# An Algorithm to Enhance Cache Efficiency in Multi-core Processors

Majid Babaei[1], Ali Ghaffari[2*]
1, 2 – Department of Computer Engineering, Tabriz branch, Islamic Azad University, Tabriz, Iran
E-mail: A.ghaffari@iaut.ac.ir (Corresponding author)

Received: August 2016 Revised: January 2017 Accepted: April 2017

**ABSTRACT**
The Cache efficiency is considered to be one of the major challenges in multi-core processors. Hence, using cache space in such processors should be meticulously managed by each of the cores. This paper addresses the issue of cache re-access and proposes an algorithm which divides the last level of cache into local and global share for the cores. The rationale behind the proposed algorithm is to activate or deactivate the ways of cache for any intended core. Consequently, the collision between cores is reduced and each of the cores can use the cache space dynamically. To simulate the proposed algorithm, the researchers used three groups of applications and the obtained results were examined and evaluated in two stages. The first phase is involved with the number of active ways in cache for each core. It should be highlighted that the proposed algorithm, in the merged state, was able to enhance the active ways up to 19%. In the second phase, cache miss rate was taken into consideration and it was observed that about 7% improvement was achieved in this stage.

**KEYWORDS:** Asymmetric Multicore-Processors, Processor Performance, Cache Performance, Shared and Dedicated Structure, Partitioning of the Last Level of Cache.

## 1. INTRODUCTION

The development in the design and production of semiconductors, integrated-circuits, computer architecture, etc. has enhanced the efficiency of multi-purpose processors [1]. Moore's Law which was introduced in 1965 indicates that the degree of integrating a transistor on a chip has been doubled within the last one and half years [2]. As a case in point, The Intel Company introduced the Itanium2 model in 2006 which included 1.72 billion transistors within just the size of 21.5 mm by 21.5mm [3], [4]. However, it should be noted that same company introduced the Pentium Pro Model in 1995 which included only 5.5 million transistors on a chip with the size of 17.3mm [5].

It should be pointed out that the more the number of processors, the faster the speed of the processors will be [6]. Multi-threading, security and virtualization are regarded as the factors and parameters which are of high significance for multi-core processors. The advantages of multi-core processors cannot be readily observed and a lot of studies and examinations on hardware products should be conducted. One of the important hardware products is the cache.

The demand for more powerful and fast processors is increasing day in day out. Nevertheless, the increase in the power and speed of processors requires significant changes and evolutions in the development and production of processors. The majority of processor manufacturing companies such as Intel, IBM and AMD produce multi-core processors rather than single-core processors [7]. In designing cache, the issue of data locality should be taken into consideration so as to enhance efficiency [8]. Moreover, another issue known as block should be considered; change in block size can alter the features of cache such as hit rate and miss rate. On a single core processor that running multiple applications, the operating system acts as a scheduler - switching contexts between the applications. This can require a complete dump of all processor registers and possibly the cache(s), which is costly in terms of completion time. It is obvious that lessening the frequency of context switching will increase the usable cycles of a processor. One way of achieving this is by creating more processors to distribute the load [6]. For example, a computer running two applications will not need to switch contexts if there are two processors working in parallel. This example is simplistic as operating systems often take control, running scheduling and other management tasks in the background.

The concept of symmetry can be understood as the creation of a number of processors and multiple cores

on a distinct chip. For the efficient utilization of multiple cores by the applications, either the programmer should split the application into simultaneous parts or the operating system should break the task into distinct parts; this operation can be conducted by means of the multi-threading feature [9].

Cache refers to a type of memory which is located between the processor registers and the main memory and it is regarded as an efficiency bottleneck. In the majority of common architectures, the optimization of cache leads to the improvement of the efficiency of the entire system. However, it should be noted that many of the methods used in cache are complicated. Studying and investigating cache memory in multi-core processors reveals that the shared or dedicated structure for the last level cache cannot enhance the efficiency by itself.

