LAST LEVEL CACHE PARTITIONING VIA MULTIVERSE THREAD CLASSIFICATION

by
Burak Sezin Ovant

Submitted to Graduate School of Natural and Applied Sciences
in Partial Fulfillment of the Requirements
for the Degree of Master of Science in
Computer Engineering

Yeditepe University
2017

LAST LEVEL CACHE PARTITIONING VIA MULTIVERSE THREAD CLASSIFICATION

APPROVED BY:

Assoc. Prof. Dr. Gürhan Küçük (Thesis Supervisor)	[illegible]
Prof. Dr. Haluk Rahmi Topçuoğlu	[illegible]
Assoc. Prof. Dr. Sezer Gören Uğurdağ	[illegible]

DATE OF APPROVAL: ..../..../2017

ACKNOWLEDGEMENTS

It is with immense gratitude that I acknowledge the support and help of my Professor Gürhan Küçük. Pursuing my thesis under her supervision has been an experience which broadens the mind and presents an unlimited source of learning.

I thank Research Assistants İsa Ahmet Güney and Muhammet Emin Savaş.

Finally, I would like to thank my family and my beautiful wife Gizem Poslu Ovant for their endless love and support, which makes everything more beautiful.

We would like to thank TUBITAK for supporting for this project under grant number 114E119.

ABSTRACT

LAST LEVEL CACHE PARTITIONING VIA MULTIVERSE THREAD CLASSIFICATION

Last Level Caches (LLCs) are positioned in the last line of defense fighting with the famous memory wall problem. Today, almost all simultaneous multithreaded (SMT) and chip multi processors (CMP) utilize a LLC for the same reason. Cache partitioning is one of the well-studied methods that targets improved system performance through isolation of cache lines dedicated to each thread. In this study, we propose a new allocation policy that chooses the amount of cache partitions through thread classification and auxiliary cache structures, which we call Parallel Universe Tag Directories (PUTDs). Each thread maintains a dedicated PUTD structure, which collects information from another execution dimension, where the owner thread receives more cache resources. Our test results show that our proposed mechanism gives better performance and fairness results with negligible hardware requirements compared to the current state of the art, in all studied processor configurations.

ÖZET

ÇOKLU EVRENLER KULLANARAK İŞ PARÇACIKLARI ÜZERİNDEN SON SEVİYE ÖNBELLEKLERİN SINIFLANDIRILMASI

Son Seviye Önbellekler (SSÖ) ünlü bellek duvarı problemiyle savaşan son hattadırlar. Günümüzde, hemen hemen bütün eşzamanlı çoklu iş parçacıklı Simultaneous MultiThreading (SMT) ve yonga çoklu işlemciler, Chip Multi Processor (CMP) SSÖ'yü aynı sebepten dolayı kullanmaktadır. Önbellek paylaşımı, her iş parçacığına özel önbellek yollarının yalıtılması yoluyla güçlendirilmiş system performansını hedefleyen, iyi çalışılmış metodlardan birisidir. Bu çalışmada, iş parçacığı sınıflandırma ve Paralel Evren Etiket Klasörleri (PUTD) olarak adlandırdığımız yardımcı önbellek yapıları yoluyla önbellek bölümlerine karar veren yeni bir bölümleme politikası öneriyoruz. Her bir iş parçacığı, kendisinin daha fazla önbellek kaynağına sahip olduğu başka bir yürütme boyutundan bilgi toplayan adanmış bir PUTD yapısı tutar. Test sonuçlarımız, önerdiğimiz mekanizmanın çalışılan tüm işlemci yapılandırmalarında, literatürdeki modern çalışmalara kıyasla, gözardı edilebilir donanım gereksinimleri ile beraber daha iyi performans ve adalet sonuçları verdiğini gösteriyor.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

ABSTRACT iv

ÖZET v

LIST OF FIGURES viii

LIST OF TABLES xi

LIST OF SYMBOLS/ABBREVIATIONS xii

1. INTRODUCTION 1

2. RELATED WORK 7

3. CACHE ALLOCATION THROUGH THREAD CLASSIFICATION 14

3.1. COLLECTION OF RUNTIME STATISTICS 15
3.2. THREAD CLASSIFICATION 16
3.3. ADAPTIVE TRAFFIC THRESHOLD MECHANISM 20
3.4. SPECIAL THREAD 21
3.5. ALLOCATION DECISION 22

4. INTEGRATION ISSUES 27

4.1. ONE-DESIGN-FITS-ALL (ODFA) STRATEGY 27
4.2. TAILOR-MADE DESIGN (TMD) STRATEGY 28

5. DESIGN AND IMPLEMENTATION 29

6. EXPERIMENTAL METHODOLOGY 39

7. TESTS AND RESULTS 42

7.1. PERFORMANCE 42
7.2. FAIRNESS 45
7.3. SENSITIVITY TO ADAPTIVE TRAFFIC THRESHOLD MECHANISM 48
7.4. SPECIAL THREAD 49
7.5. A CASE STUDY: COMPARISON OF PIPP INTEGRATION USING ODFA AND TMD 50

8. CONCLUSION 52

REFERENCES 54

LIST OF FIGURES

Figure 1.1. Framework of UCP 2

Figure 2.1. Zero reuse lines for 1MB 16-way L2 cache 7

Figure 2.2. Example operation of PIPP for a variety of cache misses (insertions) and hits (promotions) 9

Figure 2.3. Benchmarks with non-convex utility curves i.e. cliffs 11

Figure 2.4. Performance of libquantum over cache sizes. LRU causes a performance cliff at 32MB 11

Figure 3.1. Steps of the proposed cache allocation mechanism 14

Figure 3.2. Classification through the ThreadClass variable 18

Figure 3.3. Hit counts for *bwaves-deal-sjeng-mcf* benchmark combination 22

Figure 3.4. Steps of the cache allocation decision 24

Figure 3.5. Comparison of Classifier with Lookahead in terms of required micro-operations and latency (in cycles) for completion in worst ca se scenarios, in logarithmic scale 26

Figure 5.1. Set of parameters in params.in file 30

Figure 5.2. Details of script.sh file 31

Figure 5.3. Instantiation of ATD structures i.e. Tag Arrays(TA) and Hit Counters (HC) .. 32

Figure 5.4. Set selection logic of TADIP 33

Figure 5.5. Initialization of TADIP 34

Figure 5.6. Decision of insertion position of TADIP 35

Figure 5.7. Increasing and decreasing PSEL counter of TADIP 35

Figure 5.8. Determination of isLIP boolean in TADIP 36

Figure 5.9. Implementation of Classifier allocation algorithm... 37

Figure 5.10. Example output statistics of trace method implementation............................ 38

Figure 6.1. Macsim implementation of Lookahead algorithm 39

Figure 7.1. Speedup of the Classifier and the Lookahead over LRU in 4-core/16-way LLC configuration... 43

Figure 7.2. Speedup of the Classifier and the Lookahead over LRU in 8-core/32-way LLC configuration... 44

Figure 7.3. Speedup of the Classifier and the Lookahead over LRU in 16-core/64-way LLC configuration... 44

Figure 7.4. Weighted speedup and harmonic mean gain of Classifier over Lookahead in 4-core/16-way LLC configuration 46

Figure 7.5. Weighted speedup and harmonic mean gain of Classifier over Lookahead in 8-core/32-way LLC configuration 47

Figure 7.6. Weighted speedup and harmonic mean gain of Classifier over Lookahead in 16-core/64-way LLC configuration 48

Figure 7.7. Adaptive Classifier IPC gain over base Classifier 49

Figure 7.8. IPC gain of Classifier with Special thread mechanism over Lookahead 50

Figure 7.9. IPC gain of ODFA and TMD Classifier over baseline PIPP 51

LIST OF TABLES

Table 6.1. Processor specifications.. 40

Table 6.2. Benchmarks and the cliff locations for their utility curves................................ 41

LIST OF SYMBOLS/ABBREVIATIONS

A	Number of applications
ATD	Auxiliary tag directory
B	Byte
BIP	Bipolar insertion policy
C	Number of cores
CMP	Chip multi processors
D-cache	Data cache
DIP	Dynamic insertion policy
DSS	Dynamic set sampling
E	Epoch duration for allocation decisions in kcycles
EA	Number of eligible applications
h	Number of cache hits
I-cache	Instruction cache
ICEmons	In-cache estimation monitors
IPC	Instructions per cycle
ISA	Instruction set architecture
ir	Ideal number of accesses rate
KB	Kilobyte
L1	Level 1 cache
LIP	LRU insertion policy
LLC	Last level cache
LRU	Least recently used
LSQ	Load/store queue
M	Number of cache misses
M(mr)	Miss rate function
M(mrpu)	Miss rate function parallel universe

MB	Megabyte
mr	Miss rate
mrpu	Miss rate parallel universe
MRU	Most recently used
MSB	Most significant bit
PSEL	Policy selection
PUTD	Parallel universe tag directory
$MThresh_{hi}$	Miss threshold high
$MThresh_{lo}$	Miss threshold low
$MThresh_{mid}$	Miss threshold medium
N	Number of cache accesses from a thread
NH	Number of harmful threads
NL	Number of harmless threads
ODFA	One-design-fits-all strategy
PD	Protective distance
PIPP	Promotion/insertion pseudo-partitioning
PriSM	Probabilistic shared cache management
QoS	Quality of service
RISC	Reduced instruction set computer
RPD	Remaining protective distance
S(sr)	Steal rate function
S(srpu)	Steal rate function parallel universe
SMT	Simultaneous multithreaded
sr	Steal rate
srpu	Steal rate parallel universe
$SThresh_{hi}$	Steal threshold high
$SThresh_{lo}$	Steal threshold low
T(tr)	Cache traffic function
TADIP	Thread-aware dynamic insertion policy
TMD	Tailor-made design strategy

Tot_Access	Total number of accesses of all applications in one epoch
tr	Traffic rate
$TThresh_{hi}$	Traffic threshold high
$TThresh_{lo}$	Traffic threshold low
U	Number of unit cache ways to each harmful thread
UCP	Utility-based cache partitioning
UMON	Utility monitor
VIP	Value based insertion policy
w	Number of cache ways
W	Weight of a harmless thread over a harmful thread
WL	Workload

1. INTRODUCTION

In recent years, Last Level Cache (LLC) structures, which are shared by multiple threads, gain 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 (ISA) of a Reduced Instruction Set Computer (RISC), the memory instructions complete 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 (LSQ), accessing to the first level cache structure become unnecessary. In such a case, the data is forwarded from the data field of the earlier STORE instruction to the data field of the LOAD instruction within a single clock cycle. However, if such data forwarding is not possible, first level cache access, which may take 2 to 3 cycles hit latency in today's processors, is triggered. If there is a cache miss to the first level cache, then, the second level cache (i.e. the LLC) is accessed. Moreover, in today's processors, there is a third level of cache, and that level is generally being referred as the LLC. The main difference between the other cache levels and the LLC is that the LLC is a shared resource whereas the each of the other levels are privately handled by a dedicated thread. However, in such a configuration, multiple threads might compete for a small set of cache lines of the LLC even though there are enough free cache resources for all running threads over the entire cache. Unfortunately, this type of a conflict scenario is quite common since there is no smart conflict resolution scheme on caches, other than simple cache associativity (or hash bucket) mechanism. Worst of all, any cache conflict may severely degrade the overall system performance, since LLC is the last line of defense before hitting the memory wall. To overcome this problem, cache partitioning mechanisms provide isolated cache space for each running thread.

The simplest cache partitioning mechanism is known as the static partitioning. Here, the shared cache space is divided into multiple fixed-size dedicated cache partitions. Unfortunately, this over-simplistic cache organization does not provide any performance guarantee. When the size of working sets of running threads are imbalanced, static partitioning experiences a worst-case scenario. For instance, a partition dedicated to a thread with no memory traffic becomes inaccessible to other threads that can really make use of some extra cache space.

Utility-based Cache Partitioning (UCP) is one of the first cache partitioning mechanisms that target adaptive partitioning by tracking thread behavior at runtime [1]. UCP introduces a hardware-based Utility Monitor (UMON) to collect cache utility statistics to decide on the number of cache ways to be assigned to each thread. In Figure 1.1, framework of UCP [1] is shown.

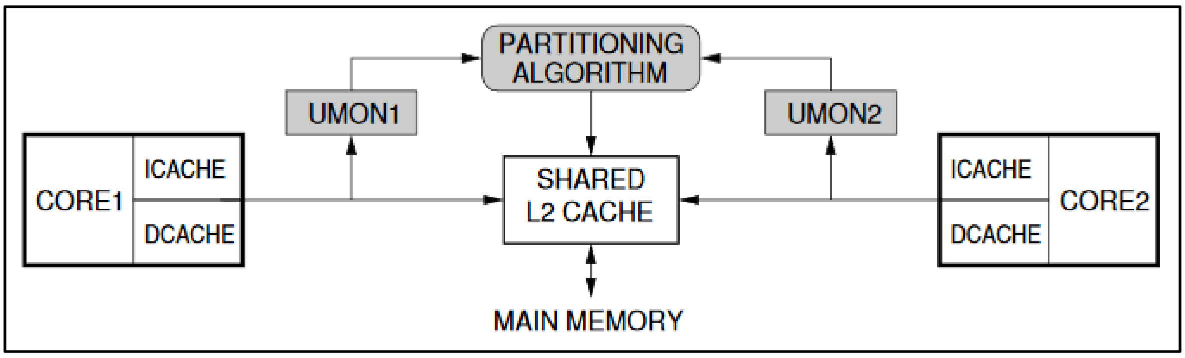


Figure 1.1. Framework of UCP

In the decision process, UCP relies on several heuristics for an accurate allocation decision. However, this only represents the front-end of a cache partitioning algorithm. In the back-end, an enforcement policy is also required for guaranteeing the allocation decisions to be somewhat met. UCP, Promotion/Insertion Pseudo-Partitioning (PIPP) [2] and Vantage [3] present a variety of enforcement policies that share the same UCP allocation policy in their front-end. Clearly, the success of a cache partitioning mechanism relies on cumulative success of both the cache allocation and the cache enforcement policies.

In this study, we focus on the front-end of Last Level Cache (LLC) partitioning mechanisms by replacing the well-known, but relatively complex, UCP Lookahead algorithm with a simple thread classifying circuitry[1]. Specifically, we periodically collect various runtime LLC statistics related to each of the running threads and identify their classes. Meanwhile, we also keep track of the behavior of each thread in their parallel universes[2] where they have more cache resources compared to their current cache allocation. Consequently, the class information obtained from the last epoch and the class information obtained from the thread's multiverse (i.e. present time and parallel universes, altogether) are combined and used to assign the right amount of cache resources to each thread. For Instance, a thread with no LLC traffic should not get any LLC resource, whereas a thread, which makes good use of the LLC in its last epoch and in multiverse, should receive a certain portion of the LLC.

The original allocation policy that is well-accepted in most of the contemporary cache partitioning algorithms is known as the Lookahead algorithm [1]. The idea is based on the cache utility curves collected by additional thread-specific way-based cache structures named the Auxiliary Tag Directories (ATD) and accompanied cache hit counters integrated into thread-specific structures named Utility Monitors (UMONs). A cache utility curve provides crucial runtime information for estimating the target cache size required for each thread since it represents the number of extra cache hits a thread can receive when new cache ways are allocated to that specific thread. Thus, the mechanism is claimed to be quite accurate as long as the cache hits (i.e. the utility curve) and the performance curve of a thread are correlated. To get a better estimate on the required number of useful cache ways for each thread, the Lookahead policy makes use of a metric known as the marginal utility. Specifically, the marginal utility is defined as h/w, where h is the number of cache hits and w is the number of cache ways. Hence, this metric gives the number of cache hits per cache way, on the average. As a result, a thread with the maximum marginal utility may receive w cache ways, and the algorithm starts from the beginning until there is no unassigned cache ways left. This is an $O(n^2)$ algorithm, which requires a complex circuitry analyzed in a prior study [4].

[1] Because of its characteristics and usage patterns, partitioning a first level cache is a more challenging subject. In this study, we focus only on the LLC as most of the prior art do.

[2] We use the term parallel universe since we collect statistics from a cache that is run in another execution dimension. In computer architecture research, collection of such statistics is not a common practice since the thread execution follows a single path and, in reality, we do not have such a chance to see the outcome of any alternate execution paths.

To further motivate our study, we would like to list the problems that we observe in the existing studies.

(i). First, in an Out-of-Order (OoO) super-scalar processor that integrates a speculative execution mechanism, a large number of cache hits does not always imply that all memory instructions related to those cache hits are to be successfully retired. Some or most of the hit-dependent instructions might be flushed away when a prior branch instruction is mispredicted. Therefore, some of the prior art focus on in-order non-speculative processors. We believe that this type of assumption is not realistic since most of the high-end processors still have a speculative OoO superscalar core.

(ii). Second, the criticality of each cache hit may greatly vary. The UMON mechanism in UCP assumes that all the cache hits of all running threads have the same positive effect on the processor performance. However, this assumption is definitely not true either. A value that is supplied by a cache hit might have many consumers, and such cache hits can really improve the processor performance. On the contrary, a cache hit with a single consumer cannot have any significant impact over the performance. Worst of all, both the first and the second factors, which are discussed up to this point, can conflict with each other. For instance, when a thread has many cache hits consumed by a high volume of instructions, it does not necessarily mean that those hits are all critical. In case of a speculative superscalar OoO execution, all of those instructions might be in the mispredicted path of a thread and has to be flushed, anyways.

(iii). Third, the UMON and similar mechanisms provide information on each thread's standalone run. The utility curves collected by the mechanism does not give any clue on the interaction of running threads when they share the same resource. For instance, in a 16-way set-associative LLC configuration, when the Lookahead mechanism finds that a thread achieves the maximum marginal utility value when it receives 10 cache ways, it immediately allocates 10 cache ways to that thread. But, if there is another thread, which has a little lower maximum marginal utility value with 10 cache ways, then, it will not be able to receive enough cache resource to achieve its peak performance. This is a very serious fairness issue, which is not considered in the UCP study.

