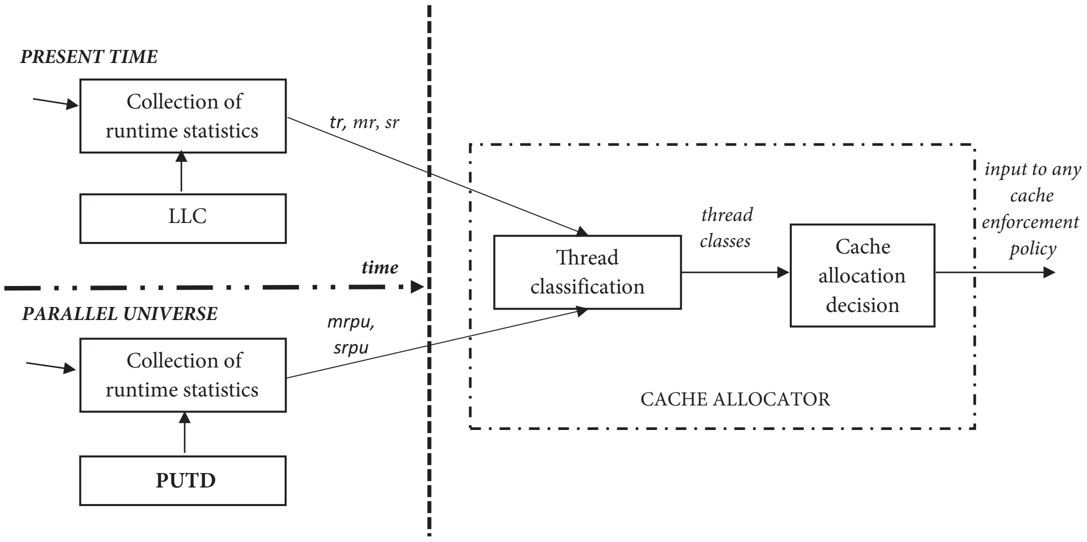
PRESENT TIME
Collection of runtime statistics
LLC
tr, mr, sr
time
PARALLEL UNIVERSE
Collection of runtime statistics
PUTD
mrpu, srpu
Thread classification
thread classes
Cache allocation decision
input to any cache enforcement policy
CACHE ALLOCATOR

Figure 1. Steps of the proposed cache allocation mechanism [5].

3.1. Collection of runtime statistics

Runtime cache statistics are collected by various hardware counters with the help of PUTDs. The physical structure of a PUTD is similar to that of an ATD described in the original study [1]. However, PUTD structures are used for a completely different purpose, since they allow access from all active threads, as the LLC does. They enable the classification hardware to recognize if a thread changes its behavior (or its class) when it receives more cache resources. For instance, on an 8-way LLC, if the current allocation decision is "four ways are given to thread A and four remaining ways are given to thread B," the PUTD of thread A may be run with "seven ways to thread A and one way to thread B," meanwhile the PUTD of thread B may be run with the "one way to thread A and seven ways to thread B" allocation decision.

We collect cache traffic rate (*tr*), miss rate (*mr*), and steal rate (*sr*) of running threads using a few hardware counters. The *tr* value is the average number of cache access per epoch duration. A thread with very low traffic rate cannot harm other threads. The *mr* value is the average number of cache misses per each cache access. It shows how a thread utilizes the cache, and when this value is very high, we can conclude that the thread does not make use of the cache. The last parameter, the *sr* value, is the average number of evictions from the cache lines that are owned by other threads per cache access. UCP runs a modified version of the LRU replacement policy. Specifically, when a thread accesses a cache set and if there is a cache miss, the policy counts the number of cache lines that belongs to that thread. If that number is less than the number of cache ways the thread is allowed to have in this epoch, the LRU cache line is evicted from the set of threads excluding that thread. We count these evictions as cache steals for the *sr* parameter. Otherwise, the LRU cache line of the pending thread is evicted. In this study, we utilize the same eviction policy. Figure 1 [5] also shows that *mr* and *sr* parameters from threads' parallel universes are also collected by the help of their PUTD structures. To reduce the hardware complexity of the PUTD structures, we use a popular method, which is known as dynamic set sampling [6].