In this paper, the proposed algorithm was intended to dynamically segment and partition the last level cache into the cores in asymmetric multi-core processors. The proposed algorithm made use of both structures of local share and dedicated share since, at first, it considers the cache memory for each core individually; and then, after an application is started and executed at a certain time span, the algorithm can change the segmentation and partitioning of the last level cache. Accordingly, it can allocate more of the cache memory to the core which needs more cache. In as much as the cache memory should be specifically allotted to a core, the cores compete with each other to obtain resource and, as a result, the problem of collision might occur. Thus, the proposed algorithm is aimed at addressing this challenge. Thus the proposed algorithm can dynamically allocate more cache to one of the cores, if needed. In this study, the results are given in two sections: the number of the active ways and miss rates.

The reminder of the paper is organized as follows: Section 2 shows the related works. Then, in section 3, the concept of re-access to cache memory is discussed. In section 4, the cache partitioning algorithm is introduced and discussed. In section 5, the simulation results are reported and evaluated and the findings of the present study are compared with those of other studies.

## 2. RELATED WORKS

The majority of modern multi-core processors utilize a shared feature cache. The main architectural problem of this type of cache is the probability of collision and competition among the cores since several cores compete with each other in using a shared cache. Uncontrolled sharing of cache results in a condition in which the core including the cache ejects the other core while the main core has not used the content. In the related studies, many solutions have been proposed for this problem. These solutions are classified into hardware and software solutions.

A software mechanism has been proposed for the operating system in [10] which allows it to segment and partition the cache in a shared way based on the physical allocation of the pages. This mechanism is virtually controlled by the operating system so that the collision degree can be observed at any moment. This method improved the efficiency up to 17% and had no negative impact on the other applications which were executed. Furthermore, the overload of this mechanism is negligibly low.

A great number of researchers in this area have admitted the challenge of collision in cache memory and have suggested the hardware support for partitioning and segmenting the cache [11-13]. A number of the proposed hardware solutions have been effective and they were reasonably complex and their power consumption was appropriate. Nevertheless, they should be used in the processors of the next generation. It should be noted that these studies are evaluated by means of simulations and they have not been examined in real life situations.

Software-based methods were proposed in [14], [15]. The advantage of these methods is that they are executable on real systems. In these methods, cache memory is segmented and partitioned based on physical page. The focus in these methods is more on how to distribute data in non-uniform cache architecture (NUCA) so as to reduce the access delay.

In [16], cache memory was examined in multi-core processors since cache memory is regarded as a component which has a remarkable impact on the processor. For investigating cache, a number of criteria were used to clearly illustrate and indicate the effects of cache on the processor. At first, the increase in the number of cores was mentioned. Then, the major inputs of the cache memory were pointed out. The impact of the cache on processors was examined and shown by means of the variations in the inputs of cache. Dual-core Intel processor has a shared cache while the dual-core AMD processor has a second level dedicated cache. Figure 1 depicts an Intel quad-core processor with a second-level shared cache [16].

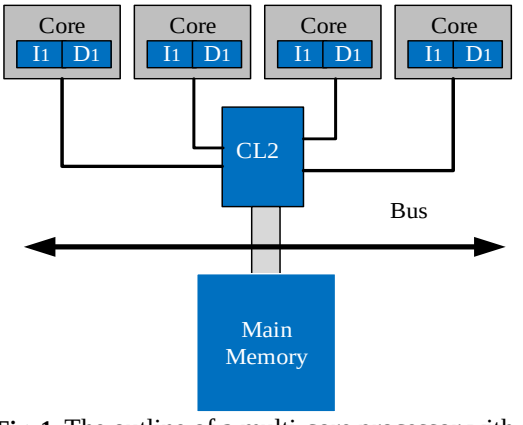


**Fig. 1.** The outline of a multi-core processor with a second-level shared cache in Intel architecture [16]

Also, figure 2 demonstrates an AMD quad-core processor with a second level dedicated cache; in this architecture, third level cache is shared.