(iv). Finally, the UMON mechanism relies on utility curves based on the LRU replacement policy and its stack-based nature. Other mechanisms, such as PIPP and Vantage, also try to utilize UMON as their cache allocation decision mechanism. However, these enforcement mechanism propose a variety of insertion and eviction policies that are not easily representable by a stack. Thus, the cache allocation decisions made by the UMON circuitry may not be suitable for the cache partitioning algorithms that do not use LRU as their replacement policy.

In this study, we propose a thread classification and cache allocation mechanism, which is based on various cache statistics rather than a simple cache hit counter-based method. Our mechanism can replace the UMON based UCP cache allocation mechanism located in the front-end of many cache partitioning mechanisms [1] [2] [3]. In the extreme case, our Classifier-based allocation mechanism may perform more than 8 per cent better than the Lookahead allocation policy. We also evaluate our mechanism in terms of its operational complexity and fairness. As a result, our evaluation shows that our Classifier-based allocation mechanism requires only a fraction (less than 1 per cent) of operations required by the Lookahead mechanism. Finally, our mechanism shows better fairness results in more than 90 per cent of the workloads that we studied.

The major contributions of this work are listed as follows:

(i). We do a comprehensive literature survey and add some of the most recent published work in this research area,

(ii). We add an example scenario explaining our allocation mechanism in much greater detail,

(iii). We extend our thread classifier to carefully classify some of the threads that require special attention,

(iv). We add a new section to address and discuss different design strategies on our partitioner mechanism, (v). We test an alternative replacement policy over the replacement policy of UCP, which we utilized in our previously published paper. [5]

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]. Figure 2.1 from the study shows that more than half the L2 cache lines installed in the cache are never reused before getting evicted. This outcome is pointing a very important problem which is wasting LLC resources. To overcome this problem, the study proposes an insertion policy called Bipolar Insertion Policy (BIP) which inserts new cache lines to 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, the Dynamic Insertion Policy (DIP) is suggested. In DIP, a small portion of cache sets are dedicated to LRU and another portion is dedicated to BIP. With the use of saturated counters, these dueling sets determine which of these policies causes less cache misses. The remaining sets are governed by the policy selected by the set dueling mechanism.

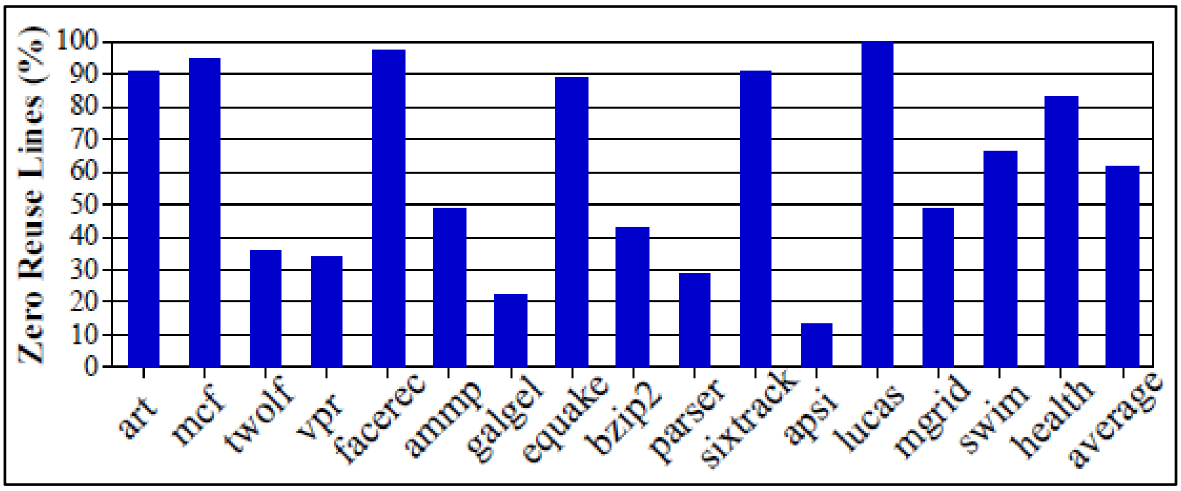


Figure 2.1. Zero reuse lines for 1MB 16-way L2 cache [6]

Jaleel et al. suggest that thread behavior should be taken into account when determining whether BIP or LRU should be utilized [7], and propose 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. Rests 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 are protected for. 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 non-inclusive caches the line is bypassed. The study derives a function which approximates hit rates for a given protective distance in non-inclusive caches. The mechanism searches through all possible values of PD to find the PD with the highest expected hit rate, E. For multi-core 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, which is 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. Figure 2.2 from the study explains an example operation of PIPP in a very clear way. The study also proposes In-Cache Estimation Monitors (ICEmons) 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.

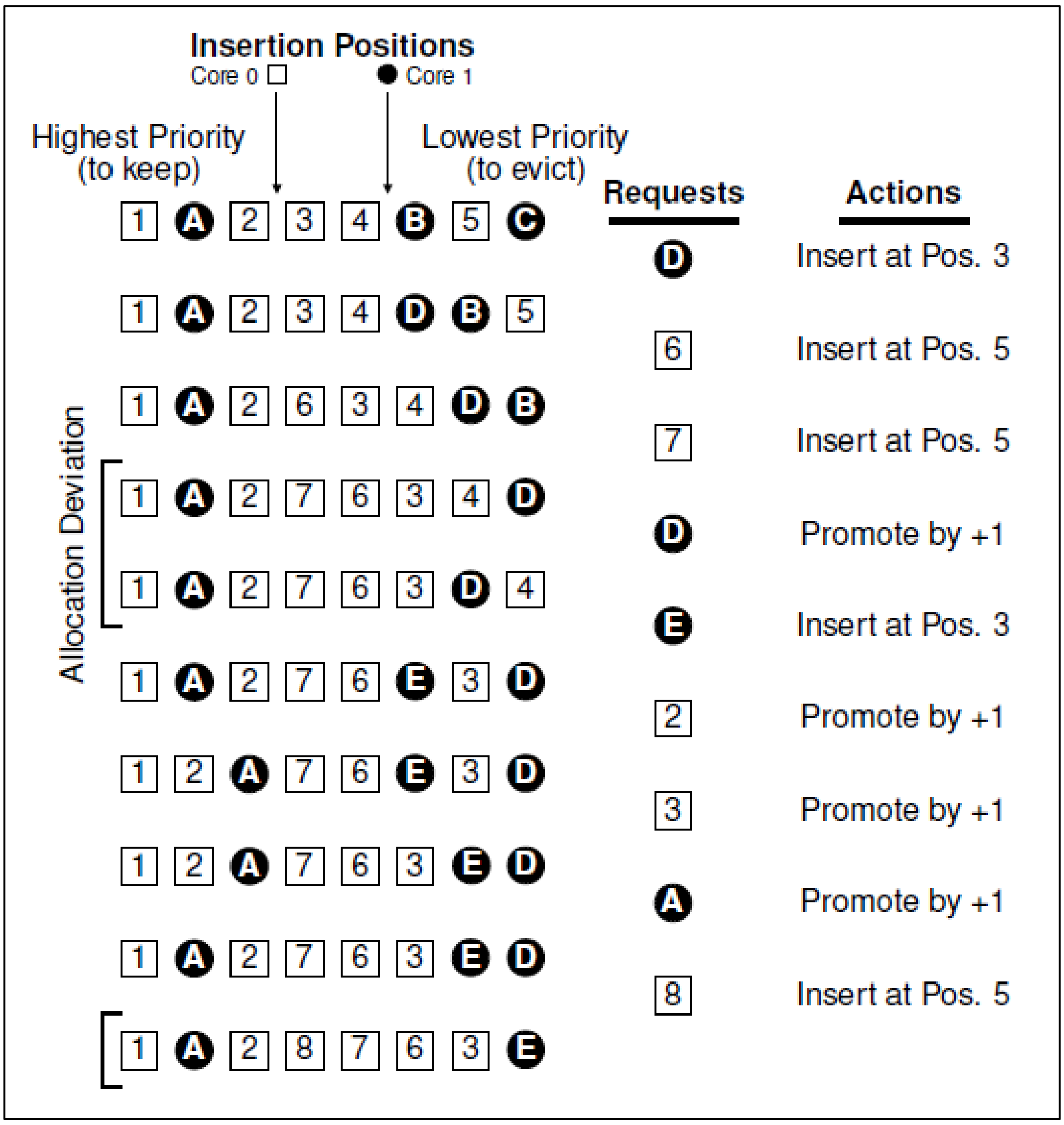


Figure 2.2. Example operation of PIPP for a variety of cache misses (insertions) and hits (promotions). Evictions always choose the lowest-priority cache line [2]

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 give 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 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 propose 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. Figure 2.3 shows the cliffs between misses and cache space from the UCP study [1]; and Figure 2.4 shows the improvement of Talus [9] on those cliffs.

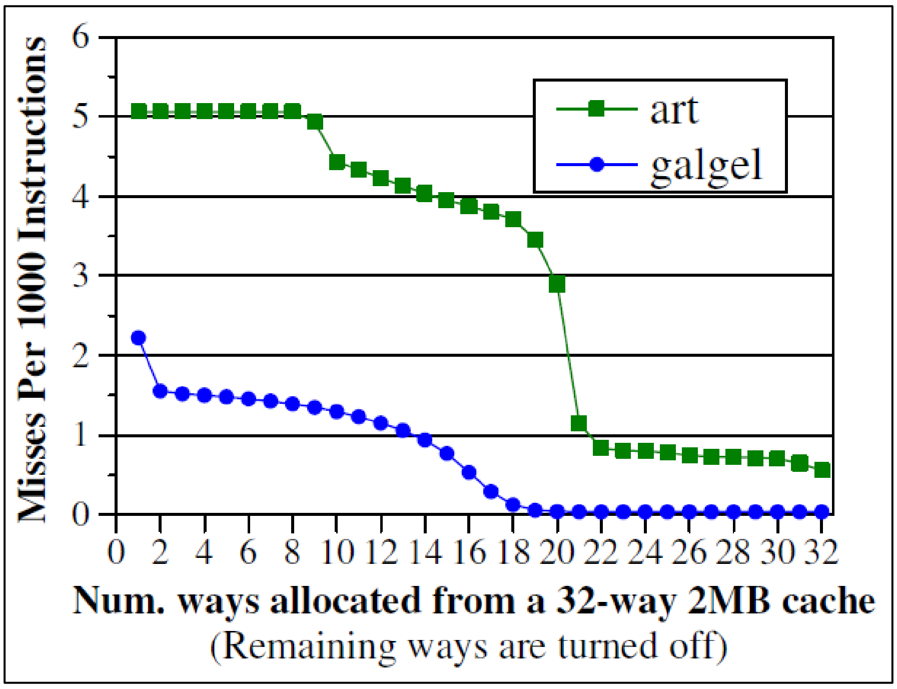


Figure 2.3. Benchmarks with non-convex utility curves i.e. cliffs [1]

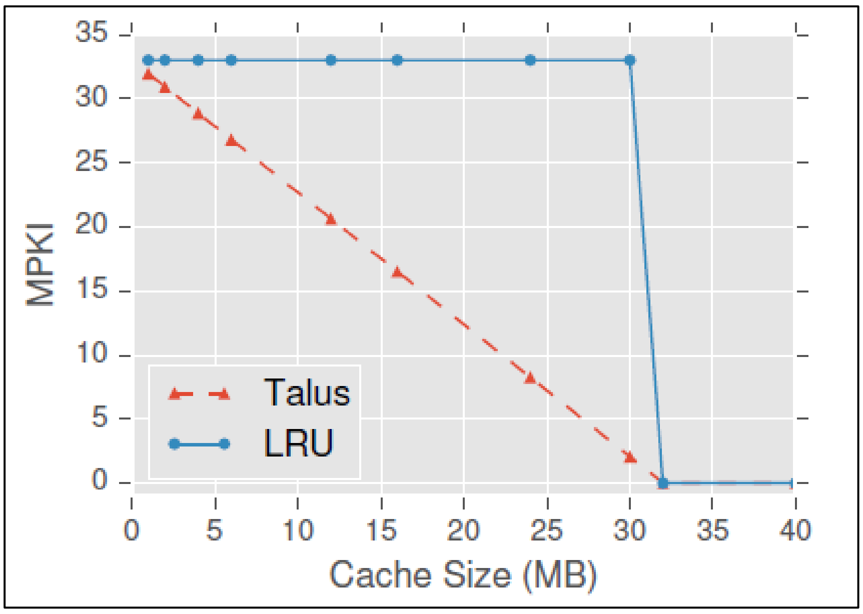


Figure 2.4. Performance of libquantum over cache sizes. LRU causes a performance cliff at 32MB [9]

Manikantan et al. propose 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 algorithm for improving fairness and QoS.

Contrary to most previous work, Li et al. devise a mechanism called Value based Insertion Policy (VIP) which 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 of finding useless lines, especially when the number of partitions is large [12], and propose Futility Scaling. The 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 are 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, coarse time-stamp LRU based implementation of Futility Scaling.

Guney et al. propose an alternative to the Lookahead partitioning algorithm named as the Lookup [4]. The Lookup algorithm utilizes a linear function, which calculates a score for each core, periodically. In the offline phase of the algorithm, coefficients are computed by using machine learning techniques on 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 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 which include last level cache space among multiple cores with a market-based approach, where each core tries to obtain most 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; contrary to 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 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 of which insertion policy causes less 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 4 ways of the 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. CACHE ALLOCATION THROUGH THREAD CLASSIFICATION

Our classification-based cache allocation mechanism requires a set of sequential and parallel steps, as shown in Figure 3.1[3]. The mechanism has a periodic nature, and we collect cache statistics within each epoch. An epoch may last millions of cycles, and, at the end of each epoch, the Cache Allocator is triggered to either change or keep the cache allocation decisions. Here, in this section, we describe the steps of our allocation mechanism, in further detail.

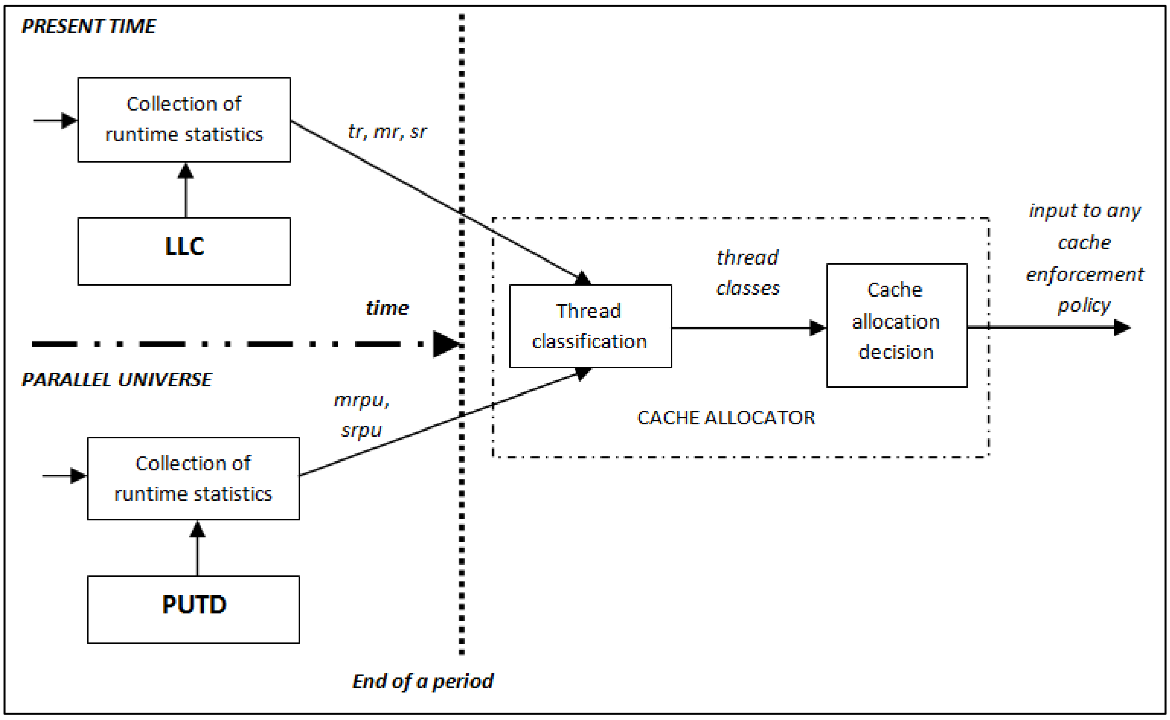

PRESENT TIME
Collection of runtime statistics
LLC
tr, mr, sr
time
PARALLEL UNIVERSE
mrpu, srpu
Collection of runtime statistics
PUTD
End of a period
Thread classification
thread classes
Cache allocation decision
CACHE ALLOCATOR
input to any cache enforcement policy

Figure 3.1. Steps of the proposed cache allocation mechanism

[3] The allocation strategy is way-based, and most of the prior works in the literature choose the same strategy. However, the granularity of the output of our mechanism can be easily mapped to more scalable and fine-grain enforcement algorithms such as Vantage.