3.2. Thread classification

We decided to classify threads in five distinct classes: *Harmful*, *Very Harmful*, *Harmless*, *Null*, and *Special*. Here we would like to clarify their differences:

1. Harmful: A thread with disruptive behavior regarding other threads. For instance, cache steals and cache conflicts due to references to nontemporal data can easily generate such a disruptive scheme. High *tr*, *mr*, and *sr* values are typical in these threads.
2. Very Harmful: When the working set of a thread does not fit into the cache or when a thread starts referencing only nontemporal data, the degree of cache disruption can be devastating to all executing threads. This class holds such catastrophic threads. High *tr* and very high *mr* and *sr* values are typical here.
3. Harmless: If a thread generates cache traffic but does not bother the other threads (a negligible amount of cache steals and cache misses), we can classify it as *Harmless*. Low *tr*, *mr*, and *sr* values are typical within this class.
4. Null class: When a thread has no (or very limited) access to LLC (zero or very low *tr* value), it falls into this class. A *Null* thread can bypass the LLC by receiving zero cache ways.
5. Special class: Finally, we track the threads that require very high cache associativity. Such threads can only perform well when they are supplied with a large number of cache ways, and we treat them in this special class. We explain this class in further detail in Section 3.5.

In a multicore environment, threads always run with other threads. Therefore, our thread classification mechanism does not try to classify threads when they are in complete isolation. Note that *tr*, *mr*, and *sr* values are collected with this concern, allowing all LLC access to target individual PUTD structures. Eq. (1) shows how our thread class (TC) classification function translates collected LLC statistics to corresponding *tr*, *mr*, and *sr* values. We extensively studied SPEC 2006 benchmarks, and we empirically determined a set of low and high thresholds for *tr*, *mr*, and *sr* functions (Eqs. (2)–(4)). Note that to classify a thread with a very high cache miss rate as *Very Harmful* we utilize an extra threshold in Eq. (3).

$$TC = T(tr) \times (1 + M(mr) \times (1 + S(sr))) \tag{1}$$

$$T(tr) = \begin{cases} 0, \; tr < TThresh_{lo} \\ 1, \; tr \geq TThresh_{lo} \wedge tr < \; TThresh_{hi} \\ 2, \; tr \geq TThresh_{hi} \end{cases} \tag{2}$$

$$M(mr) = \begin{cases} 0, \; mr < MThresh_{lo} \\ 1, \; mr \geq MThresh_{lo} \wedge mr < MThresh_{mid} \\ 2, \; mr \geq MThresh_{mid} \wedge mr < MThresh_{hi} \\ 8, \; mr \geq MThresh_{hi} \end{cases} \tag{3}$$

$$S(sr) = \begin{cases} 0, \; sr < SThresh_{lo} \\ 1, \; sr \geq SThresh_{lo} \wedge sr < \; SThresh_{hi} \\ 2, \; sr \geq SThresh_{hi} \end{cases} \tag{4}$$

In Figure 2, we show the TC curve that is the result of the various $< T, M, S >$ combinations. Here we set the interclass boundaries empirically. Note that this graph agrees with our class descriptions given in the beginning of this section. For example, we see that when the TC value is above some threshold (in this study, that threshold is fixed to 6), we assume that the corresponding thread becomes *Very Harmful* to others. However, there is still a missing point that we also would like to address. If this very same thread had received more cache resources, would it still be a *Very Harmful* thread? To investigate this important matter, we calculate the Thread Class value from a thread's parallel universe, the TC_{pu} value, where it always receives plenty of cache ways Eq. (5).