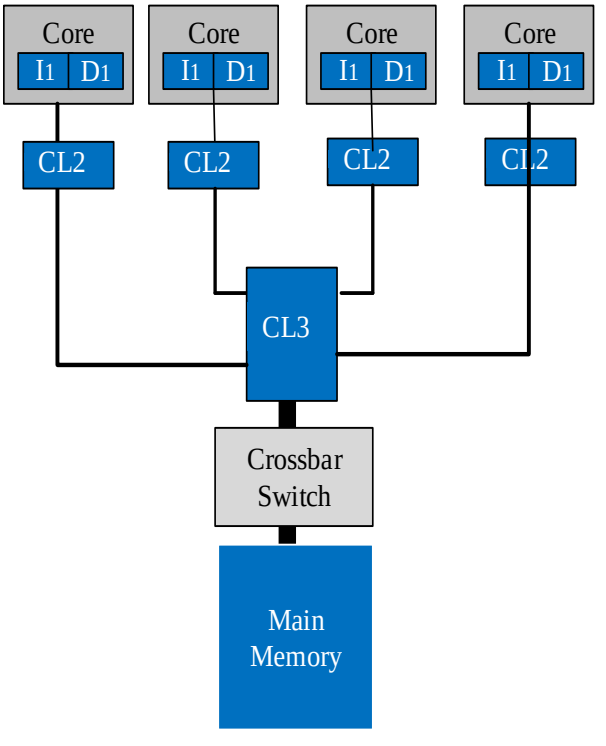


**Fig. 2.** Outline of a multi-core processor with a second level dedicated cache in AMD architecture [16]

A page replacement policy for personal computers was proposed in [17] which cache was divided into two ways. Data with more iterations and repetitions are placed in the main ways which is conducted based on the reuse distance. In contrast, data which are used much less than the main data are placed in the Deli way of the cache. In case there is a request for data removal from cache, the data from the Deli way are thrown out. Figure 3 illustrates these two ways of cache. The two bits of L and M were used in [17]. When bit M is set, it indicates that the line belongs to data with more repetitions and when L is set, it means that the line belongs to the data with less repetition. In conducting these experiments, non-uniform shared cache was used. The obtained results indicate that this replacement method has had 9.6%, 30% and 33% improvements in dual-core, quad-core and octal-core processors, respectively.

In [20], [13], cache was segmented based on the monitoring of the application which is executed. That is, at first, a memory is allocated to each core; however, this memory partitioning might dynamically change throughout the execution of applications. Nevertheless, a solution was proposed in [21] in which cache is segmented in a more interesting way before the execution of an application. In the present study, it was shown that as the size of the cache memory increases, the miss rate decreases.

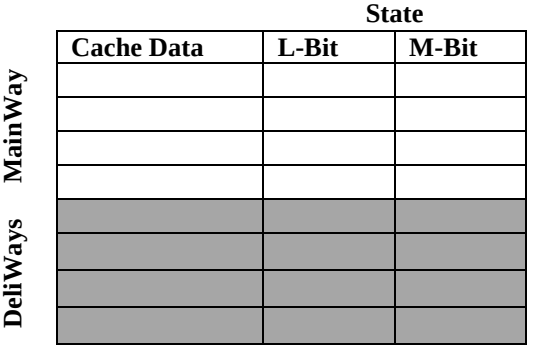


**Fig. 3.** Segmentation of the cache memory in the present study

A smart software-based method was proposed in [21] for sharing the last level cache in which the probability of collision among cores decreases. The rationale behind this method was to use the following factors in order to achieve the best condition in cache partitioning: figure of cache re-access, the frequency of cache access and cache miss rate.

A predictive method was considered in [22] for the cache memory. Hardware counter was used to allocate an appropriate size of cache for each core in the present study. In [23], a quasi-partitioning method was used for the last level cache which resulted in 10% efficiency improvement and 9% fairness improvement. This method effectively allocates the last level of cache by minimizing the destructive interferences of the applications which compete with each other for obtaining resources. Indeed, the allocation of cache is based on the features of applications which are to be executed such performance sensitivity to cache misses and thrashing. In this paper, specific counters and

hardware were used for collecting essential data such as cache miss rate or access rate and information of cache tags for each core.

Mixed-cell cache architecture was proposed in [29] for three levels of cache. In this architecture, for enhancing cache efficiency and reducing power consumption of the multi-core processors, the mixture of robust and non-robust cells were used in the cache memory. In all three levels of the cache, robust cells were used for the first and second ways. For the remaining ways, non-robust (standard) cells were used. In this architecture, if an error occurs in the first level of the cache, it will be treated as a miss rate which will have little impact on the efficiency. Nevertheless, if an error occurs in the second and third levels of the cache, this error must be corrected. In [29], the policies of reading and writing in the cache are different from each other. That is, the modified and altered data should be saved only in the robust ways. With respect to replacing in the case of miss rate in writing, only robust ways are used. Also, in the case of miss rate in reading, non-robust ways are used. The simulation results reported in [29] indicate a 17% improvement in efficiency. Figure 4 depicts this architecture.