3.1. COLLECTION OF RUNTIME STATISTICS

The first step of our proposed mechanism focuses on collecting crucial cache-related statistics about each running thread in the present time and in each thread's parallel universe. Each thread keeps track of a dedicated Tag Directory, which is physically identical to the Auxiliary Tag Directory (ATD) structure described in the original UCP study [1]. However, we named our structure as the Parallel Universe Tag Directory (PUTD) since its task is totally different. A PUTD does not only hold cache tags for a specific thread as an ATD structure does. It is also updated by the other running threads just as a shared LLC. While the LLC is updated with the current allocation decisions, the PUTD structures are updated as if their corresponding threads hold more cache resources. For instance, on an 8-way LLC, if the current allocation decision is “four ways to thread A and four ways 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 “one way to thread A and seven ways to thread B" decision. Specifically, at the beginning of each epoch, the mechanism forks and run multiverse, where it investigates if each thread behaves differently among others when it is supplied with more cache resources.

During this first step, several cache statistics are collected from the multiverse structures by the help of a few hardware counters. These statistics are cache traffic rate (*tr*), miss rate (*mr*) and steal rate (*sr*) of running threads. Equation 3.1 shows how *tr* value of a thread is calculated. The *tr* value is simply *N*/*E*, where N represents the number of cache accesses from a thread and *E* represent the epoch duration for allocation decisions in Kcycles. The *tr* value is a direct indicator of a cache activity of a thread, and when it is low, then, we can safely assume that the corresponding thread has no harm on other threads.

$$tr = N \;/\; E \tag{3.1}$$

The second parameter, the *mr* value, which is simply *M*/*N*, where *M* represents number of cache misses, and *N*, again, represents the number of cache accesses, is an indicator of the amount of the wasteful activity of a thread, and, when it is high, we can be certain that the amount of cache resources available to that thread is hardly utilized. Note that when *mr* is 1, all cache accesses generate cache misses. Finally, the *sr* value is an indicator of the public order within the processor, and when it is high, it may trigger a chain of cache steals, which, in turn, introduce a noticeable performance drop in the overall system. To calculate the *sr* value, we counted the number of evictions from the cache lines that do not belong to the thread that initiates the cache access. In UCP mechanism, when a cache access triggers a cache miss, the number of cache lines that belongs to the current thread is counted. If that number is less than the number of target allocations for that thread, evictions are done from other threads. Here, we apply the same eviction policy and assume that those evictions are cache steals accounted for the thread that actually triggers them. Note that when *sr* is 1, all cache misses cause cache steals. In Figure 3.1, we also show that we collect the miss rate and the steal rate from a thread's parallel universe through its PUTD structure. We name these two variables miss rate from parallel universe (*mrpu*) and steal rate from parallel universe (*srpu*).

We believe that these statistics are indicative enough to identify the behavior of a thread among others, and the number of cache ways a thread deserves can be determined by a careful observation of these statistics within the thread classification step of the mechanism. The implementation of this step on hardware might be quite expensive, though. To reduce its hardware complexity, we choose a method known as Dynamic Set Sampling (DSS) proven in one of the milestone papers [6]. Once a sufficient number of DSS sets are chosen for collection of these statistics, it is analytically proven that they become representative enough of the actual LLC traffic.

3.2. THREAD CLASSIFICATION

In our study, we categorize threads in four classes: *Very Harmful*, *Harmful*, *Harmless* and *Null* (We add an additional category named as *Special* but we will discuss it later in Section 3.4). As its name implies, a *Harmful* thread is a thread that has a disruptive power on the execution performance of others. If its degree of disruption is devastating, then it can be classified as *Very Harmful*. A *Harmless* thread, on the other hand, is a thread that can continue its execution in harmony with others. Finally, a *Null* thread has no interest on LLC space. The LLC is a shared resource, and we cannot assume that all of the running threads will be in similar characteristics. For instance, a thread with high *tr*, *mr* and *sr* values can be easily classified as a *Very Harmful* thread to others. However, another thread with similar *mr* and *sr* but low *tr* can be classified as totally *Harmless* to others. Note that the thread classification step does not try to guess the class of each thread when they are running standalone. On the contrary, it classifies threads when they coexist and run among others. We can further clarify this by an example: A thread might have high *tr* and low *mr* and *sr* values in its standalone run. But, this does not necessarily make the thread *Harmless*. When, it is run with many other threads, its *mr* and *sr* values may be much higher due to cache conflicts and cache steals. As a result, the classification mechanism might be forced to classify the thread as *Harmful*, indicating that the thread might be *Harmful* to the performance of others.

To implement the thread classifier, we devise an empirical classification function as shown in Equation 3.2. For each of the threads, the class is determined by calculating three thread-specific functions, *T* for the cache traffic, *M* for the cache miss rate and *S* for the cache steal rate. Consequently, these functions help us to discretize and scale down the raw values of *tr*, *mr* and *sr*. After running a series of tests and collecting runtime statistics on actual SPEC 2006 traces, we generated a set of distinct low ($Thresh_{lo}$) and high ($Thresh_{hi}$) thresholds for *T*, *M* and *S* functions and one additional ($Thresh_{mid}$) for *M*. Equations 3.3 through 3.5 show how a raw value is quantized by the *T*, *M* and *S* functions, respectively. Note that *M* function has an exceptional case that can immediately bump up its value to 8. By doing so, when a thread experiences extremely high cache miss rates, we can directly set its class to *Very Harmful*.

$$ThreadClass = T(tr) \times (1 + M(mr) \times (1 + S(sr))) \tag{3.2}$$

$$T(tr) = \begin{cases} 0 & tr < TThresh_{lo} \\ 1 & tr \geq TThresh_{lo} \cap tr < TThresh_{hi} \\ 2 & tr \geq TThresh_{hi} \end{cases} \tag{3.3}$$

$$M(mr) = \begin{cases} 0 & mr < MThresh_{lo} \\ 1 & mr \geq MThresh_{lo} \cap mr < MThresh_{mid} \\ 2 & mr \geq MThresh_{mid} \cap mr < MThresh_{hi} \\ 8 & mr \geq TThresh_{hi} \end{cases} \tag{3.4}$$

$$S(sr) = \begin{cases} 0 & sr < SThresh_{lo} \\ 1 & sr \geq SThresh_{lo} \cap sr < SThresh_{hi} \\ 2 & sr \geq SThresh_{hi} \end{cases} \tag{3.5}$$

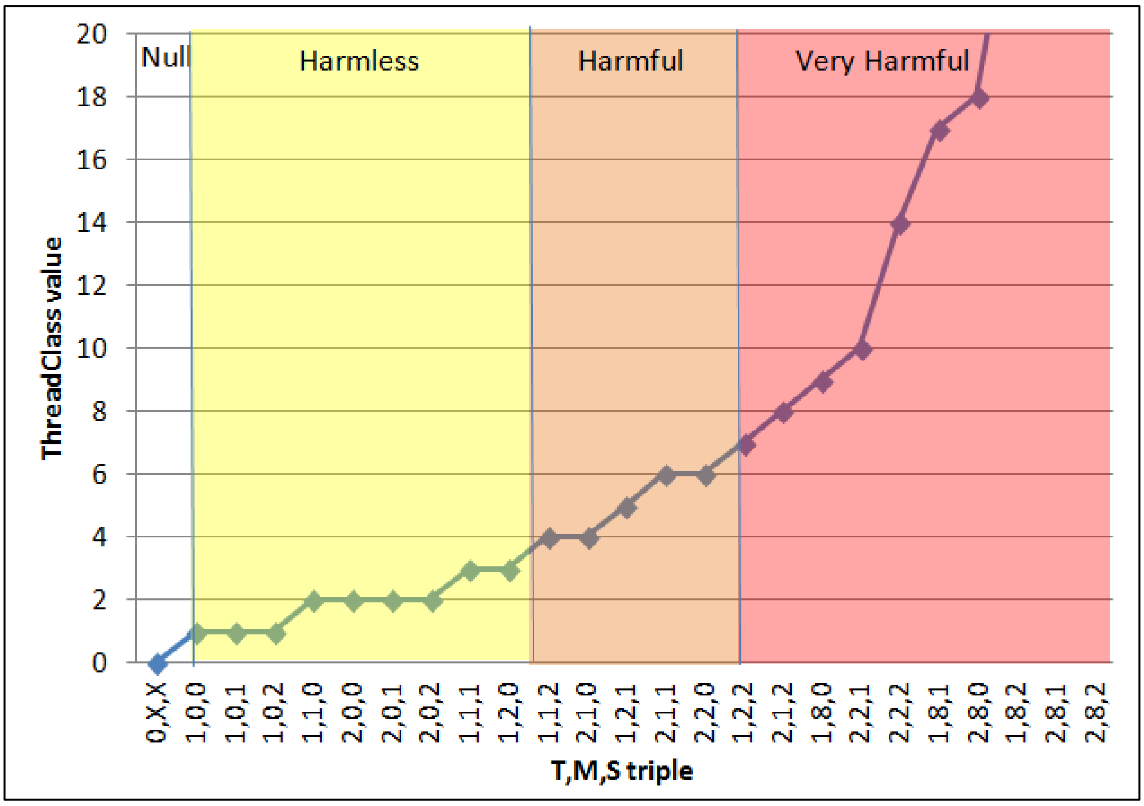


Figure 3.2. Classification through the *ThreadClass* variable

Figure 3.2 shows the exponentially increasing *ThreadClass* curve, which depends on various <*T*, *M*, *S*> combinations. The boundaries between the classes are empirically determined. Note that the *T* value itself is sufficient for a *Null* type classification, since it is the multiplier in Equation 3.2. As long as the thread generates very low cache traffic, high miss rate and high steal rate does not mean anything, and the thread we consider should be classified as a *Null* thread, which does not receive any cache lines during the allocation decision stage.

However, when the *T* value is not equal to zero, we have to consider the *M* and *S* values, as well.

In Equation 3.2, *T* and *M* values are multiplied. This implies that both *T* and *M* values are quite significant for the classification process. When the *T* and *M* values are both high at the same time, the thread immediately moves to higher classes. Additionally, in Equation 3.2, *M* and *S* values are also multiplied. This indicates that when a thread generates a very low cache miss rate, its cache steals become insignificant for the classification process. However, when the *T*, *M* and *S* values are all high at the same time, the thread immediately moves to the highest class. From the Figure 3.2, we see that when the *ThreadClass* value is higher than 6, we can safely assume that the thread is *Very Harmful* to others in its current context. However, what we do not really know is if the thread would still be in the same class when it had received more cache resources. To get a close estimate on this issue, we also calculate the $ThreadClass_{pu}$ value, which is available from the thread's parallel universe, as shown in Equation 3.6. Note that the *tr* value is assumed to be identical for both the present time and the parallel universe calculations. This is somewhat necessary since we have a single execution trace, and, for the time being, we do not focus on the estimation of the cache traffic rate on parallel universes. But, this may be definitely a direction for further research on this topic.

$$ThreadClass_{pu} = T(tr) \times (1 + M(mrpu) \times (1 + S(srpu))) \tag{3.6}$$

When *ThreadClass* and $ThreadClass_{pu}$ are evaluated together, the class of a thread may be more accurately estimated. Continuing the same example given above, when the $ThreadClass_{pu}$ value of the same thread is also greater than 6, then, the classifier can conclude that the thread has a cache thrashing behavior and set its final decision. However, when its $ThreadClass_{pu}$ value is 4, the classifier becomes more reluctant to classify the thread as *Very Harmful*, since the thread promises a well-behaved characteristic with additional cache resources.

Note that the Equation 3.6 does not guarantee that the $ThreadClass_{pu}$ value will always be lower, i.e. the thread will be less harmful, when it is supplied with more cache resources.

For instance, a thread with a <*T*, *M*, *S*> of <2, 1, 0> may turn into a thread with <*T*, *M*, *S*> of <2, 1, 1>. This directly shows us that the thread is a *Harmful* thread and giving additional cache ways will make its behavior no different since it will start stealing cache lines from others. On the other hand, if the same thread turns into a <2, 1, 2>, then, it is classified as *Very Harmful* in its multiverse. In this study, we decided to return the minimum *ThreadClass* value that is gathered from both execution paths.

Throughout this study, we allocate three more cache ways to each target thread by reducing the allocations of remaining threads by one cache way, within each PUTD structure. For instance, for a 4-core processor with a 16-way LLC configuration, in the PUTD structure of the first thread, the first thread receives three extra cache ways by taking one cache way from each of the other three threads. In higher processor configurations, such as an 8-core processor with a 32-way LLC, the threads that lose a cache way are arbitrarily chosen.

3.3. ADAPTIVE TRAFFIC THRESHOLD MECHANISM

In the first phase of our study, 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 scalable approach for every cache configuration and organization, and an adaptive traffic threshold mechanism is needed. To overcome this problem, indicators of cache traffic of threads are measured with relative to each other and the total cache traffic. Equations from 3.7 through 3.10 show the measurement of the traffic value (*tr*).

$$ir_1 = Tot_{Access}/A \tag{3.7}$$

$$tr_1 = N/ir_1 = N \times A/Tot_{Access} \tag{3.8}$$

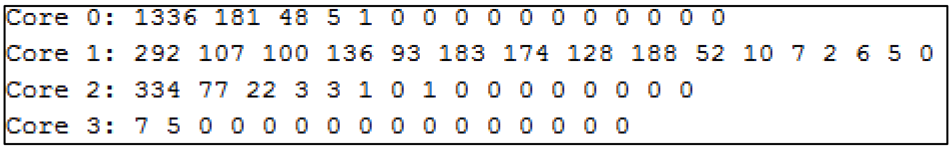
$$ir_2 = Tot_{Access}/EA \tag{3.9}$$

$$tr_2 = N/ir_2 = N \times EA/Tot_{Access} \tag{3.10}$$

Here, *tr* of an application is calculated in two iterations. At first iteration, in Equation 3.7, ideal number of accesses (*ir*) of an application is calculated by the total number of accesses of all applications in each specific epoch divided by the number of applications. This ideal rate represents the expected rate of access of an application compared to accesses of all applications. Then, in Equation 3.8, *tr* of an application is compared to *ir* by number of accesses of application (*N*) divided by *ir*. Eligible applications (*EA*) are determined by whether *tr* of first iteration is beyond the $TThresh_{lo}$ or not. So at first iteration, *tr* values of all applications are calculated, and number of eligible applications is determined. At the second iteration, in Equation 3.9, ideal rate is calculated again with the eligible applications. It is derived from total accesses of all applications divided by the number of eligible applications. Since the new ideal rate is calculated in second iteration, *tr* value can be updated with this new ideal rate in Equation 3.10. As a result of these two iterations, *tr* value is finalized. Then, as in Section 3.2., *tr* value is compared against $TThresh_{hi}$ and $TThresh_{lo}$. Those threshold values are obtained from a set of simulations. 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 as a high traffic. If its traffic rate goes down below the $TThresh_{lo}$ (0.125 - 1/8), it is considered as *Null* thread.

3.4. SPECIAL THREAD

In our early classification mechanism, there are four 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 incorrect classification. Figure 3.3 shows this problem. Hit counts in the figure was obtained from ATD structure of Lookahead implementation. Benchmarks in the example are *bwaves, deal, sjeng,* and, *mcf* respectively. It can be cleary seen that, *deal* has 52 hits even in the 10th cache way while others have none.



```
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 3.3. Hit counts for *bwaves-deal-sjeng-mcf* benchmark combination

To solve this problem in our Classifier, we utilize the PUTD which is similar to ATD of UCP, to observe whether a thread can still receive cache hits from the tenth cache way. We give highest priority to these so called *Special* threads by assigning $4 \times W$ in our classification mechanism.

3.5. ALLOCATION DECISION

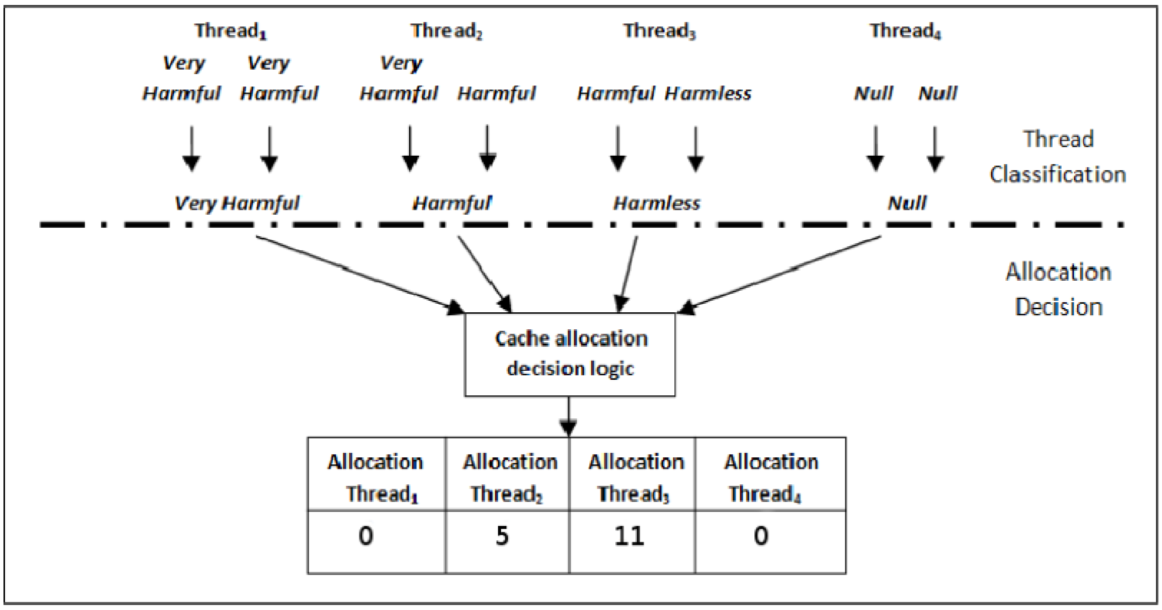
This final step decides on the amount of cache ways each thread is to receive for the next epoch. When all the thread classes are determined to be the same, then the allocation task is pretty straightforward, i.e. all the threads receive the same amount of cache ways. However, the cache allocation mechanism should be ready for any class combination, and this requirement makes its design much harder. The allocator can be implemented by a small logic circuit, which accepts multiverse thread classes and generates output for their corresponding cache allocations.

To simplify the process, we decided to give zero cache ways to both *Null* and *Very Harmful* threads, since it is evident that threads with both types do not effectively utilize the LLC. Then, the decision circuitry only focuses on allocating ways for *Harmless* and *Harmful* threads. To further simplify the decision process, first, we created a Unit function to calculate the number of unit cache ways for assigning to each *Harmful* thread as shown in the sixth line of Algorithm 3.1. Here, Ways represents the number of total LLC cache ways, *NL* represents the number of *Harmless* threads, *NH* represents the number of *Harmful* threads, and, finally, *W* represents the weight of a *Harmless* thread over a *Harmful* thread. In this study, we empirically fixed the value of *W* to 2, indicating that a *Harmless* thread should receive twice as many cache ways as of a *Harmful* thread does (We give value of *W* to 4 for the Special thread which we explain in next section). Finally, at the end of each epoch, Algorithm 3.1 is used by the allocator to set the number of allocations.

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

    $Unit = Ways/(W \cdot NL + NH)$
    $allocations[i]$ = 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
```

Algorithm 3.1. Classifier allocation algorithm

Notice that our allocation mechanism does not guarantee strict cache way isolation among threads as the total number of allocated cache ways might be higher than the number of available physical cache ways. For instance, after functions are applied in our algorithm, *Harmful* thread in Figure 3.4, receives 6 cache ways instead of 5. This implies that there might be cache conflicts between the *Harmless* and the *Harmful* threads over a cache line. In this study, we gladly pay this type of a penalty for the sake of simpler decision hardware.