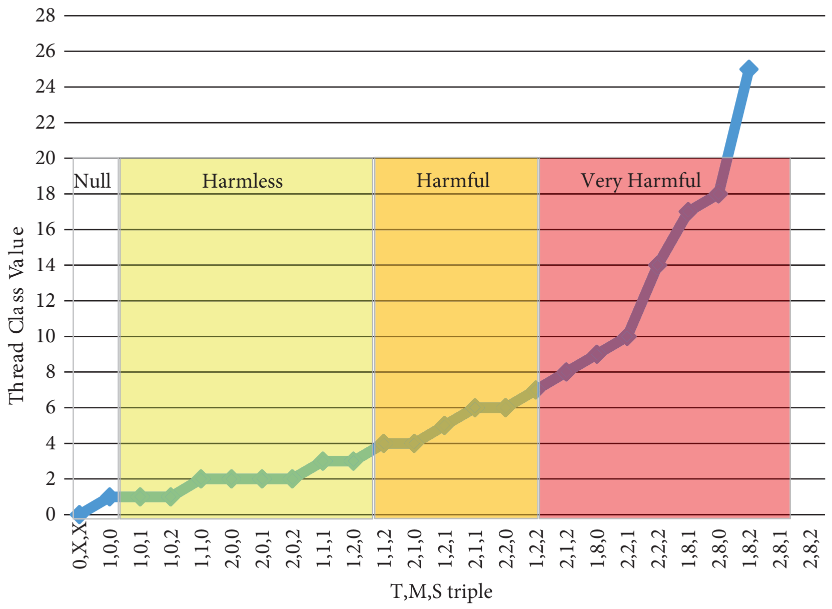

Figure 2. Classification through the TC variable [5].

$$TC_{pu} = T(tr) \times (1 + M(mrpu) \times (1 + S(srpu))) \tag{5}$$

It seems like the TC_{pu} value would be always lower than the actual TC value, since a thread always receives more cache ways when it is running in its parallel universe. However, this is not always true. For instance, a thread with a $< T = 2,\ M = 1,\ S = 0 >$ (i.e. a *Harmful* thread) might turn into a more destructive thread with more cache resources $< T = 2,\ M = 1,\ S = 2 >$(i.e. a *Very Harmful* thread). In this study, we apply an allocation decision according to the minimum TC value that was gathered from the multiverse. By doing so, we think that the class of a thread can be more accurately determined.

In this entire study, within each PUTD, we allocate three more cache ways to each target thread by reducing the number of cache ways of the remaining threads by one cache way. In higher processor configurations, the threads that donate a cache way are randomly chosen.

3.3. The adaptive traffic threshold mechanism

In our previous work, traffic thresholds are empirically determined. A series of simulations are run, and optimum values are obtained for a certain L1 cache configuration. Obviously, this is not a portable approach for every cache configuration and organization, and an adaptive traffic threshold mechanism is needed. In this study, indicators of cache traffic of threads are calculated relative to each other and to the total cache traffic. Eqs. (6)–(9) show the calculation of the new traffic rate value.

$$ir_1 = Tot_{Access} \,/\, A \tag{6}$$

$$tr_1 = N \,/\, ir_1 = N \times A \,/\, Tot_{Access} \tag{7}$$

$$ir_2 = Tot_{Access} \,/\, EA \tag{8}$$

$$tr_2 = N \,/\, ir_2 = N \times EA \,/\, Tot_{Access} \tag{9}$$

Here *tr* of an application is calculated in two steps. First, in Eq. (6), the ideal rate of accesses (*ir*) of an application is calculated by the total number of access of all applications in each specific epoch divided by the number of applications, A. This ideal rate represents the expected rate of access of an application compared to access of all applications. Then, in Eq. (7), *tr* of an application is compared to *ir* by the number of access of the application (N) divided by *ir*. Eligible applications (EA) are determined by whether tr_1 is beyond the $TThresh_{lo}$ or not. At the end of the first step, *tr* values of all applications are calculated, and the number of eligible applications is determined. At the second iteration, in Eq. (8), ideal rate is calculated again only for eligible applications. It is derived from total access of all applications divided by the number of eligible applications. Since the new ideal rate is calculated in this second step, the *tr* value can be finalized with this new ideal rate in Eq. (9). Then, as in our previous work in Section 3.2., *tr* value is compared against $TThresh_{hi}$ and $TThresh_{lo}$. Those threshold values are obtained from a set of simulations. The optimal value of $TThresh_{hi}$ is 1.0 and $TThresh_{lo}$ is 0.125. These values express the following: if the traffic rate of an application exceeds the given share of the total traffic, it is considered high traffic. If its traffic rate goes down below the $TThresh_{lo}$, it is directly considered a *Null* thread.

3.4. Allocation decision

This last stage determines the number of cache ways each thread is allowed to have during an incoming epoch. The allocation algorithm is very straightforward. All threads get the same number of ways if thread classes are all the same. Nevertheless, the algorithm must be prepared for any combination of classes, and this condition causes more complexity. Fortunately, overall decision logic can be realized with a small hardware that receives multiverse runtime statistics and produces allocation decisions.