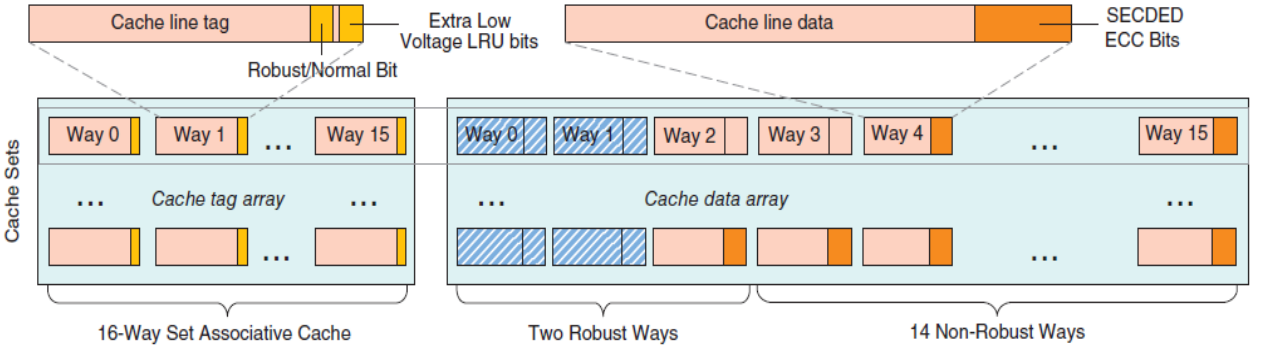


**Fig. 4.** The Mixed-cell architecture

The method of simulating a single-pass cache for two-level cache was proposed in [30]. This method operates based on stacks which maintains the address of cache blocks in a set of stacks. In [30], a hierarchy of dedicated cache memories was used for preventing the impacts of the inherent complexities of the rapid implementation of the applications. This method consists of a unified two-level cache. It uses the configurable first-level data instruction cache and the unified configurable second level cache. The output of this methodology is the number of data and instruction miss-rates in the first level and the number of write-backs and miss-rates in the second level for configuring cache to improve efficiency.

Studies reveal that the cache lock enhances the predictive capability in multi-core systems. The cache lock is intended to prevent the cache from overwriting a part of data or instruction. In [31], the technique of locking the ways of first level cache was used in multi-core systems for enhancing efficiency. Three techniques were introduced in [31]: random, static and dynamic techniques. In the random technique, a block is randomly selected for the lock; however, in the static and dynamic techniques, the selection of the block is based on a particular algorithm where the number of miss rate is taken into consideration.

Cache re-access is the concept that was used in [18, 19]. When a previously used page in the cache is reused with a certain distance of time, it is known as cache re-access. The data obtained from page reuse distance can be used to indicate the degree of the locality of application execution. The concept of cache re-access is used to define the following equation [18], [19]:

$$F = \frac{the\ number\ of\ the\ highest\ distance\ of\ reuse}{the\ number\ of\ the\ lowest\ distance\ of\ reuse} \quad (1)$$

In Eq. (1) in case F is a big value, it indicates that the executed application has little locality feature since the number of the highest distance of reuse is greater than the number of the lowest distance of reuse. In other words, it can be argued that if the value of F is low, it indicates that the executed application has a good locality feature. We will use this equation in our proposed algorithm.

## 3. THE PROPOSED ALGORITHM

The proposed algorithm was intended to activate or deactivate a number of the cache ways which are related to each of the cores. When the way of each core is deactivated, its possession is taken out of that core and if other cores need it, it will be given to them. In this way, cache can be dynamically allocated and divided among the cores.