Allocation Thread$_1$	Allocation Thread$_2$	Allocation Thread$_3$	Allocation Thread$_4$
0	5	11	0

Figure 3.4. Steps of the cache allocation decision

The Lookahead and the Classifier mechanisms need a number of addition, comparison, multiplication, and division operations. Since both mechanisms can be implemented sequentially where a single arithmetic logic unit would be sufficient for each operation, we evaluate and compare the worst case scenarios of both mechanisms in terms of micro-operations and latency. Moreover, power requirements of these mechanisms might be easily related to the number of micro-operations required. Thus, the comparison on the number of micro-operations also gives us a rough idea to compare the power consumption of both mechanisms.

The Lookahead requires that each core calculate the maximum marginal utility until all cache ways are allocated, which requires addition, division, and comparison operations proportional to $C \times N$, where C is the number of cores and N is the associativity. In the worst case scenario, cache ways are allocated to cores one by one, in which case way allocation will be repeated N times, thus total number of addition, comparison, and division operations will be proportionate to $C \times N^2$.

The Classifier, on the other hand, uses only a few addition, division, and comparison operations per core in order to determine the class of each core. Once these classes are determined, one addition, one division, and two multiplications are carried out in order to compute U and $U \times W$. Lastly, Classifier uses two comparisons per core in order to assign cache ways according to their classes. It makes a significant difference in terms of micro-operations and latency that the number of operations required by the Classifier is only affected by the number of cores, and is independent of cache associativity.

We show that our mechanism requires only a fraction of operations and clock cycles required by the Lookahead mechanism in three processor configurations, as shown in Figure 3.5. Intel Skylake architecture is chosen for the basis of comparison, and operation costs are taken from [14]. According to the complexity analysis above, number of required addition, division, comparison, and multiplication operations were calculated and multiplied with their respective costs in Skylake architecture for each configuration. Addition, comparison and division operations are assumed to be 16-bit, whereas the multiplication operation is assumed to be 8-bit. All operations are assumed to be integer operations.

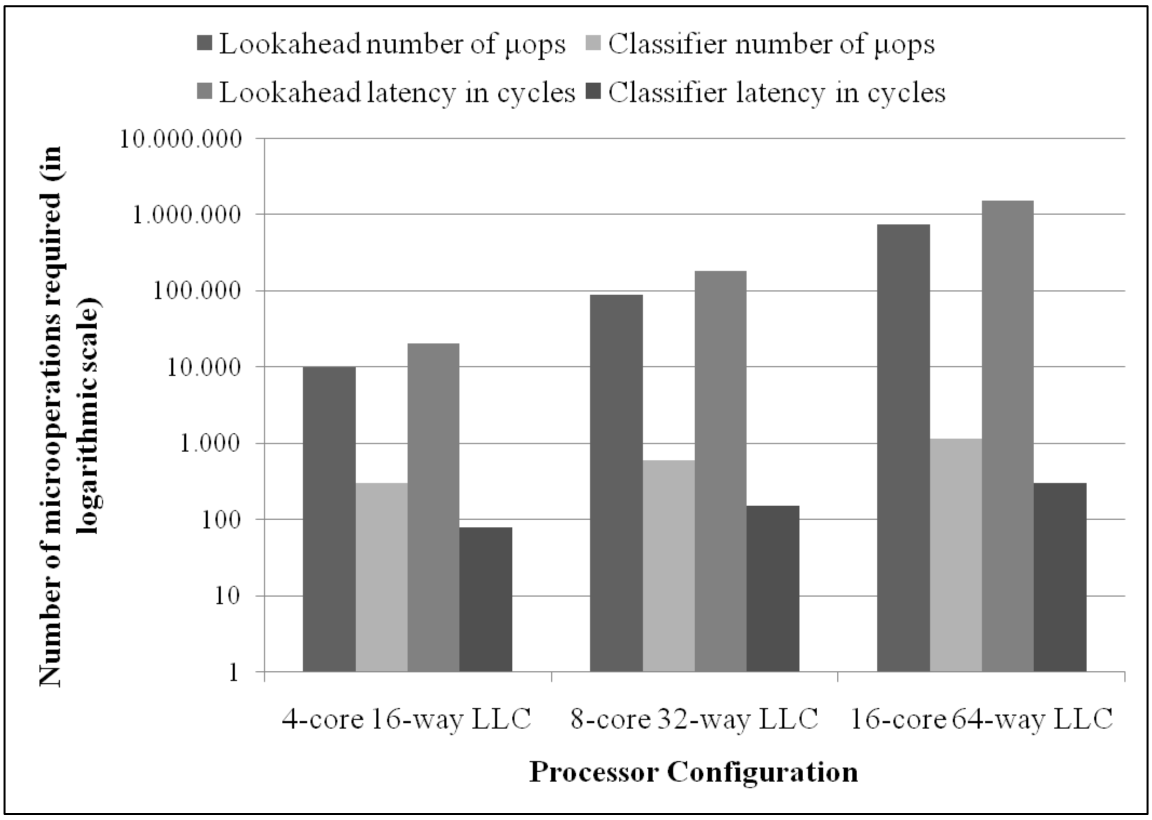


Figure 3.5. Comparison of Classifier with Lookahead in terms of required micro-operations and latency (in cycles) for completion in worst ca se scenarios, in logarithmic scale

We evaluate the additional area requirement of our design depending only on the introduction of PUTD structures as the remaining components require much smaller area and therefore can be ignored. Qureshi and Patt show that the ATD structure required for UCP takes about only 0.17 per cent of LLC cache area for a 1MB, 16-way cache [1]. This ratio will almost be fixed as the associativity of the LLC changes. The area requirements of PUTD structures of the Classifier would be equal to 0.17 per cent times number of cores, i.e. 0.68 per cent of the LLC area for a 4-core configuration, 1.36 per cent for an 8-core configuration and 2.72 per cent for a 16-core configuration.

4. INTEGRATION ISSUES

When it comes to integration with the existing partitioning mechanisms, there are two conflicting strategies for our proposed mechanism:

i. One-Design-Fits-All strategy
ii. Tailor-Made Design strategy.

4.1. ONE-DESIGN-FITS-ALL (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 the 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 on PIPP [2], on Vantage [3] or on 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 3.1 (i.e., the LLC box is replaced with a PUTD box, or more precisely, with a Present Time Tag Directory (PTTD) box, in the figure).

4.2. TAILOR-MADE DESIGN (TMD) STRATEGY

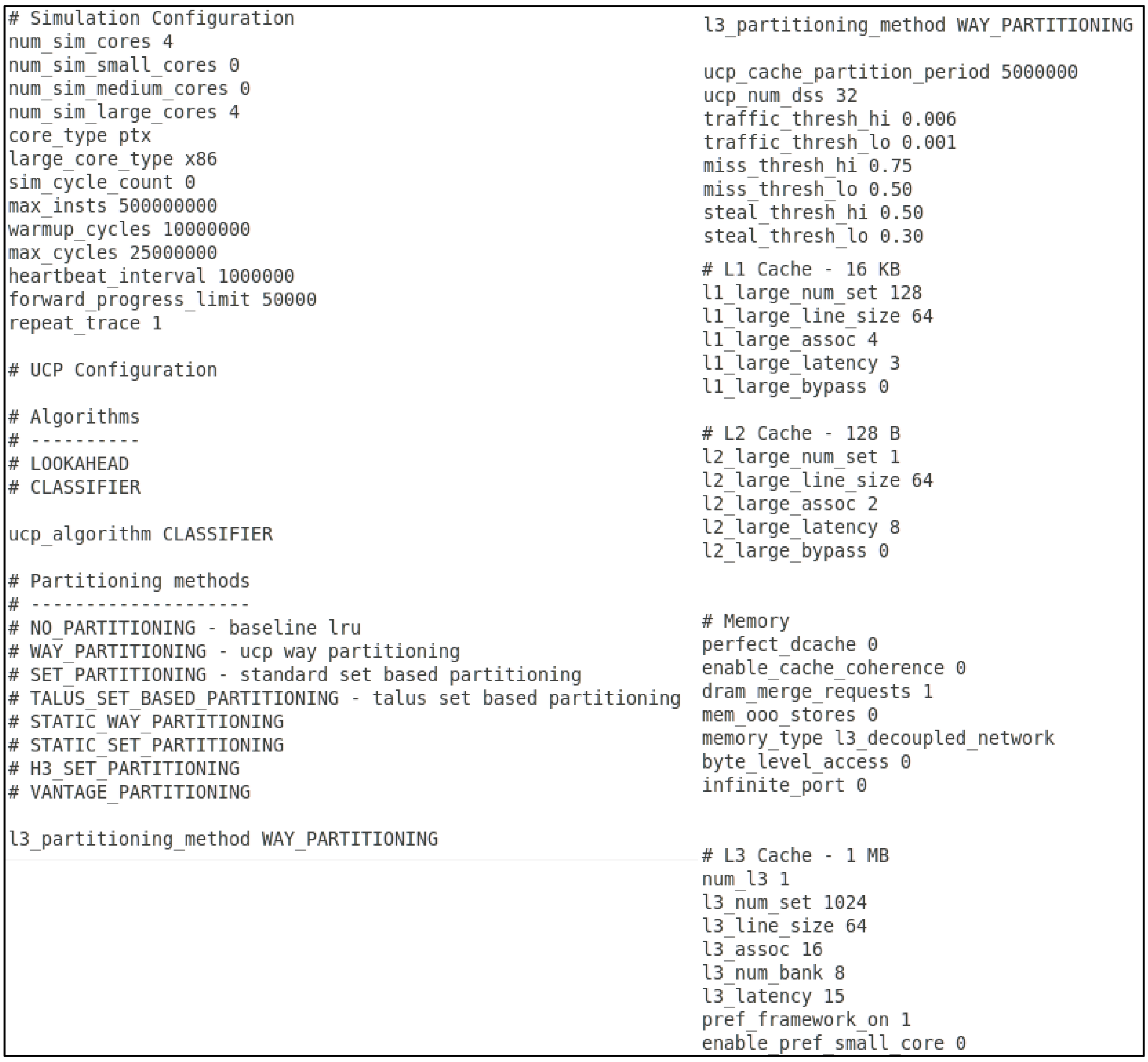
Each cache partitioning enforcement mechanism has its own assumptions that might not always be appropriate for the ODFA strategy. For instance, PIPP [2] completely changes the promotion/insertion policy of the cache, and Vantage [3] assumes a special cache organization (ZCache) is in place [15]. The TMD strategy is based on a custom design of the allocation policy so that the allocation and the enforcement policies work in harmony. Intuitively, one can believe 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. Though, 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 an integration scenario.

5. DESIGN AND IMPLEMENTATION

Macsim is trace-driven and cycle-level heterogeneous architecture simulator. It systematically simulates architectural behaviors, including detailed pipeline stages, multi-threading, and memory systems [16]. Its written in C++ language and built with phyton. In our implementations, we mainly redesign and extend memory implementations of the simulator i.e. memory.cc and cache.cc source files. We implement state of the art algorithms for comparisons such as UCP, Vantage, PIPP, TADIP and our study Classifier as well.

In the main memory cycle of the simulation, we check for that epoch is reached and trigger our allocation algorithm. Simulator has a file called params.in which contains runtime configurations. We defined additional parameters for algorithm name, threshold values for Classifier, promotion rate for PIPP and other many required parameters such as described in Table 6.1. Figure 5.1. shows set of parameters in params.in file for an example scenario.

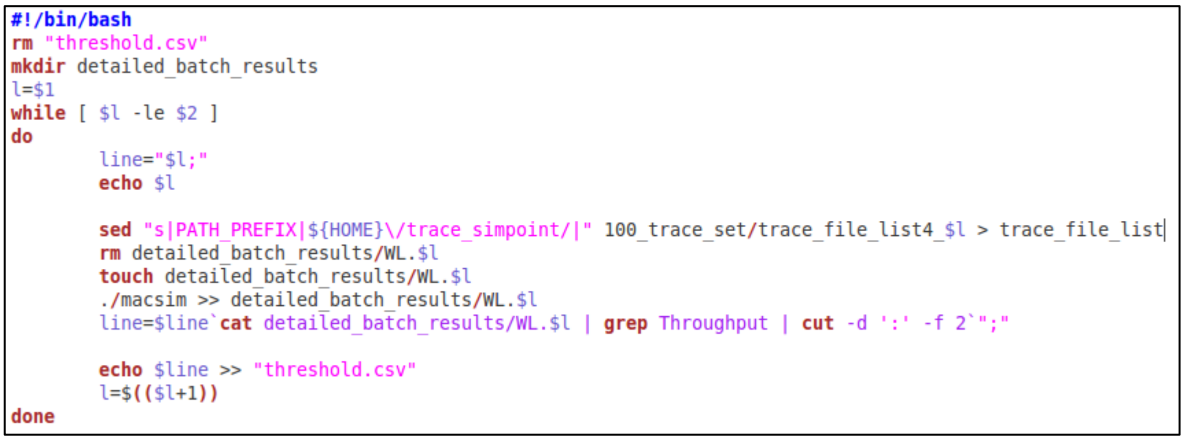


```
# Simulation Configuration
num_sim_cores 4
num_sim_small_cores 0
num_sim_medium_cores 0
num_sim_large_cores 4
core_type ptx
large_core_type x86
sim_cycle_count 0
max_insts 500000000
warmup_cycles 10000000
max_cycles 25000000
heartbeat_interval 1000000
forward_progress_limit 50000
repeat_trace 1

# UCP Configuration

# Algorithms
# ----------
# LOOKAHEAD
# CLASSIFIER

ucp_algorithm CLASSIFIER

# Partitioning methods
# --------------------
# NO_PARTITIONING - baseline lru
# WAY_PARTITIONING - ucp way partitioning
# SET_PARTITIONING - standard set based partitioning
# TALUS_SET_BASED_PARTITIONING - talus set based partitioning
# STATIC_WAY_PARTITIONING
# STATIC_SET_PARTITIONING
# H3_SET_PARTITIONING
# VANTAGE_PARTITIONING

l3_partitioning_method WAY_PARTITIONING
```

```
l3_partitioning_method WAY_PARTITIONING

ucp_cache_partition_period 5000000
ucp_num_dss 32
traffic_thresh_hi 0.006
traffic_thresh_lo 0.001
miss_thresh_hi 0.75
miss_thresh_lo 0.50
steal_thresh_hi 0.50
steal_thresh_lo 0.30
# L1 Cache - 16 KB
l1_large_num_set 128
l1_large_line_size 64
l1_large_assoc 4
l1_large_latency 3
l1_large_bypass 0

# L2 Cache - 128 B
l2_large_num_set 1
l2_large_line_size 64
l2_large_assoc 2
l2_large_latency 8
l2_large_bypass 0


# Memory
perfect_dcache 0
enable_cache_coherence 0
dram_merge_requests 1
mem_ooo_stores 0
memory_type l3_decoupled_network
byte_level_access 0
infinite_port 0


# L3 Cache - 1 MB
num_l3 1
l3_num_set 1024
l3_line_size 64
l3_assoc 16
l3_num_bank 8
l3_latency 15
pref_framework_on 1
enable_pref_small_core 0
```