The algorithm in Figure 3 is used to decide the amount of allocated ways at the end of every epoch. We chose to offer zero cache ways to both Very Harmful and Null threads to make the algorithm simpler, as both threads waste LLC resources, anyway. Next, the algorithm concentrates on allocating cache ways for only Harmful and Harmless threads. To simplify the allocation algorithm further, in line six of our allocation algorithm (see Figure 3), we define the *Unit*, which computes the amount of ways for allocating to every Harmful thread. In the algorithm, NL symbolizes the amount of Harmless threads, NH symbolizes the amount of Harmful threads, W symbolizes the weight of a Harmless thread over Harmful, and finally, Ways symbolizes the amount of total LLC ways. Here we decided a Harmless thread must receive twice the number of cache ways a Harmful thread receives. As a result, the value of W is set to 2.

Note that, since the amount of available physical ways could be lower than the entire number of allocated ways, the allocation algorithm does not promise strict way isolation among threads. This shows that, over a cache line, there is still a possibility of a cache conflict among Harmful and Harmless threads.

```
function ALLOCATE
  NL: Number of Harmless threads
  NH: Number of Harmful threads
  W: Weight of Harmless over Harmful

  Unit = Ways/(W x NL + NH)
  allocations[i] = 0 for each thread i

  for each thread i, do
    if i is Harmless then
      allocations[i] = Ceil(Unit * W)
    else if i is Harmful then
      allocations[i] = Ceil(Unit)
    end if
  end for
end function
```

Figure 3. Allocation algorithm.