In the proposed algorithm, each core executes the algorithm independently and makes the right decisions

based on the results. The activation or deactivation of ways is important because not using the ways is tantamount to wasting the accessible resources. Since a lot of cost has been paid for cache, it is expected that this memory should of remarkable importance and efficiency. Thus, it can be maintained that the activation of a large number of ways in the cache indicates an optimized utilization of the cache which can enhance efficiency. The flowchart of the proposed algorithm is depicted in figure 5. The proposed algorithm is defined as follows:

(1) Start of the processing task
(2) Sampling
(3) Calculating F
(4) Phase one: Comparing with the threshold
(5) Phase two: Preventing instantaneous changes
(6) Phase three: The proper selection of a page which should be extracted.
(7) Continued implementation of the application (go to 1)

After the application is started, in the sampling stage, the collected data from the re-access table was used to obtain the number of the highest reuse distance and the lowest reuse distance during running the program. In the next section, the data obtained in the sampling stage and the above mentioned equation is used to determine F value. The *F* value is of high significance for understanding the locality or non-locality of the application which is executed. Based on the results of the conducted experiments and simulations, the *F* value for indicating the locality of application was determined to be 0.02 and 0.005 (T1=0.005 and T2= 0.02).

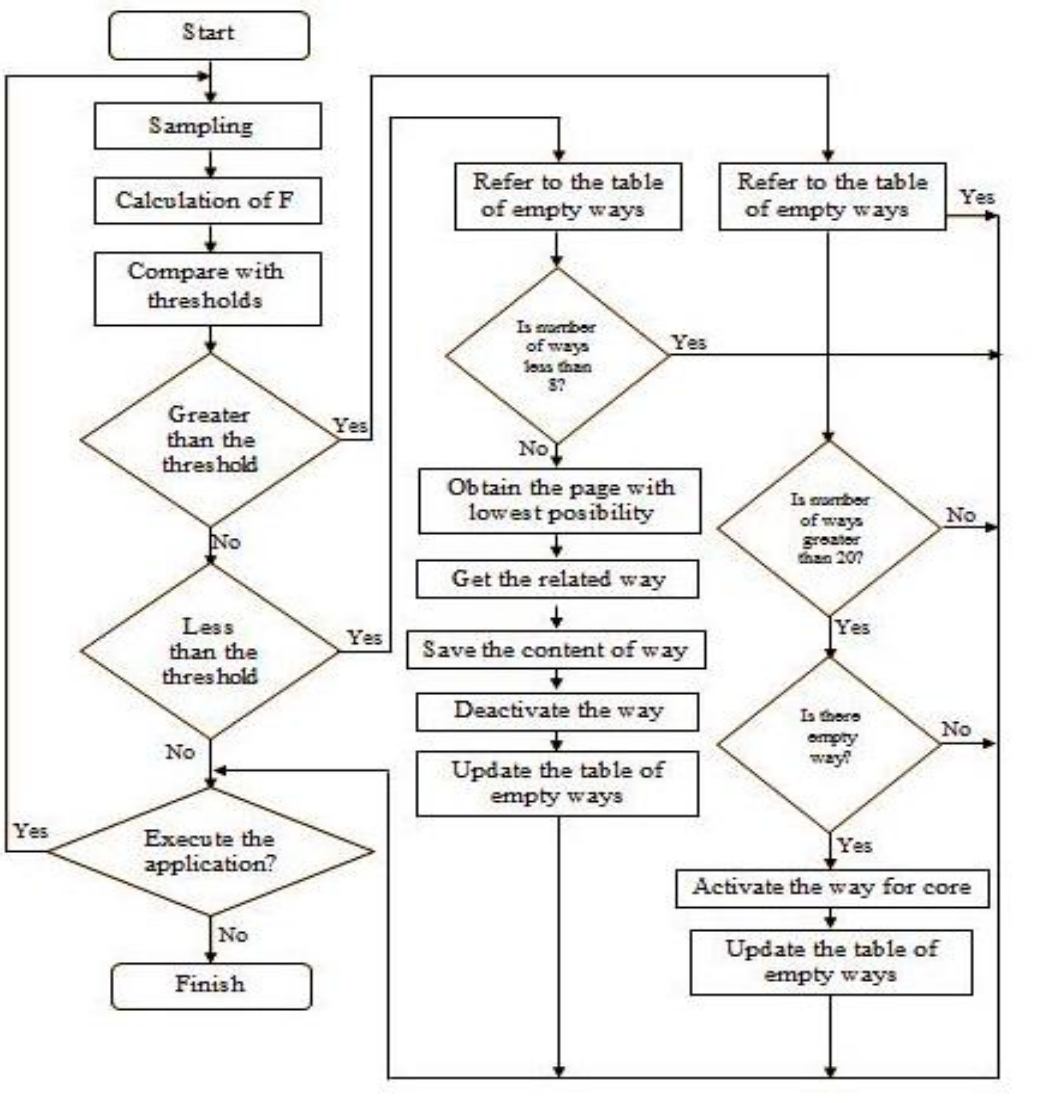


**Fig. 5.** The flowchart of the proposed algorithm

### 3.1. Phase one

In this phase, F algorithm was compared with high and low limits. In case *F* value is lower than the low limit (*F<T1*), one of the ways of a core can be reduced since the application has the locality feature by itself and the number of local pages is limited. Hence, inasmuch as all the local pages are available in the cache of the respective core, one of the ways of the core can be deactivated. This way can be used by other cores. However, in case the *F* value is greater than the high limit of the threshold (*F>T2*), it means that the application is not locally enough and there is a lot of distance among pages which are reused. Thus, the need for additional cache for the respective core is essential. That is, by adding cache, fewer pages are removed from the cache. As a result, the algorithm requests the additions of one more way to the cache. If the obtained F value is within the range of high and low limits, the output of the initial phase is considered to be stable. That is, there is no need for increasing or reducing the ways of the cache for the respective core.

(1) If *F<T1*: Decrease is requested (Dec)

(2) If *F>T2*: There is a request for increase (Inc)