Figure 5.1. Set of parameters in params.in file

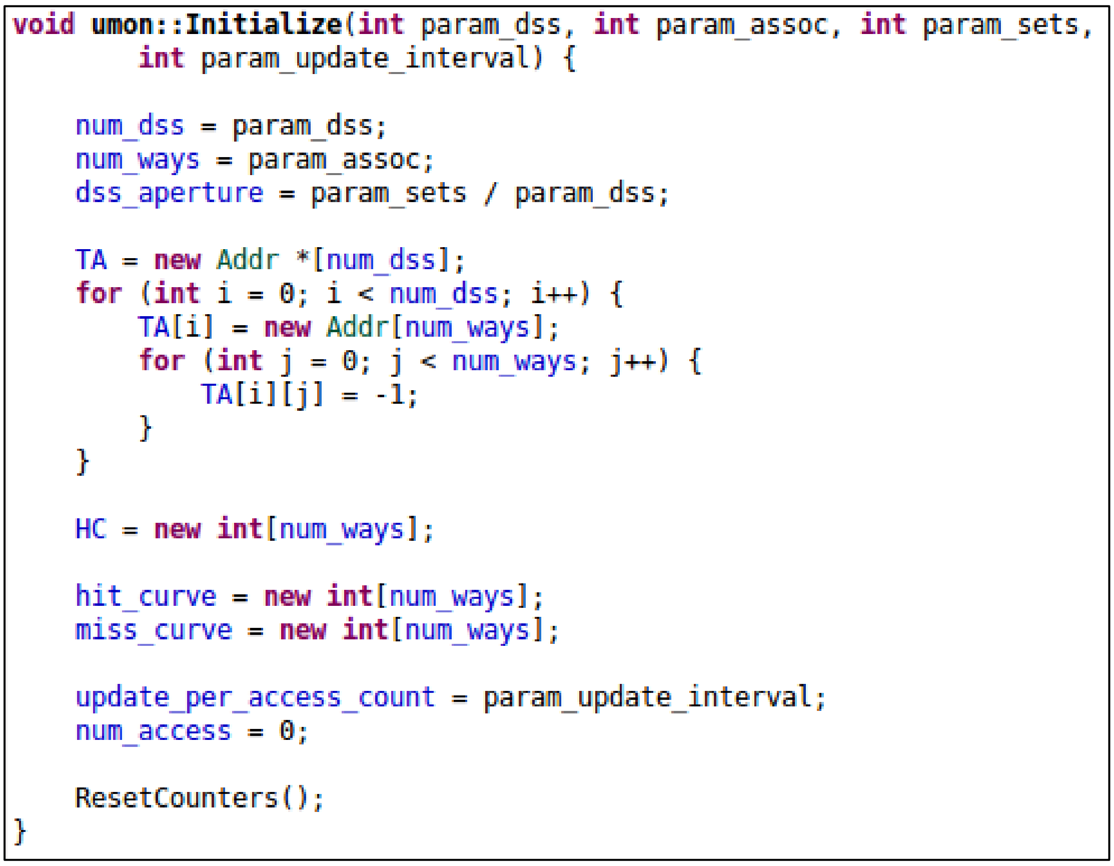
When an epoch is reached, our implementation runs decision algorithms with the help of auxillary structures (ATDs for UCP, Vantage and PIPP, PUTDs for Classifier) and updates allocation array. For every memory operation at each cycle, replacement policy of each algorithm is also applied. For instance, UCP and our study Classifier has the same replacement policy but Vantage, PIPP and TADIP have their own original policies. We implement all of their replacement policies in memory.cc and cache.cc. As a result of those implementations, simulation execution is adapted to each algorithms and results are obtained from the default output file of the macsim. Each of the cores and total throughput values are included of those output files. We prepared a batch script for running all of the workload combinations for all algorithms for 4-core, 8-core and 16-core simulations. Batch script changes the params.in file for choosing algorithms, parameter values and trace files, then collects the throughput values from output files. Figure 5.2. shows the details of the batch script file.



```
#!/bin/bash
rm "threshold.csv"
mkdir detailed_batch_results
l=$1
while [ $l -le $2 ]
do
        line="$l;"
        echo $l

        sed "s|PATH_PREFIX|${HOME}\/trace_simpoint/|" 100_trace_set/trace_file_list4_$l > trace_file_list
        rm detailed_batch_results/WL.$l
        touch detailed_batch_results/WL.$l
        ./macsim >> detailed_batch_results/WL.$l
        line=$line`cat detailed_batch_results/WL.$l | grep Throughput | cut -d ':' -f 2`";"

        echo $line >> "threshold.csv"
        l=$(($l+1))
done
```

Figure 5.2. Details of script.sh file

For UCP, we implement ATDs and UMONs with two dimensional arrays in LLC structure. One dimension for way ID and the other dimension is for DSS set ID. With the help of this three dimension array, we are able to store tag information while simulation executes. Figure 5.3 shows the definition and instantiation of ATD structure in our code. Also we implemented methods for resetting counters, halving counters, Greedy partitioner and Lookahead partitioner as UCP suggests.



```
void umon::Initialize(int param_dss, int param_assoc, int param_sets,
        int param_update_interval) {

    num_dss = param_dss;
    num_ways = param_assoc;
    dss_aperture = param_sets / param_dss;

    TA = new Addr *[num_dss];
    for (int i = 0; i < num_dss; i++) {
        TA[i] = new Addr[num_ways];
        for (int j = 0; j < num_ways; j++) {
            TA[i][j] = -1;
        }
    }

    HC = new int[num_ways];

    hit_curve = new int[num_ways];
    miss_curve = new int[num_ways];

    update_per_access_count = param_update_interval;
    num_access = 0;

    ResetCounters();
}
```

Figure 5.3. Instantiation of ATD structures i.e. Tag Arrays(TA) and Hit Counters (HC)

For TADIP, we only implement replacement policy since it does not force any allocation policy. TADIP changes only insertion policy of cache organization. It gathers feedbacks from current threads and decide between BIP (Bipolar Insertion Policy) or LRU. BIP basically inserts replacing cache lines into MRU position with epsilon probability. i.e epsilon=$1/2^9$ is taken for simulation. Otherwise, it inserts into LRU position (Base LRU policy inserts into MRU position.). For this way, threads that polluting cache (i.e. streaming applications) will be restricted to only LRU position. For fairness, with epsilon probability, it inserts into MRU position. Deciding between BIP and LRU is managed with a saturation counters. Samples from cache sets are monitored with those counters for each thread. Some sets are always use BIP policy, some others use LRU policy. Set selection logic is shown in Figure 5.4. Each thread has a saturation counter. For every miss on BIP, counter increased and for every miss on LRU, counter is decreased. Other follower sets that are not monitored use those counters to decide. Behaviour of other threads are used as a feedback in counters. If MSB is 0 LRU policy is chosen and if it 1, BIP is chosen.

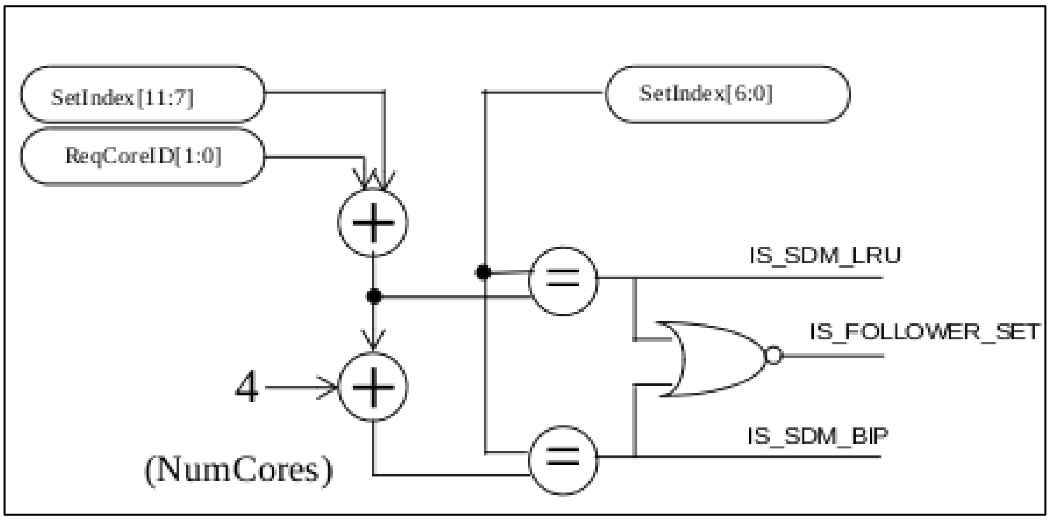
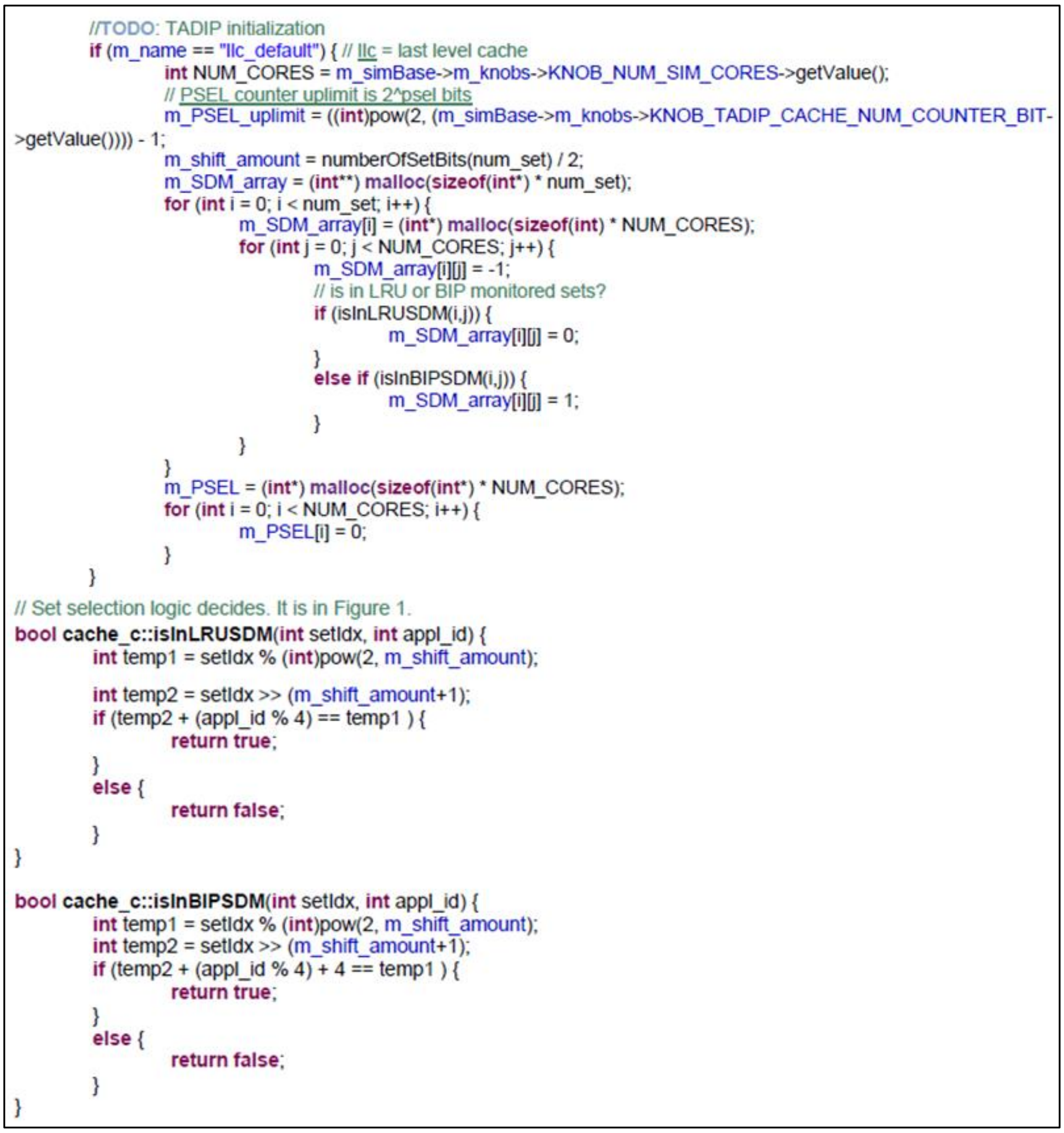


Figure 5.4. Set selection logic of TADIP

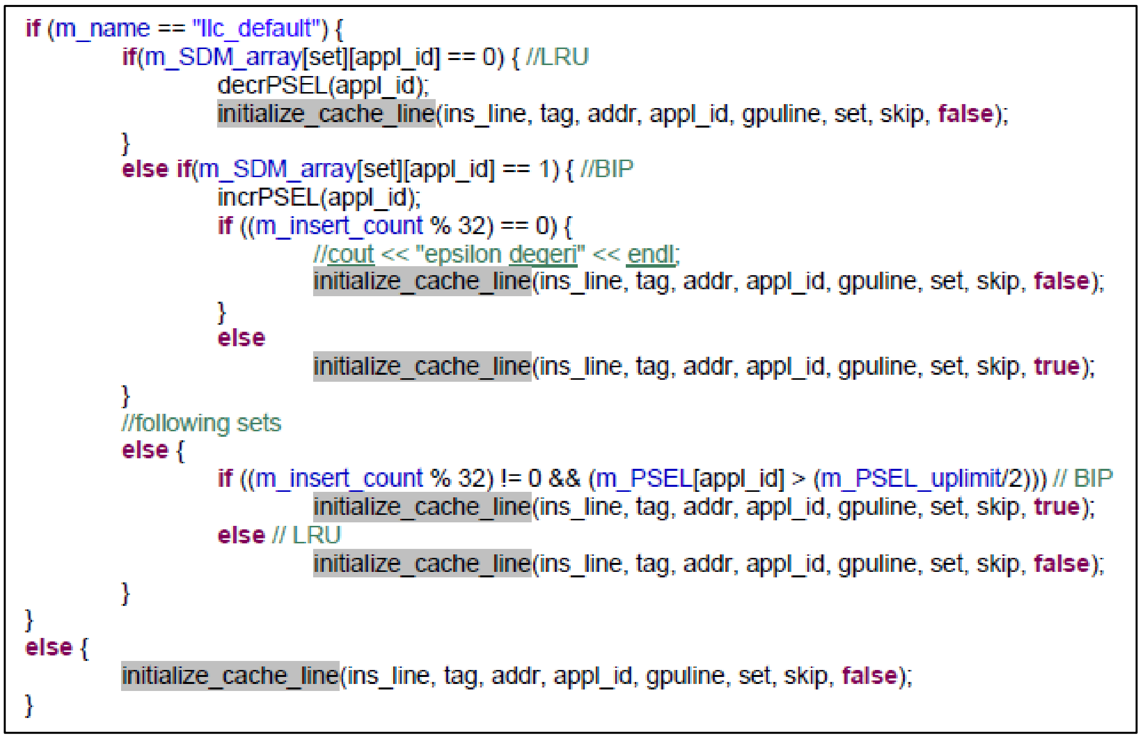
While initiating, in constructor of cache(), TADIP overhead is initiated in Figure 5.5.



```
		//TODO: TADIP initialization
		if (m_name == "llc_default") { // llc = last level cache
				int NUM_CORES = m_simBase->m_knobs->KNOB_NUM_SIM_CORES->getValue();
				// PSEL counter uplimit is 2^psel bits
				m_PSEL_uplimit = ((int)pow(2, (m_simBase->m_knobs->KNOB_TADIP_CACHE_NUM_COUNTER_BIT-
>getValue()))) - 1;
				m_shift_amount = numberOfSetBits(num_set) / 2;
				m_SDM_array = (int**) malloc(sizeof(int*) * num_set);
				for (int i = 0; i < num_set; i++) {
						m_SDM_array[i] = (int*) malloc(sizeof(int) * NUM_CORES);
						for (int j = 0; j < NUM_CORES; j++) {
								m_SDM_array[i][j] = -1;
								// is in LRU or BIP monitored sets?
								if (isInLRUSDM(i,j)) {
										m_SDM_array[i][j] = 0;
								}
								else if (isInBIPSDM(i,j)) {
										m_SDM_array[i][j] = 1;
								}
						}
				}
				m_PSEL = (int*) malloc(sizeof(int*) * NUM_CORES);
				for (int i = 0; i < NUM_CORES; i++) {
						m_PSEL[i] = 0;
				}
		}
// Set selection logic decides. It is in Figure 1.
bool cache_c::isInLRUSDM(int setIdx, int appl_id) {
		int temp1 = setIdx % (int)pow(2, m_shift_amount);

		int temp2 = setIdx >> (m_shift_amount+1);
		if (temp2 + (appl_id % 4) == temp1 ) {
				return true;
		}
		else {
				return false;
		}
}

bool cache_c::isInBIPSDM(int setIdx, int appl_id) {
		int temp1 = setIdx % (int)pow(2, m_shift_amount);
		int temp2 = setIdx >> (m_shift_amount+1);
		if (temp2 + (appl_id % 4) + 4 == temp1 ) {
				return true;
		}
		else {
				return false;
		}
}
```

Figure 5.5. Initialization of TADIP

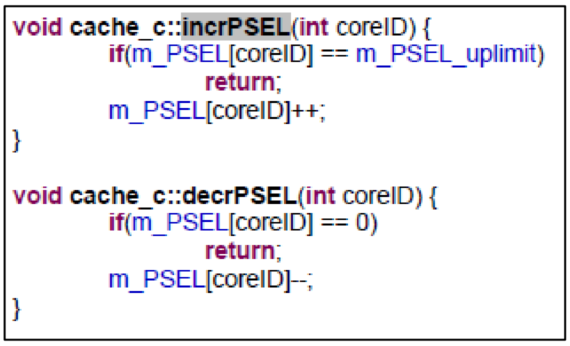
Once cache is initied like this, insert_cache() method is modified. Since TADIP does not include any policy for cache hits, it has policies only on cache miss, i.e. insert_cache(). This method initializes a new cache line while inserting. A new boolean parameter called isLIP is added as a parameter into this method in order to decide the insertion position. If it is LIP (LRU insertion policy), it inserts into LRU position, it can be called only for BIP policy, or if it is not, it inserts into MRU position, i.e. LRU policy. But how this boolean is decided is shown in Figure 5.6.



```
if (m_name == "llc_default") {
        if(m_SDM_array[set][appl_id] == 0) { //LRU
                decrPSEL(appl_id);
                initialize_cache_line(ins_line, tag, addr, appl_id, gpuline, set, skip, false);
        }
        else if(m_SDM_array[set][appl_id] == 1) { //BIP
                incrPSEL(appl_id);
                if ((m_insert_count % 32) == 0) {
                        //cout << "epsilon degeri" << endl;
                        initialize_cache_line(ins_line, tag, addr, appl_id, gpuline, set, skip, false);
                }
                else
                        initialize_cache_line(ins_line, tag, addr, appl_id, gpuline, set, skip, true);
        }
        //following sets
        else {
                if ((m_insert_count % 32) != 0 && (m_PSEL[appl_id] > (m_PSEL_uplimit/2))) // BIP
                        initialize_cache_line(ins_line, tag, addr, appl_id, gpuline, set, skip, true);
                else // LRU
                        initialize_cache_line(ins_line, tag, addr, appl_id, gpuline, set, skip, false);
        }
}
else {
        initialize_cache_line(ins_line, tag, addr, appl_id, gpuline, set, skip, false);
}
```

Figure 5.6. Decision of insertion position of TADIP

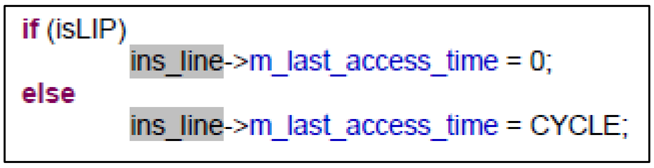
Incresing and decreasing PSEL counter is straightforward. It only checks for upper bound which is described before and lower bound which is equal to=0. See Figure 5.7.



```
void cache_c::incrPSEL(int coreID) {
        if(m_PSEL[coreID] == m_PSEL_uplimit)
                return;
        m_PSEL[coreID]++;
}

void cache_c::decrPSEL(int coreID) {
        if(m_PSEL[coreID] == 0)
                return;
        m_PSEL[coreID]--;
}
```

Figure 5.7. Increasing and decreasing PSEL counter of TADIP

Finally, in initialize cache line method, setting last access time of the cache line is decided as in Figure 5.8.