3.5. Special threads

In our early classification mechanism, there are four main thread classes. However, our observations from comparison of Lookahead and Classifier results show that a fifth class, which we call *Special*, is needed to mark some threads that require special attention. These *Special* threads are high-utility threads that may still improve their performance with ten or more cache ways. For instance, benchmarks, such as *deal* and *astar*, need very high cache associativity, and our early classification mechanism does not respond well to such demanding benchmarks and may classify them as either *Harmless* or *Harmful*. As a result, those threads may receive only a small number of cache ways due to this somewhat weak classification. Figure 4 shows this problem. Hit counts in the figure were obtained from ATD structure of Lookahead implementation. Benchmarks in the example are *bwaves*, *deal*, *sjeng*, and *mcf*, respectively. It can be clearly seen that *deal* has 52 hits even in the tenth cache way, while others have none. To solve this problem, we utilize the PUTD to observe whether a thread can still receive cache hits from the tenth cache way. We give the highest priority to these special threads by assigning $4 \times W$ ways in our classification mechanism.

```
Core 0: 1336 181 48 5 1 0 0 0 0 0 0 0 0 0 0 0
Core 1: 292 107 100 136 93 183 174 128 188 52 10 7 2 6 5 0
Core 2: 334 77 22 3 3 1 0 1 0 0 0 0 0 0 0 0
Core 3: 7 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 4. Hit counts for *bwaves-deal-sjeng-mcf* benchmark combination.

4. Integration issues

When it comes to integration with the existing partitioning mechanisms, there are two conflicting strategies for our proposed mechanism: 1) one-design-fits-all (ODFA) strategy and 2) tailor-made design (TMD) strategy. We test both of these strategies and compare their results in Section 6.3.

4.1. ODFA strategy

This plug-and-play type design and integration strategy advocates a general design that can be accepted and used by the partitioning enforcement mechanisms without any further modifications. The main advantage of this approach is its simplicity. There is no further design complexity required to integrate the mechanism with others. For instance, the allocation mechanism can be designed to work with the LRU replacement policy and its simplistic stack nature. UMON does that, and today, it is a well-accepted allocation mechanism for a variety of cache partitioning mechanisms. However, such a strategy can be the source of various incompatibility problems between the allocation and the enforcement stages of a cache partitioning mechanism. This is especially true when the allocation and enforcement mechanism relies on a different insertion/eviction policy.

To implement such a strategy with PIPP, Vantage, or another partitioning enforcement mechanism, an extra PUTD structure is required for maintaining the present time information, which is updated by the UMON replacement policy. This requires slight modifications to the proposed scheme shown in Figure 1 [5] (i.e. the LLC box is replaced with a present time tag directory (PTTD) box).

4.2. TMD strategy

Each cache partitioning enforcement mechanism has its own assumptions that might not always be appropriate for the ODFA strategy. For instance, PIPP completely changes the promotion/insertion policy of the cache, and Vantage assumes a special cache organization (ZCache) is in place [14]. The TMD strategy is based on a custom design of the allocation policy so that the allocation and enforcement policies work in harmony. Intuitively, one can think that such a design might perform much better than the ODFA design, since it would track down the behavior of the actually running enforcement policy.