(3) If the stages of one and two are not correct: There is a request for not changing or it is kept stable (keep)

### 3.2. Phase two

Examining the results of simulations for the first phase indicates that these results do not improve the efficiency; rather, they reduce the efficiency. Thus, it can be argued that another phase is needed. In this phase, three bits were used for reducing the speed of change [33]. The state diagram is given in figure 6. As shown in the diagram, the increase or decrease is not immediately carried out which prevents the rapid realization of changes in the algorithm.

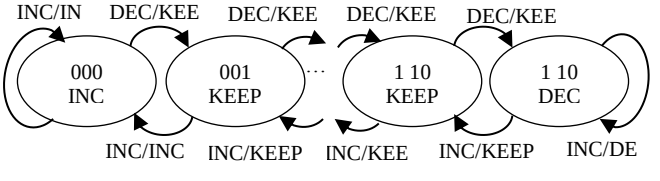


**Fig. 6.** Three-bit state diagram preventing immediate realization of changes in the algorithm [33]

### 3.3. Phase three

This phase is used for the proper selection of a page which should be taken out. The outputs of the second phase are: increase, decrease and stable. For the stable state of the way, another phase is not needed. However, a third phase is needed for increase or decrease. The dynamic cache segmentation algorithm has a set of inputs; these inputs as well as obtained probability of each phase are used to properly predict the future events. Hence, in phase three, the proposed algorithm is used to dynamically segment the cache. The inputs of the dynamic segmentation algorithm are pages which are fed into the cache by the respective application. Each of the pages is located in one way. In this stage, a page replacement method such as LRU was used. This algorithm is separately implemented on each of the cores and each core has its own output. The outputs of the second phase are used to consider the following probabilities:

(1) The output of the second phase is considered to be stable: In this case, no ways are activated or deactivated and the algorithm is re-implemented by starting the sampling operation.

(2) The output of the second phase is assumed to be an increase: In this case, we go towards a core which has requested a way increase and one of the inactive ways is activated for the intended core so that more pages can be included in the core.

(3) The output of the second phase is a decrease: In this case, firstly, the algorithm of the third phase is used to find the page which should be taken out; then, the way of that page is deactivated. Nevertheless, its data is stored before it is removed.

Consequently, the cores use the cache space optimally. Moreover, the cache space is divided fairly among the cores. That is, the core for which more space had been allocated at first voluntarily gives its additional cache space to the cores which had requested more space. Thus, algorithm operates in such a way that all the cores achieve the highest efficiency as a product of working in harmony with each other. The output of the third phase is considered to be the ultimate output of the algorithm; then, the algorithm starts again.