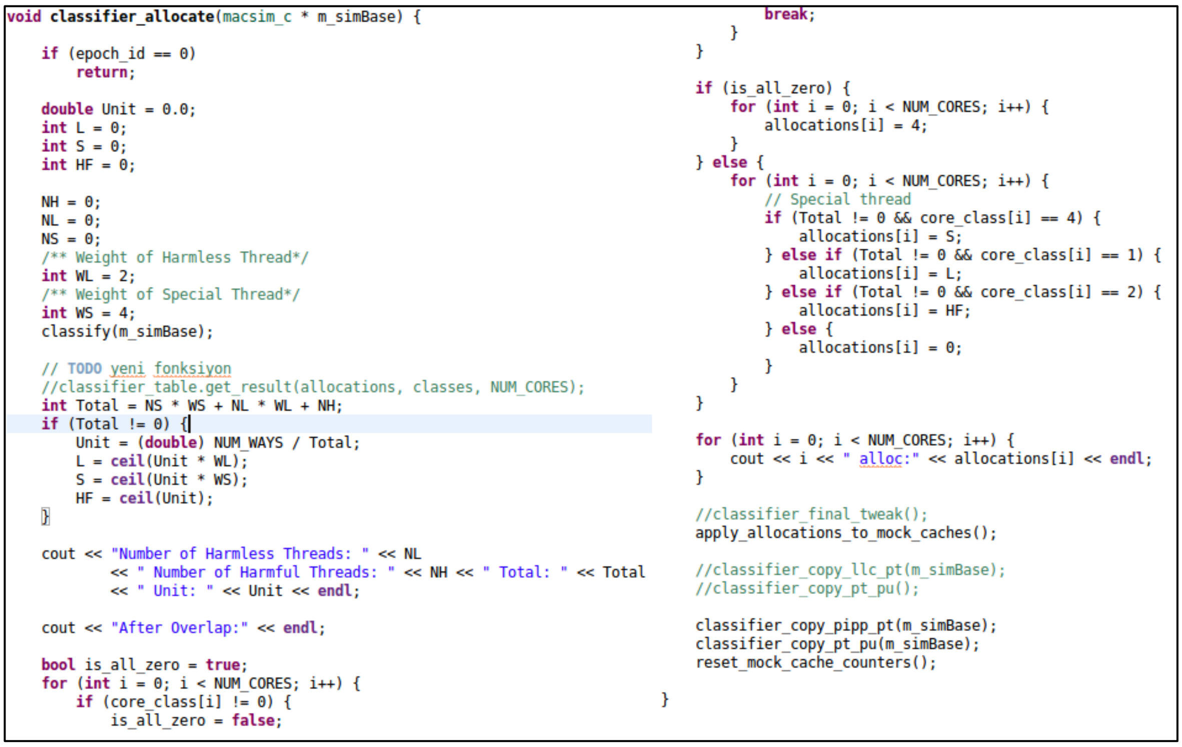
```
if (isLIP)
        ins_line->m_last_access_time = 0;
else
        ins_line->m_last_access_time = CYCLE;
```

Figure 5.8. Determination of isLIP boolean in TADIP

For PIPP, we use same ATD structures and allocation methods as it uses Lookahead algoritgm for allocation. However, PIPP uses different replacement policy with insertion pointers, promotion and demotion processes. We changed replacement policy of Lookahead according to PIPP and created a package for further implementations of ODFA and TMD case study which described in Section 5.5. For ODFA, we combined Classifier with PIPP for only present time in multiverse. For TMD, Classifier run with PIPP package multiverse.

For Vantage, again we use same ATD structures and allocation methods as it uses Lookahead algorithm. However, Vantage has managed and unmanaged regions for replacement policy. We also implemented those structures in memory.cc and cache.cc. accordingly.

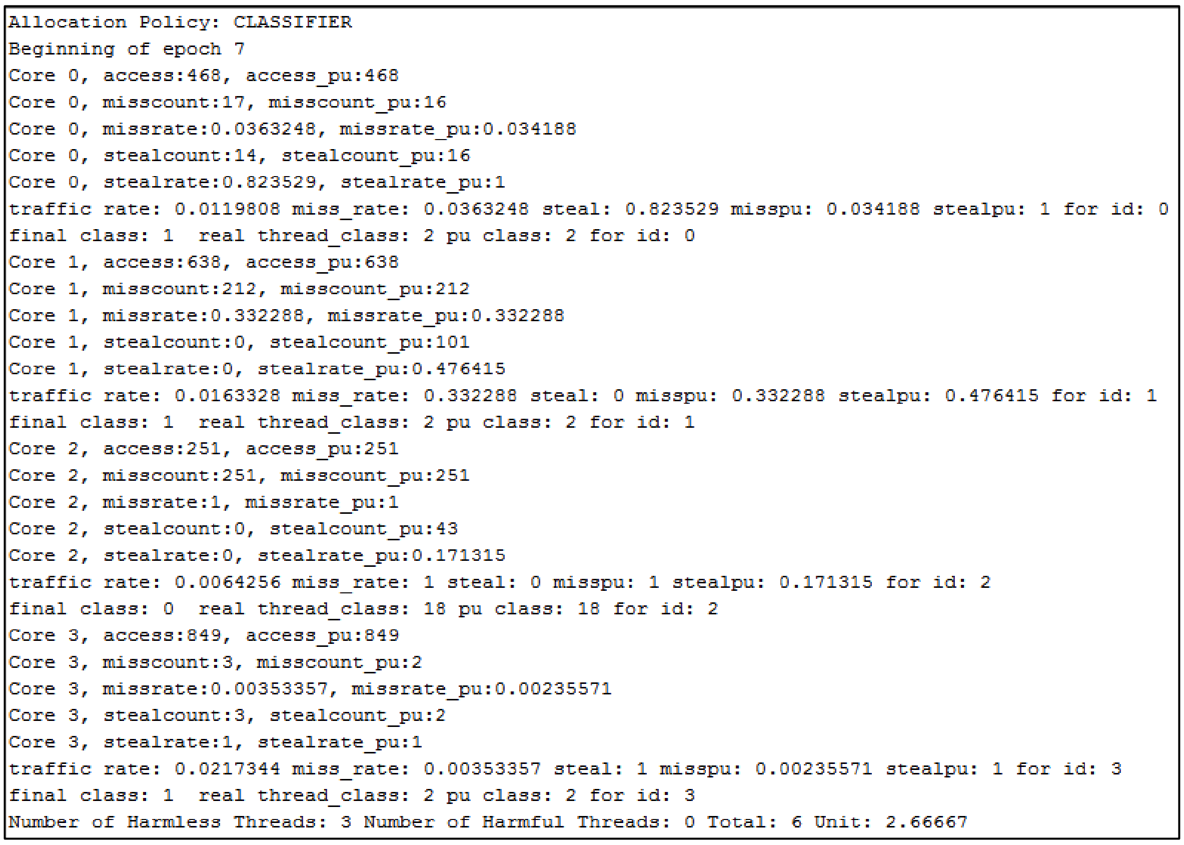
Finally for the Classifier, we implemented our PUTD structures which is similar to ATDs in UCP. We use same replacement policy of UCP, however, our study Classifier distinguishes from UCP with the allocation algorithm. As a result, in Figure 5.9, we implement our thread classification mechanism and allocation algorithm in Algorithm 3.1. In later phase of our study, we also implement adaptive traffic threshold mechanism and Special thread described in sections 3.3 and 3.4 respectively. Our classifer algorithm run at the end of every epoch as in other studies.



```
void classifier_allocate(macsim_c * m_simBase) {

    if (epoch_id == 0)
        return;

    double Unit = 0.0;
    int L = 0;
    int S = 0;
    int HF = 0;

    NH = 0;
    NL = 0;
    NS = 0;
    /** Weight of Harmless Thread*/
    int WL = 2;
    /** Weight of Special Thread*/
    int WS = 4;
    classify(m_simBase);

    // TODO yeni fonksiyon
    //classifier_table.get_result(allocations, classes, NUM_CORES);
    int Total = NS * WS + NL * WL + NH;
    if (Total != 0) {
        Unit = (double) NUM_WAYS / Total;
        L = ceil(Unit * WL);
        S = ceil(Unit * WS);
        HF = ceil(Unit);
    }

    cout << "Number of Harmless Threads: " << NL
            << " Number of Harmful Threads: " << NH << " Total: " << Total
            << " Unit: " << Unit << endl;

    cout << "After Overlap:" << endl;

    bool is_all_zero = true;
    for (int i = 0; i < NUM_CORES; i++) {
        if (core_class[i] != 0) {
            is_all_zero = false;
            break;
        }
    }

    if (is_all_zero) {
        for (int i = 0; i < NUM_CORES; i++) {
            allocations[i] = 4;
        }
    } else {
        for (int i = 0; i < NUM_CORES; i++) {
            // Special thread
            if (Total != 0 && core_class[i] == 4) {
                allocations[i] = S;
            } else if (Total != 0 && core_class[i] == 1) {
                allocations[i] = L;
            } else if (Total != 0 && core_class[i] == 2) {
                allocations[i] = HF;
            } else {
                allocations[i] = 0;
            }
        }
    }

    for (int i = 0; i < NUM_CORES; i++) {
        cout << i << " alloc:" << allocations[i] << endl;
    }

    //classifier_final_tweak();
    apply_allocations_to_mock_caches();

    //classifier_copy_llc_pt(m_simBase);
    //classifier_copy_pt_pu();

    classifier_copy_pipp_pt(m_simBase);
    classifier_copy_pt_pu(m_simBase);
    reset_mock_cache_counters();
}
```

Figure 5.9. Implementation of Classifier allocation algorithm

For tracing purposes, we implement trace methods for every special structure of each individual algoritm. Parameters such as hit/miss counters, hit/miss rates, classifier hit/miss/steal rates (*tr*, *mr*, *sr*, *mrpu*, *srpu*), *ThreadClass* values etc. At the end of each epoch we print those parameters as statistics. Those trace methods helped us to debug our macsim implementation codes many times. Figure 5.10 shows example output of those trace methods.

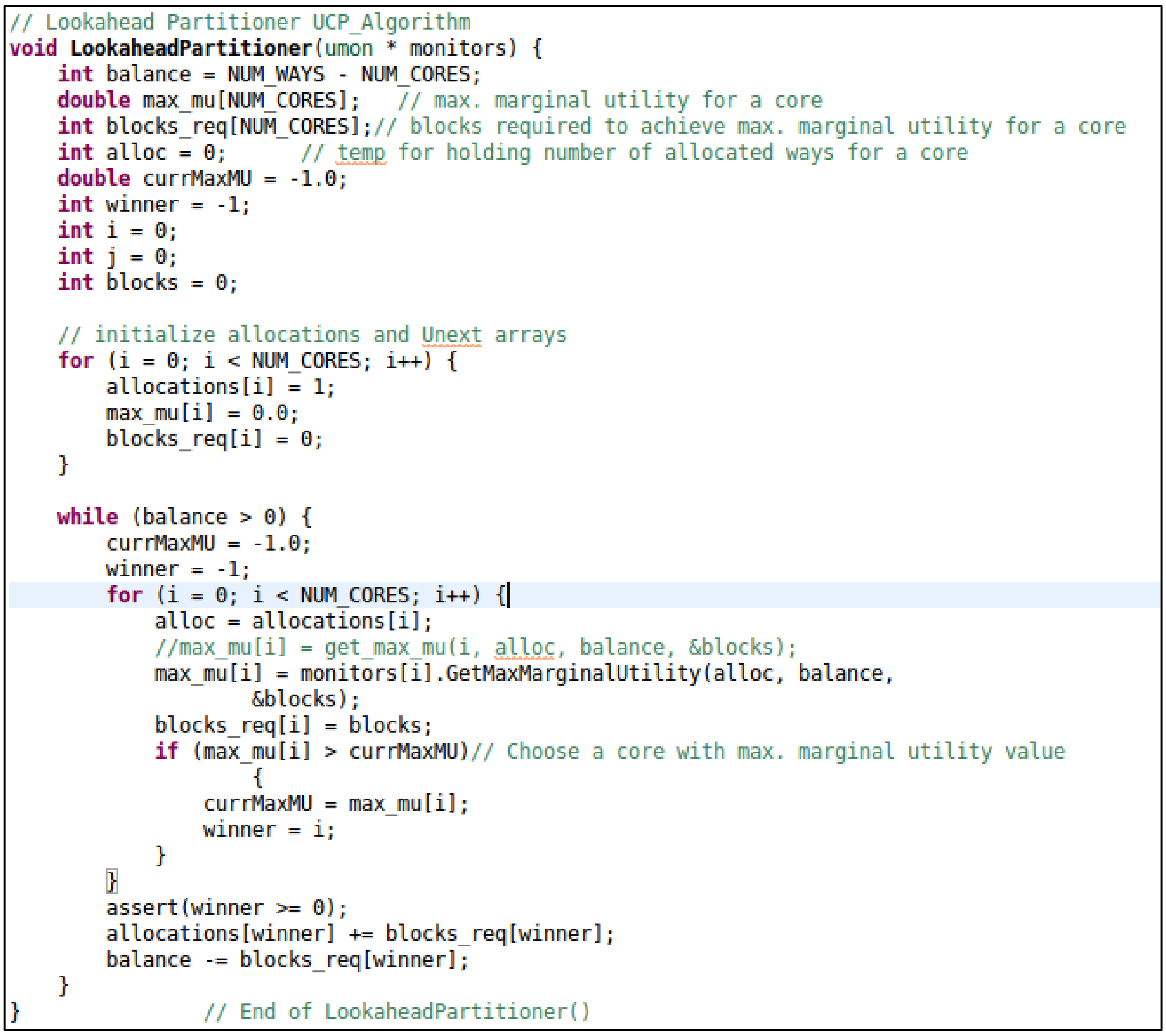


```
Allocation Policy: CLASSIFIER
Beginning of epoch 7
Core 0, access:468, access_pu:468
Core 0, misscount:17, misscount_pu:16
Core 0, missrate:0.0363248, missrate_pu:0.034188
Core 0, stealcount:14, stealcount_pu:16
Core 0, stealrate:0.823529, stealrate_pu:1
traffic rate: 0.0119808 miss_rate: 0.0363248 steal: 0.823529 misspu: 0.034188 stealpu: 1 for id: 0
final class: 1  real thread_class: 2 pu class: 2 for id: 0
Core 1, access:638, access_pu:638
Core 1, misscount:212, misscount_pu:212
Core 1, missrate:0.332288, missrate_pu:0.332288
Core 1, stealcount:0, stealcount_pu:101
Core 1, stealrate:0, stealrate_pu:0.476415
traffic rate: 0.0163328 miss_rate: 0.332288 steal: 0 misspu: 0.332288 stealpu: 0.476415 for id: 1
final class: 1  real thread_class: 2 pu class: 2 for id: 1
Core 2, access:251, access_pu:251
Core 2, misscount:251, misscount_pu:251
Core 2, missrate:1, missrate_pu:1
Core 2, stealcount:0, stealcount_pu:43
Core 2, stealrate:0, stealrate_pu:0.171315
traffic rate: 0.0064256 miss_rate: 1 steal: 0 misspu: 1 stealpu: 0.171315 for id: 2
final class: 0  real thread_class: 18 pu class: 18 for id: 2
Core 3, access:849, access_pu:849
Core 3, misscount:3, misscount_pu:2
Core 3, missrate:0.00353357, missrate_pu:0.00235571
Core 3, stealcount:3, stealcount_pu:2
Core 3, stealrate:1, stealrate_pu:1
traffic rate: 0.0217344 miss_rate: 0.00353357 steal: 1 misspu: 0.00235571 stealpu: 1 for id: 3
final class: 1  real thread_class: 2 pu class: 2 for id: 3
Number of Harmless Threads: 3 Number of Harmful Threads: 0 Total: 6 Unit: 2.66667
```

Figure 5.10. Example output statistics of trace method implementation

6. EXPERIMENTAL METHODOLOGY

We evaluate our proposed mechanism using trace-driven simulations on Macsim [17]. Specifically, for the baseline configuration, we faithfully implement the UCP UMON mechanism, which integrates the Lookahead cache allocation policy. In Figure 6.1, our implementation of Lookahead algorithm can be observed.



```
// Lookahead Partitioner UCP_Algorithm
void LookaheadPartitioner(umon * monitors) {
    int balance = NUM_WAYS - NUM_CORES;
    double max_mu[NUM_CORES];   // max. marginal utility for a core
    int blocks_req[NUM_CORES];// blocks required to achieve max. marginal utility for a core
    int alloc = 0;      // temp for holding number of allocated ways for a core
    double currMaxMU = -1.0;
    int winner = -1;
    int i = 0;
    int j = 0;
    int blocks = 0;

    // initialize allocations and Unext arrays
    for (i = 0; i < NUM_CORES; i++) {
        allocations[i] = 1;
        max_mu[i] = 0.0;
        blocks_req[i] = 0;
    }

    while (balance > 0) {
        currMaxMU = -1.0;
        winner = -1;
        for (i = 0; i < NUM_CORES; i++) {
            alloc = allocations[i];
            //max_mu[i] = get_max_mu(i, alloc, balance, &blocks);
            max_mu[i] = monitors[i].GetMaxMarginalUtility(alloc, balance,
                    &blocks);
            blocks_req[i] = blocks;
            if (max_mu[i] > currMaxMU)// Choose a core with max. marginal utility value
                    {
                currMaxMU = max_mu[i];
                winner = i;
            }
        }
        assert(winner >= 0);
        allocations[winner] += blocks_req[winner];
        balance -= blocks_req[winner];
    }
}               // End of LookaheadPartitioner()
```

Figure 6.1. Macsim implementation of Lookahead algorithm

Then, we replace the Lookahead policy with our proposed policy. We compare the results in terms of throughput and fairness metrics. The details of the simulation parameters are shown in Table 6.1. Simulations are fast forwarded until SimPoints [18] are reached and executed for up to 1 Billion instructions.