The implementation of TMD requires all PUTD structures to be maintained and updated by the policies that are utilized by the existing cache partitioner. However, this may not always be a straightforward scheme. For instance, in a TMD strategy, integration of PUTD structures to Vantage requires all PUTD blocks to be organized and maintained as dynamically set sampled ZCache structures. Unfortunately, there is no clear-cut solution for this type of integration scenario.

5. Materials and methods

Our proposed algorithm is evaluated on Macsim (http://code.google.com/p/macsim/) by trace-driven simulations. Specific to the baseline configuration on Macsim, we designed a UCP algorithm that implements the Lookahead allocation. After that, we switched the Lookahead algorithm with our proposed algorithm. In Table 1, the simulation parameters are given. To reach the essential execution path of traces, simulations were fast-forwarded until SimPoints, and executed for one billion clock cycles [15].

Table 1. Processor specifications.

Processor	4, 8, or 16 cores, 128 entry ROB, OoO execution, 1 thread per core
L1 Cache	8, 16, 32, or 64KB I- and D-cache, 4-way, 64B line, 3 cycle hit latency
L2 Cache	2, 4, or 8 MB, shared, 20 to 40 cycles hit latency
DSS sets	32

After removing benchmarks with similar cliff locations and utility curves, 19 benchmarks in SPEC2006 were selected for the workload (WL) generation. For the second stage of our study, the set of simulated WLs in our original study was extended with *deal* and *astar* benchmarks, and randomly generated 100 4-thread WLs, whose cliff locations are at 16 LLC ways, i.e. require 16 ways. In Table 2, the list of benchmarks is provided.

Table 2. Benchmarks and the cliff locations for their utility curves.

Benchmarks	Cliff location
bwaves, libquantum, mcf, milc	1-way
sphinx3, sjeng, povray, namd, zeusmp, games, hmmer, astar	4-ways
gems, xalanc, gromacs	8-ways
leslie3d	10-ways
omnetpp, gobmk, deal	12-ways
soplex, href264	16-ways

6. Tests and results

6.1. Sensitivity to the adaptive traffic threshold mechanism

In Section 3.3, we describe the requirements of the adaptive traffic threshold mechanism to achieve better portability to different sizes of the L1 cache. Here we discuss the results of the simulations. We run the simulations with 4 cores and four different L1 sizes (32, 64, 128, and 256 sets). We compare the baseline LRU algorithm, the base classifier and, finally, the adaptive mechanism, which we propose in this work. Figure 5 shows the comparison of these results. Instruction per clock cycle is the metric for performance calculations. For 32 sets, it performs slightly better (0.07%). For 64 sets, it performs as the same as base classifier, since the optimal threshold values are obtained from this configuration of L1 cache. For 128 sets, it performs slightly worse (–0.004%) and for 256 sets, it performs slightly worse again (–0.09%). On average, an adaptive classifier results in a 0.01% decrease in performance. Results show that our adaptive performs similarly to the base classifier. This proves that the adaptive mechanism can easily be implemented in the base classifier since it does not require any predefined traffic threshold and can adapt to any size of L1 cache.

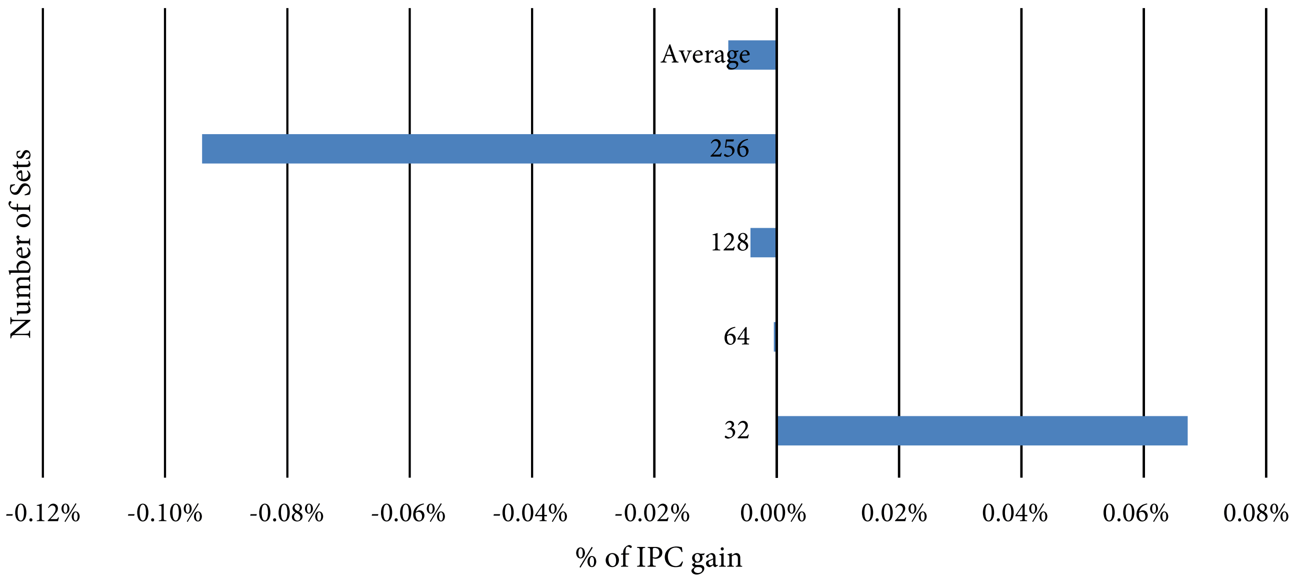

Figure 5. Adaptive classifier IPC improvement over base classifier.

6.2. Special thread

In Section 3.4., we mention the necessity of an additional class for some exceptional threads, which we call *Special*. Implementation of this additional classification results in better performance when *deal* and *astar* benchmarks are taken into consideration. In our previous work, we chose 50 random WLs for 4 core simulations but for the *Special* thread study, we add extra 50 random WLs that also include *deal* and *astar*. In Figure 6, for 100 WLs, the base classifier algorithm has a 1.24% worse performance than the Lookahead algorithm. However, our special thread implementation has a 0.75% worse performance than the Lookahead algorithm. In other words, special thread implementation improves the base classifier algorithm when running way-demanding benchmarks, such as *deal* and *astar*. Specifically for the additional 50 WLs that include *deal* and *astar*, improvement is 1.61%. Performance improved by nearly 10% in WL43, by more than 8% in WL86, and by nearly 7% in WL87. All three of those WLs include the *deal* benchmark. On the other hand, WL14 and WL15 show decreases in performance (3% and 5% drop, respectively), since those WLs do not include *deal* or *astar*. On average, Special Thread implementation performs better and makes the base classifier more robust against all kind of threads.

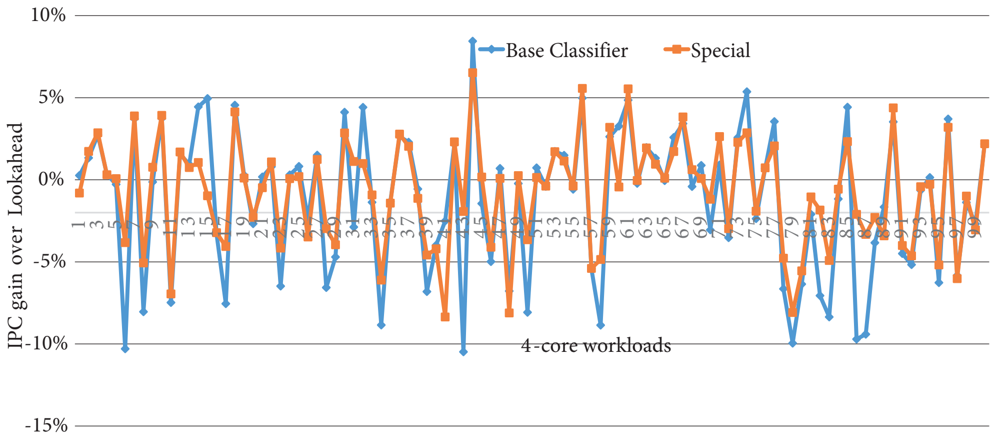

Figure 6. Special thread classifier IPC improvement with over Lookahead.

6.3. A case study: comparison of PIPP integration using ODFA and TMD

While ODFA strategy keeps LRU-based replacement policy in all PUTD structures, TMD approach focuses on implementing the replacement policy of a partitioning mechanism to provide a better harmony between the allocation and enforcement mechanisms. Therefore, in this case study, we choose PIPP as our enforcement mechanism, and make PUTD structures of TMD to run a PIPP-based replacement policy instead of the LRU-based policy that ODFA implements. In Figure 7, results show that performance is insensitive to whether the implementation is ODFA or TMD for PIPP. Another outcome of these results is that 6% average improvement in ODFA or TMD performance originated from the success of the classifier in classifying and measuring the applications, not from the fitness of ODFA or TMD for PIPP.

7. Discussion

Contemporary processors utilize a LLC that is shared by all running threads. Cache partitioning mechanisms propose resource management strategies for allocating sufficient cache resources to these threads so that the overall system performance is improved. Today, Lookahead cache allocation mechanism for UCP is one of the

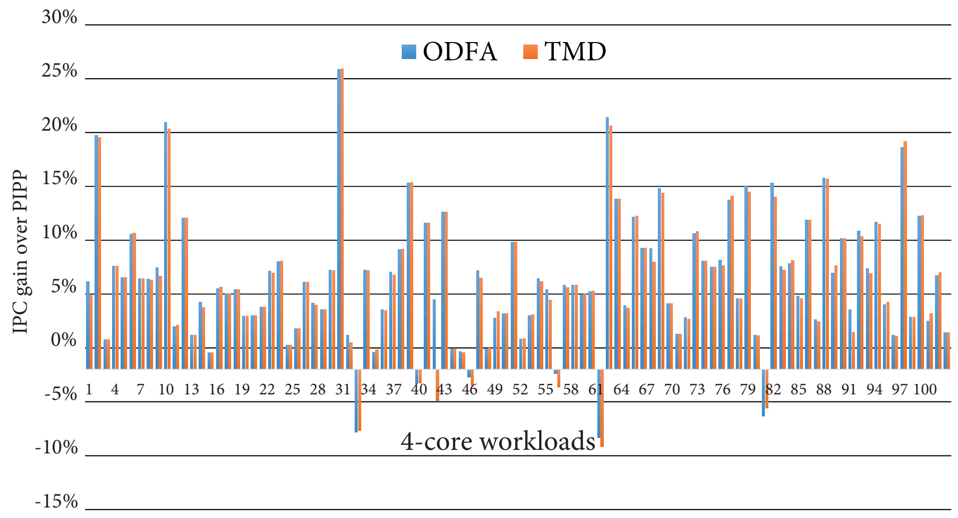

Figure 7. ODFA and TMD classifier IPC improvement over PIPP.

well-known mechanisms that is heavily utilized in many cache-partitioning mechanisms. Specifically, the UCP proposes a cache utility monitor named UMON to periodically track the utility curves for each thread and make allocation decisions, accordingly. Utility curves are constructed by collecting cache misses in ATDs dedicated to each thread.

In this study, we propose a new cache allocation policy that chooses the amount of cache partitions through thread classification and PUTDs, which are structurally identical to ATDs. However, the function of these structures is to create parallel execution dimensions, which we call a multiverse, in which we can test whether each thread can make good use of extra cache resources or not. We collect cache traffic, cache miss rate, and cache steal rate from these structures and classify each thread according to their harmful behavior on other threads. By the help of a simple allocation function, we assign cache ways to threads with different classes.

We extend our earlier work by introducing an adaptive traffic threshold mechanism, a special class to cover way-hungry applications, and, finally, two implementation strategies that define different ways of connecting various allocation and enforcement policies.

We evaluate the proposed mechanism in 4-core processor configurations with 100 WLs, which contain SPEC2006 benchmarks. There are three major findings in this study. First, the results show that our adaptive traffic threshold mechanism performs comparably to the base classifier. An adaptive approach makes our classifier more robust regarding different sizes of L1 cache, whereas, in our earlier work, we needed to modify our traffic rate thresholds when integrating with different L1 sizes.