Table 6.1. 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-cache and D-cache, 4-way, 64B line size, 3 cycle hit latency
L2 Cache	2, 4 or 8MB, shared, 20 to 40 cycles hit latency
DSS sets	32

For workload generation, we chose 19 SPEC2006 CPU benchmarks after eliminating benchmarks with similar utility curves and cliff locations. The list of benchmarks is given in Table 6.2. We randomly generated 100 4-thread, 50 8-thread and 25 16-thread workloads that require 16, 32 and 64 LLC ways, respectively. Then we extend the set of simulated workloads in first phase of our study from 50 to 100 workloads.

Table 6.2. 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

7. TESTS AND RESULTS

Threshold values for traffic rate, miss rate, and steal rate are determined by a sensitivity study. Traffic rate thresholds are tested in a fine-grain (step size of 0.005) brute force manner in the first phase of our study but adaptive traffic threshold values are used in the second phase. Miss rate and steal rate thresholds fixed at three different preset points: low (0.25, 0.50), medium (0.50, 0.75) and high (0.75, 0.95). Next, by fixing the traffic rate thresholds at their optimal settings, same sensitivity study was applied on miss rate followed by steal rate thresholds. The results show that the performance of the classifier is not very sensitive to threshold changes around peak threshold values.

Performance and fairness results presented in sections 7.1 and 7.2 are obtained from validation workloads, which are different than training workloads used in sensitivity studies, and are run for longer simulation periods. The threshold values obtained by our sensitivity studies are shown in Table 6.1.

In Figures 7.1 and 7.2, workloads are sorted in ascending order according to their gains, separately for each legend. The workloads on the x-axis are different for each LLC configuration. Also note that the index of a workload in two graphs are not necessarily same.

7.1. PERFORMANCE

Figure 7.1 shows the percentage of IPC gains compared to a non-partitioned baseline LRU scheme for both Lookahead and Classifier mechanisms. Compared to Lookahead mechanism, our Classifier-based allocation mechanism gives better or equal performance in 49 (out of 50) workloads that we studied. While the average performance improvement reaches to 3 per cent, peak performance improvement is 8.5 per cent in workload 25, which contains bzip2, libquantum, gamess and GemsFDTD benchmarks. Another important fact, which we notice in the same graph, is that Lookahead fails to beat baseline LRU scheme in more than 20 workloads, while Classifier does not lose a single round. As a result, this graph emphasizes that our Classifier-base allocation mechanism has a significant advantage over the Lookahead mechanism. We strongly believe that this advantage comes from the better distribution of resources among threads with the use of our thread classification scheme.

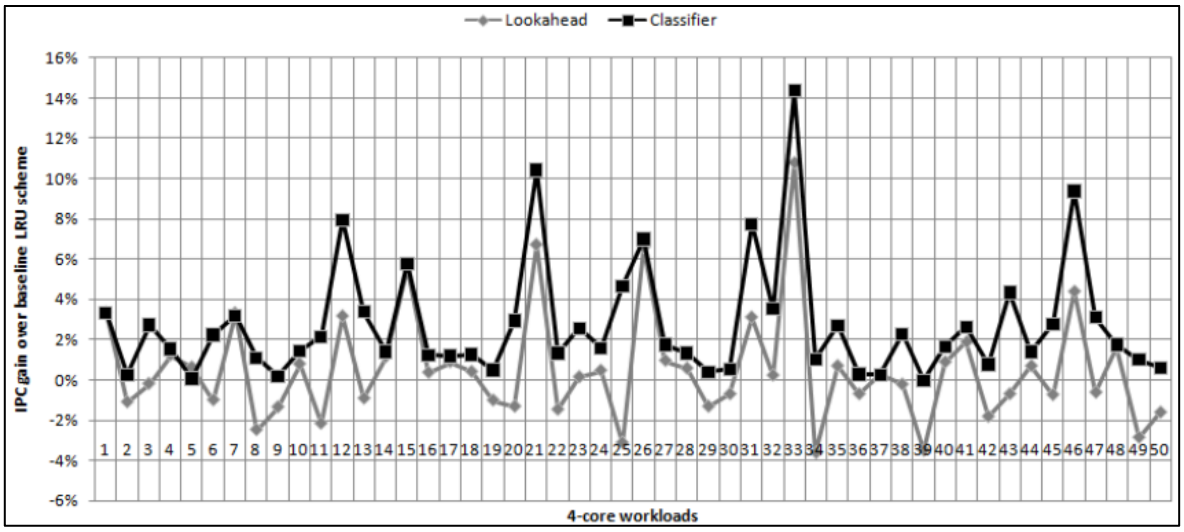


Figure 7.1. Speedup of the Classifier and the Lookahead over LRU in 4-core/16-way LLC configuration

Figures 7.2 and 7.3 show similar trends in performance for 8- and 16-core processor configurations. For the 8-core processor configuration, peak IPC gain over Lookahead is nearly 5 per cent in workload 9, which contains gamess, namd, bwaves, gromacs, omnetpp, GemsFDTD, mcf and xalanc benchmark mixture. For the 16-core processor configuration, we also ran Vantage mechanism with 64-way LLC cache and compared the effectiveness of both Lookahead and Classifier side-by-side. As a result, Figure 7.3 shows both UCP and Vantage results in one graph. Our Classifier-based mechanism performs better than Lookahead in all 25 workloads with the UCP enforcement policy. When we consider the Vantage enforcement mechanism, this number reduces to 21 workloads. Classifier's peak performance gain over Lookahead is again nearly 5 per cent in workload 24. Remembering that the hardware complexity of our Classifier is much simpler compared to the Lookahead mechanism, overall IPC gain shows that our Classifier-based allocation mechanism is also very suitable to CMPs with a large number of cores.

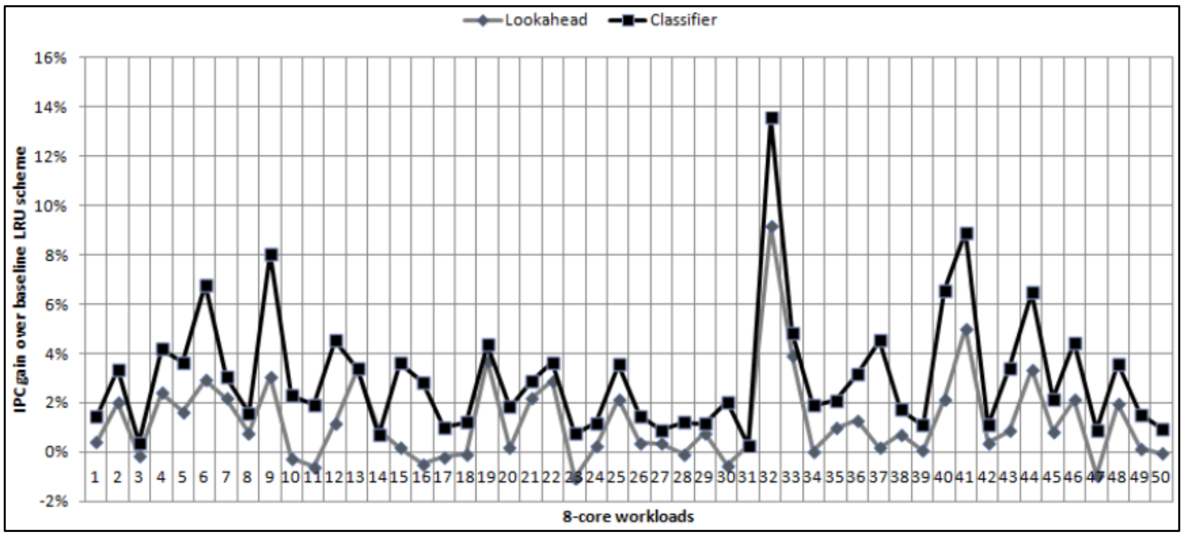


Figure 7.2. Speedup of the Classifier and the Lookahead over LRU in 8-core/32-way LLC configuration

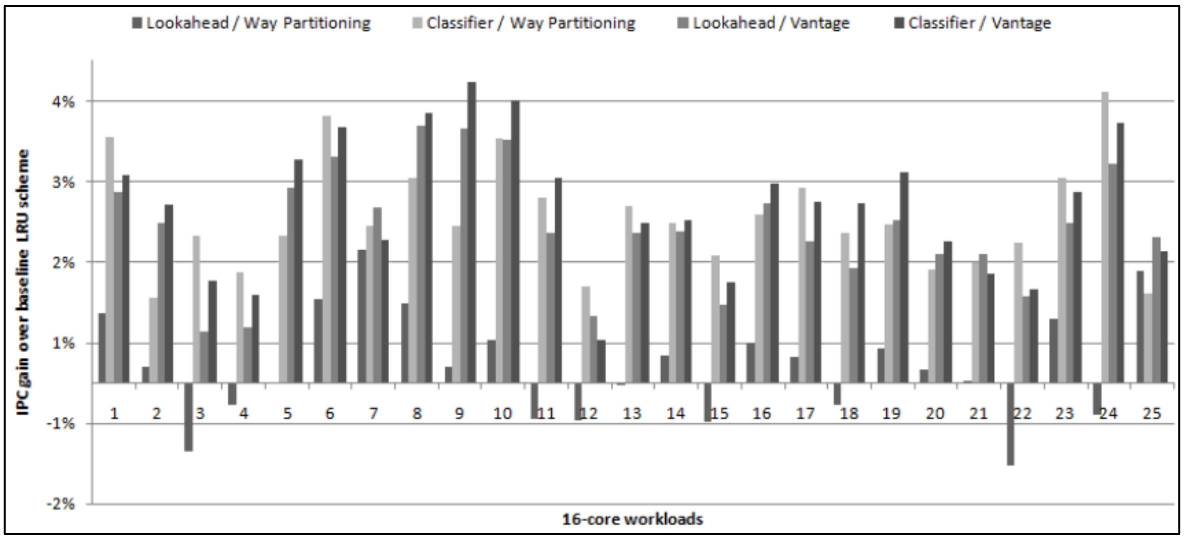


Figure 7.3. Speedup of the Classifier and the Lookahead over LRU in 16-core/64-way LLC configuration

7.2. FAIRNESS

We also evaluate the fairness of our mechanism in terms of weighted speedup and harmonic mean metrics as shown in Equations 7.1 and 7.2. A number of fairness metrics including harmonic mean have been discussed by Vandierendonck and Seznec in [19]. Figure 7.4 shows that, for the 4-core processor configuration, these metrics gives almost identical results in all simulated workloads. Moreover, we also show that the fairness of our Classifier is better than the fairness of Lookahead in 46 (out of 50) workloads. While the average fairness improvement reaches to 2 per cent, peak performance improvement is around 6 per cent in workload 50, which contains zeusmp, milc, povray and mcf benchmarks. As a result, we can safely say that our Classifier-based allocation mechanism not only increases the performance of workloads but also improves the level of fairness compared to the Lookahead mechanism. Here, the important point is that *Null* or *Very Harmful* threads do not lose performance since they do not utilize LLC, meanwhile other threads can make good use of extra cache ways. Notice that only 4 workloads have slightly negative weighted speedup and harmonic mean but their IPC gains are still positive. We believe that in these four workloads *Very Harmful* classification might have been failed. As we already discussed in Section 3.2, the classification of threads are done by the help of some empirically-chosen thresholds. In a later study, we would like to study these thresholds in further detail. We also would like to emphasize the possibility of making these thresholds adaptive to the behaviors of the running threads.

$$Weighted\ Speedup = \sum_{i=1}^{N} \frac{IPC_i^{Classifier}}{IPC_i^{Lookahead}} \quad (7.1)$$

$$Harmonic\ Mean = N \Big/ \sum_{i=1}^{N} \frac{IPC_i^{Classifier}}{IPC_i^{Lookahead}} \quad (7.2)$$

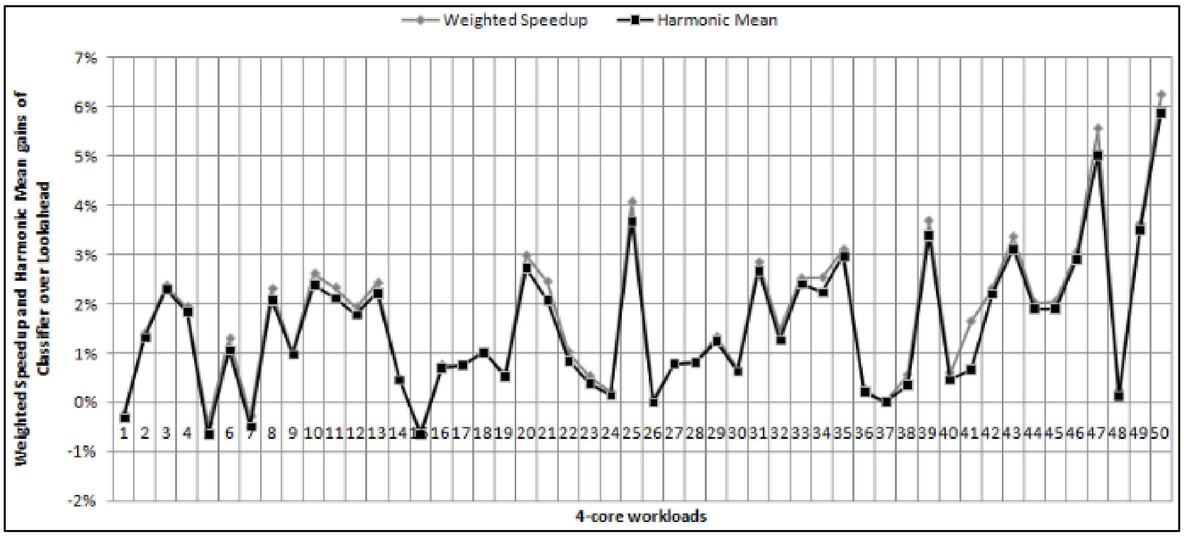


Figure 7.4. Weighted speedup and harmonic mean gain of Classifier over Lookahead in 4-core/16-way LLC configuration

For the 8-core runs, we observe similar results, which are shown in Figure 7.5. Here, the fairness of our Classifier is better than the fairness of Lookahead in 43 (out of 50) workloads. While the average fairness improvement reaches to 2 per cent, peak performance improvement is more than 5 per cent in workload 42, which contains omnetpp, soplex, sphinx3, bwaves, href264, leslie3d, namd and milc benchmarks. Also note that there is a large discrepancy between the weighted speedup and harmonic mean results of workload 11, which contains milc, gamess, povray, bwaves, soplex, xalanc, mcf and sjeng benchmarks. This problem is due to misjudgment of a thread class resulting in huge performance drop in one of the benchmarks. In such a case, the overall performance may still be high (see workload 11 in Figure 7.2), but the harmonic mean will get the maximum penalty, indicating that the resources are not fairly distributed among threads. In this specific case, the problematic benchmark is mcf. Although, it is sufficient to allocate one cache way to satisfy this benchmark, the Classifier fails to classify it as a *Harmless* class but a *Null* class giving it none of the cache ways. Although, the performance of mcf benchmark is quite low, after this allocation decision, its IPC is nearly halved resulting in a huge harmonic mean drop.