Second, the introduction of a new special thread class makes the classifier more robust regarding WLs that require high-associativity caches. Our proposed new classifier mechanism performs 0.5% better than the base classifier.

Finally, we show that the two implementation alternatives for integration with other partitioning enforcement policies have nearly the same performance results. We conclude that our classification mechanism is insensitive to such implementation strategies, and simple ODFA design strategy is sufficient to replace any allocation policy with a performance that is around 6% better, on average.

Acknowledgments

This work is supported by the Scientific and Technical Research Council of Turkey (TÜBİTAK) under the title Wise-Cache and Grant No:114E119.

References

[1] Qureshi K, Patt YN. Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches. In: Int Symp Microarch 2006; 9–13 December 2006; Orlando, FL, USA: IEEE Computer Society. pp. 423-432.

[2] Xie Y, Loh GH. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In: ISCA 2009; 20–24 June 2009; Austin, TX, USA: ACM. pp. 174-183.

[3] Sanchez D, Kozyrakis C. Scalable and efficient fine-grained cache partitioning with Vantage. IEEE Micro 2012; 3: 26-37.

[4] Guney IA, Yildiz A, Bayindir IU, Serdaroglu KC, Bayik U, Kucuk G. A machine learning approach for a scalable, energy-efficient utility-based cache partitioning. In: ISC 2015; 12–15 July 2015; Frankfurt, Germany: Springer International Publishing. pp. 409-421.

[5] Ovant BS, Guney IA, Savas ME, Kucuk G. Allocation of last level cache partitions through thread classification with parallel universes. In: HPCS 2016; 18–22 July 2016; Innsbruck, Austria: IEEE. pp. 204-212.

[6] Qureshi MK, Jaleel A, Patt YN, Steely SC, Emer J. Adaptive insertion policies for high performance caching. In: ISCA 2007; 9–13 June 2007; San Diego, CA, USA: ACM. pp. 381-391.

[7] Jaleel A, Hasenplaugh W, Qureshi MK, Sebot J, Steely SC, Emer J. Adaptive insertion policies for managing shared caches. In: PACT 2008; 25–29 October 2008; Toronto, Canada: IEEE Computer Society. pp. 208-219.

[8] Duong N, Zhao D, Kim T, Cammarota R, Valero M, Veidenbaum AV. Improving cache management policies using dynamic reuse distances. In: Int Symp Microarch 2012; 1–5 December 2012; Vancouver, BC, Canada: IEEE Computer Society. pp. 389-400.

[9] Beckmann N, Sanchez D. Talus: a simple way to remove cliffs in cache performance. In: HPCA 2015; 7–11 February 2015; San Francisco, CA, USA: IEEE Computer Society. pp. 64-75.

[10] Manikantan R, Rajan K, Govindarajan R. Probabilistic shared cache management (PriSM). In: ISCA 2012; 9–13 June 2012; Portland, OR, USA: IEEE Computer Society. pp. 428-439.

[11] Li L, Lu J, Cheng X. Block value based insertion policy for high performance last-level caches. Int Symp Microarch 2013; 7–11 December 2013; Davis, CA, USA: IEEE Computer Society. pp. 284-296.

[12] Wang R, Chen L. Futility scaling: high-associativity cache partitioning. Int Symp Microarch 2014; 13–17 December 2014; Cambridge, UK: IEEE Computer Society. pp. 356-367.

[13] Wang X, Martinez JF. Xchange: a market-based approach to scalable dynamic multi-resource allocation in multicore architectures. HPCA 2015; 7–11 February 2015; San Francisco, CA, USA: IEEE Computer Society. pp. 113-125.

[14] Sanchez D, Kozyrakis C. The zcache: decoupling ways and associativity. In: Int Symp Microarch 2010; 4–8 December 2010; Atlanta, Georgia, USA: IEEE Computer Society. pp. 187-198.

[15] Sherwood T, Perelman E, Calder B. Basic block distribution analysis to find periodic behavior and simulation points in applications. In: PACT 2001; 3–7 September 2001; La Jolla, CA, USA: University of California at San Diego. pp. 3-14.