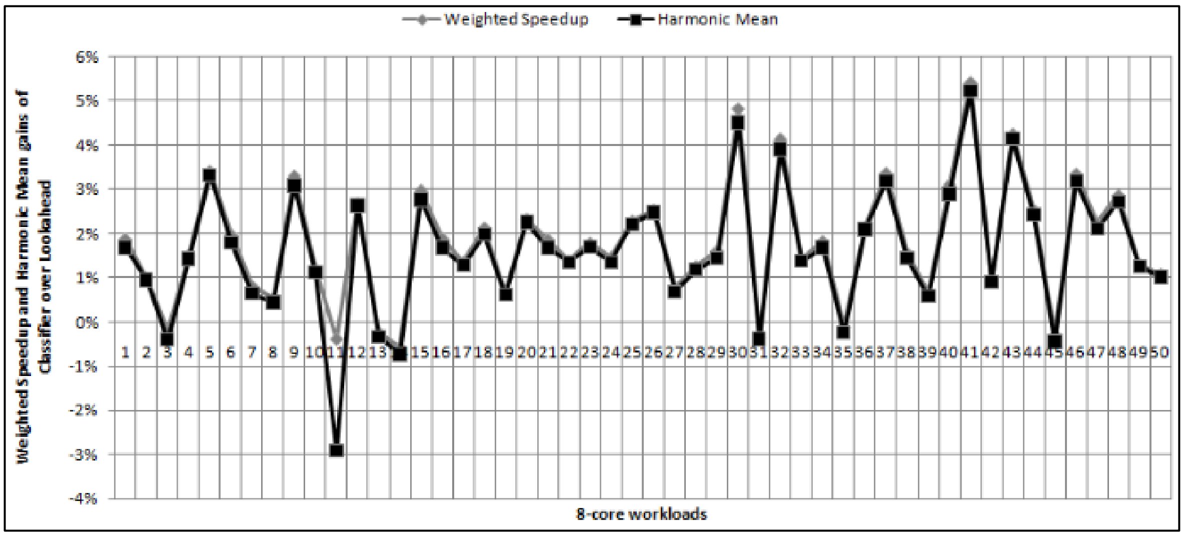


Figure 7.5. Weighted speedup and harmonic mean gain of Classifier over Lookahead in 8-core/32-way LLC configuration

Finally, for the 16-core runs, we again combined the UCP and the Vantage results in a single graph, which is shown in Figure 7.6. Fairness results for the UCP runs show that the Classifier is the sole winner in all workloads when the weighted speedup metric is considered. Moreover, When harmonic mean metric is considered, Classifier still gives better fairness results in 23 (out of 25) workloads. However, although performance results for our Classifier in Vantage mechanism is almost always better than the Lookahead mechanism, the fairness graphs shows that Classifier performs very poorly in 7 workloads. This is the first time that we encounter such a disparity. This might be the result of the ODFA strategy, which we choose for the integration process. In a later study, we are planning to investigate a TMD-based integration, since we believe that it would create a more natural binding between the allocation and the enforcement mechanisms.

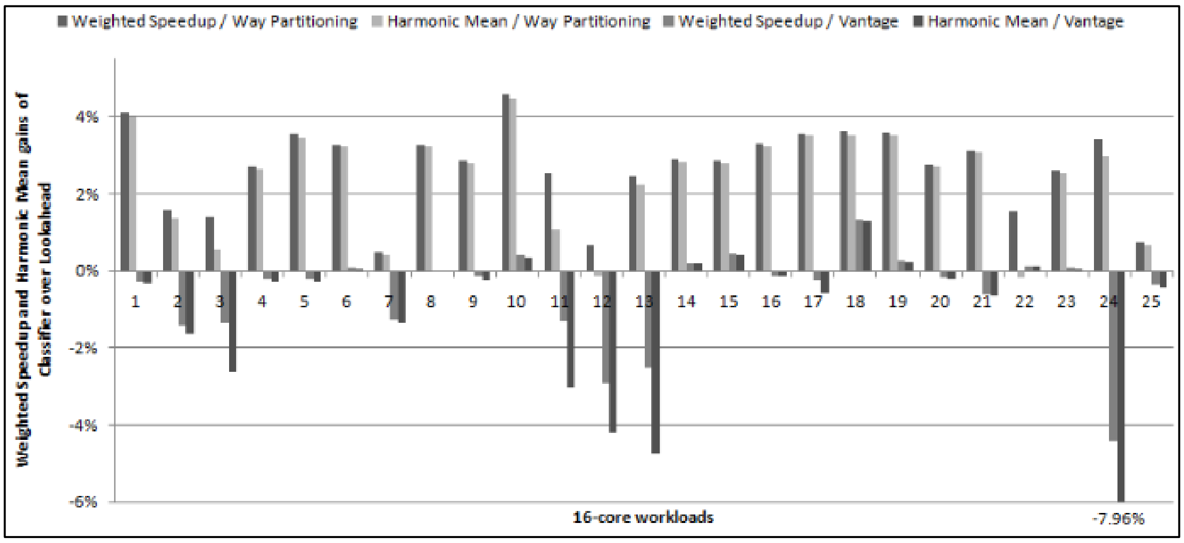


Figure 7.6. Weighted speedup and harmonic mean gain of Classifier over Lookahead in 16-core/64-way LLC configuration

7.3. SENSITIVITY TO ADAPTIVE TRAFFIC THRESHOLD MECHANISM

In section 3.3, we describe the requirements of the adaptive threshold mechanism to achieve better scalability to different sizes of the L1 cache. Here, we discuss the results of the simulations. We run the simulations with 4-cores and 4-sets of L1 size (32, 64, 128 and 256 sets). We compare the baseline LRU algorithm, the base classifier and, finally, the adaptive classifier, which we propose in this work. Figure 7.7 shows the comparison of these results.

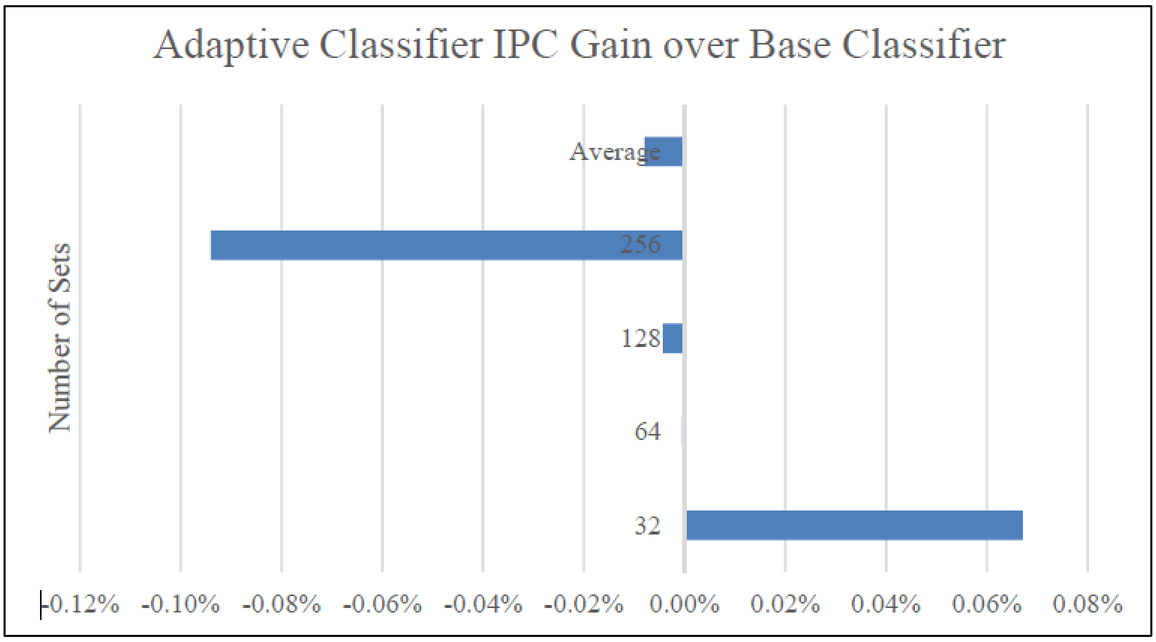


Figure 7.7. Adaptive Classifier IPC gain over base Classifier

For 32 sets L1, it performs slightly better 0.07 per cent, 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 per cent and for 256 sets of L1 cache, it performs slightly worse again; -0.09 per cent On the average, adaptive classifier results 0.01 per cent decrease in performance. Results show that our adaptive approach gives similar performance as compared to the base classifier. This proves that the adaptive mechanism can easily be implemented upon the base classifier since it does not require any predefined traffic threshold and can adapt to any sizes of L1 cache.

7.4. SPECIAL THREAD

In section 3.4, we describe the requirement of an additional class of threads which we call *Special* Thread. Implementation of this additional classification gives better performance results when *deal* and *astar* benchmarks are taken into consideration. In the first phase of our work, we choose randomly 50 WL for 4 core simulations but for the Special thread study, we add extra 50 WLs that include *deal* and *astar*. In Figure 7.8, for 100 WLs, base classifier algorithm gives 1.24 per cent worse performance as compared to Lookahead algorithm.

However, our special thread implementation gives 0.75 per cent worse performance as compared to 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 includes *deal* and *astar*, improvement is 1.61 per cent.

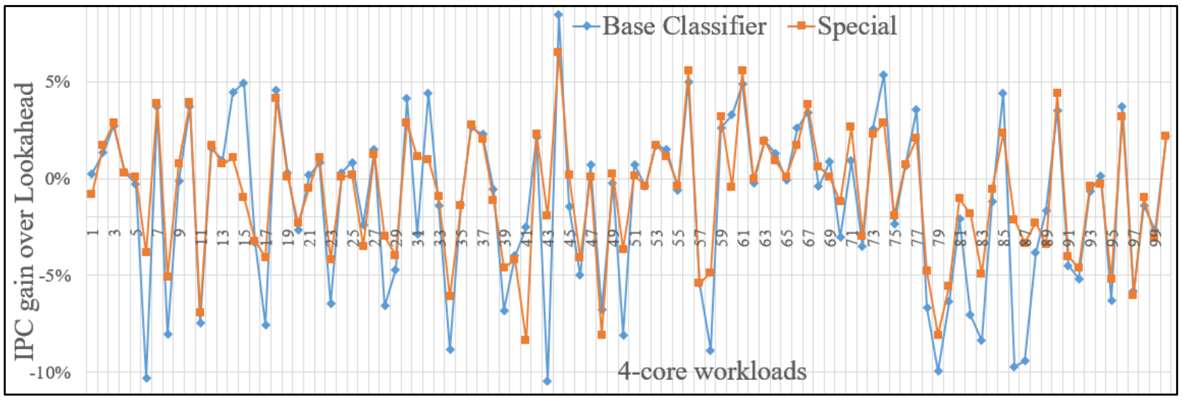


Figure 7.8. IPC gain of Classifier with *Special* thread mechanism over Lookahead

In WL43 nearly 10 per cent improvement, in WL86 more than 8 per cent improvement and in WL87 nearly 7 per cent improvement is observed. All of those three WLs include *deal* benchmark. On the other hand, WL14 and WL15 shows decrease in performance (3 per cent and 5 per cent drop, respectively), since those WLs do not include *deal* or *astar* benchmarks. On the average, Special Thread implementation gives better performance and makes the base classifier more robust against all kind of threads.

7.5. 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 PIPP-based replacement policy instead of the LRU-based policy that ODFA implements. In Figure 7.9, results show that performance is insensitive whether the implementation is ODFA or TMD for PIPP. Another outcome of these results is the 6 per cent average increased performance of ODFA or TMD is originated from the success of classifier in classifying and measuring the applications; not from the fitness of ODFA or TMD for PIPP.

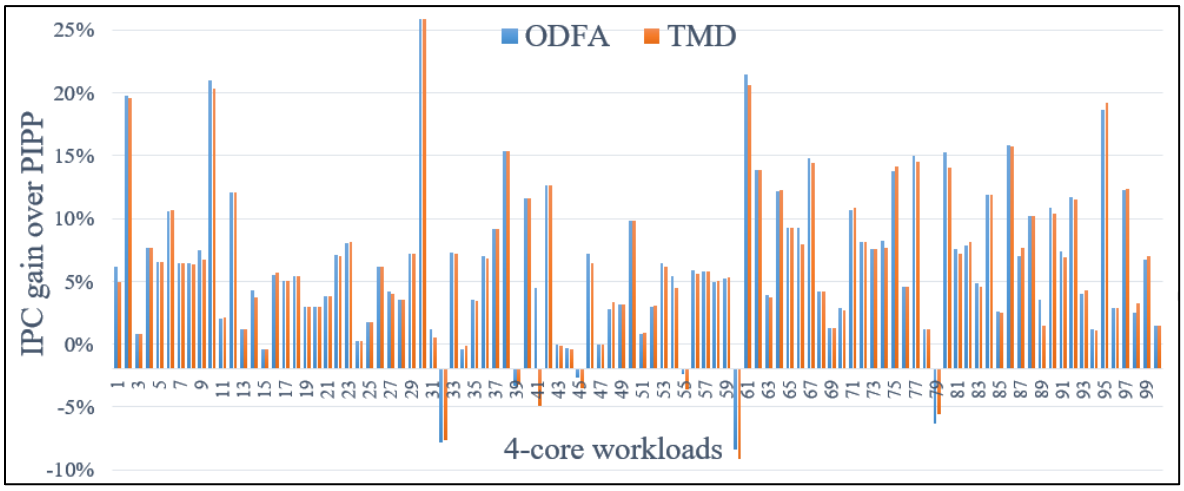


Figure 7.9. IPC gain of ODFA and TMD Classifier over baseline PIPP

8. CONCLUSION

Contemporary processors utilize a Last Level Cache (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 Utility-based Cache Partitioning (UCP) is one of the 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 structures named Auxiliary Tag Directory (ATD) 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 Parallel Universe Tag Directories (PUTDs), which are structurally identical to ATDs. However, the function of these structures is to create parallel execution dimensions, which we call 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 show that our Classification-based allocation mechanism requires only a fraction of operations (less than 1 per cent) compared to the Lookahead allocation mechanism, and this makes it highly scalable and suitable for CMPs with a large number of cores. We also evaluate the proposed mechanism in terms of performance and fairness metrics in 4-core, 8-core and 16-core configurations with 16-way, 32-way and 64-way set associative cache organizations, respectively. The results show that the mechanism performs consistently better than Lookahead in all of the studied configurations. The IPC gain over Lookahead can be as high as 8.5 per cent whereas fairness improvement can be as high as 6 per cent.

We also discuss the possibility of two design alternatives for integration with other partitioning enforcement policies. The One-Design-Fits-All (ODFA) strategy, which we focus in this study, is a generic LRU based implementation of the mechanism that may not be suitable for enforcement policies with different cache organizations and replacement policies. As a result, our ODFA-based Classifier gives better results when LRU-based UCP enforcement policy is utilized. However, its integration with Vantage enforcement mechanism gives somewhat lower gains. The IPC gain of the Classifier is still higher than the IPC gain of the Lookahead in all the workloads that we studied. But, the fairness results are notably worse than the Lookahead in 7 workloads of the 16-core configuration.

REFERENCES

1. K. Qureshi, and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. *Proceedings of the 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Orlando, 423-432, 2006.

2. Y. Xie, and G. H. Loh. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. *Proceedings of the 2009 36th Annual International Symposium on Computer Architecture*, Austin, 174-183, 2009.

3. D. Sanchez, and C. Kozyrakis. Scalable and Efficient Fine-Grained Cache Partitioning with Vantage. *IEEE Micro*, 3:26-37, 2012.

4. I. A. Guney, A. Yildiz, I. U. Bayindir, K. C. Serdaroglu, U. Bayik, and G. Kucuk. A Machine Learning Approach for a Scalable, Energy-Efficient Utility-Based Cache Partitioning. *ISC High Performance Computing*, Frankfurt, 409-421, 2015

5. M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive Insertion Policies for High Performance Caching. *Proceedings of the 2007 34th Annual International Symposium on Computer Architecture*, San Diego, 381-391, 2007.

6. B. S. Ovant, I. A. Guney, M. E. Savas, and G. Kucuk. Allocation of Last Level Cache Partitions through Thread Classification with Parallel Universes. *Proceedings of the 2016 14th International Conference on High Performance Computing and Simulation*, Innsbruck, 204-212, 2016.

7. A. Jaleel, W. Hasenplaugh, M. K. Qureshi, J. Sebot, S. C. Steely, and J. Emer. Adaptive Insertion Policies for Managing Shared Caches. *Proceedings of the 2008 17th International Conference on Parallel Architectures and Compilation Techniques*, Toronto, 208-219, 2008.

8. N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum. Improving Cache Management Policies Using Dynamic Reuse Distances. *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Vancouver, 389-400, 2012.

9. B. Nathan, and S. Daniel. Talus: A Simple Way to Remove Cliffs in Cache Performance. *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture*, San Francisco, 2015.

10. R. Manikantan, K. Rajan, and R. Govindarajan. Probabilistic Shared Cache Management (PriSM). *Proceedings of the 2012 39th Annual International Symposium on Computer Architecture*, Portland, 428-439, 2012.

11. L. Li, J. Lu, and X. Cheng. Block Value Based Insertion Policy for High Performance Last-Level Caches. *Proceedings of the 2014 28th ACM International Conference on Supercomputing*, Munich, 63-72, 2014.

12. R. Wang, and L. Chen. Futility Scaling: High-Associativity Cache Partitioning. *Proceedings of the 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Cambridge, 356-367, 2014.

13. X. Wang, and J. F. Martinez. Xchange: A Market-Based Approach to Scalable Dynamic Multi-Resource Allocation in Multicore Architectures. *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture,* San Francisco, 113-125, 2015.

14. A. Fog. Instruction Tables Lists of Instruction Latencies, Throughputs and Micro Operation Breakdowns for Intel, AMD and VIA CPUs, http://www.agner.org/optimize/instruction_tables.pdf [retrieved 20 January 2016].

15. D. Sanchez, and C. Kozyrakis. The Zcache: Decoupling Ways and Associativity. *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Atlanta, 187-198, 2010.

16. Macsim Manual. CompArch, http://comparch.gatech.edu/hparch/macsim/macsim.pdf [retrieved 1 October 2013].

17. Macsim Simulator, http://code.google.com/p/macsim/ [retrieved 1 October 2013].

18. T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. *Proceedings of the 2001 6th International Conference on Parallel Architectures and Compilation Techniques,* Novosibirsk, 2001.

19. H. Vandierendonck, and A. Seznec. Fairness Metrics for Multi-Threaded Processors, *IEEE Computer Architecture Letters*, 1:4-7, 2011.