## 4. SIMULATION RESULTS

For conducting the simulations, at first, the applications are executed in the second level cache so as to obtain the re-access distance. For doing so, Gem5 simulator was used [24]. This simulator operates in Linux operating system which is particularly used for simulating memory in multi-core processors.

### 4.1. Simulation parameters

For simulating cache memory, a memory compatible with way was used. That is, it was possible to activate or deactivate the cache ways separately. For evaluating the performance and efficiency of the proposed algorithm, a precise clock cycle created by Simple-Scalar was used [25]. Each application was firstly executed for 5E+05 time period and the obtained information was used to determine the re-access distance. The applications were executed for 1E+08 clock cycle. Table 1 illustrates the applications designed for the simulator based on Alpha 212-64 [26].

**Table 1.** Applications that were used in the simulation

| *Suites* | *Benchmarks* | *INS Exec* | *Cache Ref* |
|---|---|---|---|
| SPECint95 | *go* | 100M | 29M |
| | *gcc* | 100M | 37M |
| | *compress* | 78M | 3M |
| | *perl* | 19M | 7M |
| SPECfp2000 | *ammp* | 19M | 8M |
| | *art* | 100M | 14M |
| | *equake* | 100M | 37M |
| Media Bench | *mpeg2encode* | 100M | 24M |
| | *mpeg2decode* | 100M | 18M |
| | *g721-encode* | 100M | 46M |
| | *g721-decode* | 100M | 46M |

For conducting the experiments, SPEC int/fp [27] and Media Bench [28] were used. Table 2 shows the input parameters of the simulation for the above-mentioned applications.

**Table 2.** Simulation parameters

| *Parameter* | *Value* |
|---|---|
| Fetch queue | 4 entries |
| Branch predictor | comb(bimodal, 2-level gshare)<br>bimodal - 2048 entries<br>gshare Level1 1024<br>Level2 4096(global)<br>Combining pred. 1024 entries |
| Branch misread | RAS entries-32; BTB-1024 * 2ways |
| Latency | 10 cycles |
| Decode width | 4 instructions |
| Issue width | 4 instructions |
| Commit width | 4 instructions |
| Register update unit | 16 entries |
| Load/Store queue | 32 entries |
| Instruction TLB | 64*4-way, 8K pages, 30 cycles |
| Data TLB | 128* 4-way, 8K pages, 30 cycles |
| Memory latency | 80 cycles |
| Memory access bus | 32 entries |
| Functional Units | Int 4, FP 2 |
| L1 I-Cache | 8KB, 16-Way32B line, 2 cycles |
| L1 D-Cache | 8KB, 16-Way32B line, 2 cycles |
| L2 unified-Cache | 512KB, 16-way128B line, 12 cycles |

As shown in this table, parameters such as fetch queue including four entries and branch predictor 1024 with hint 10 for the first level and 4096 entries with 10 cycles delay for the second level were used in the simulation. It should be noted that I-cache used in the table denotes the instruction cache and D-cache refers to the data cache.

### 4.2. Examining the increase of active ways

#### 4.2.1. Investigating active way enhancement separately

In this section, the figures related to the active ways in the applications are examined. By active ways, researchers refer to the ways which include data within themselves. In other words, they are not empty. In case an application does not use its ways actively, it means that consumption energy is wasted. That is, the application has allocated ways to itself but it does not use them. As shown in table 1, the applications used in the simulation are divided into three groups: the first group of applications, as shown in figure 7, has little locality feature and use all the cache ways. In this group of applications, all the four applications utilize their maximum cache. In the second group of applications, although each core has sixteen ways, they use only twelve of those ways. Hence, the algorithm can be used to partition the cache more precisely. The third group of applications has locality feature; hence, they do not need to increase their ways. Moreover, all the applications which are executed have strong locality feature. Hence, the proposed algorithm does not enhance the active ways remarkably. Nevertheless, the probability that all the executed applications behave similarly is very meager. Thus, for proper examination of the proposed algorithm, it is better to execute the applications in the merged mode.

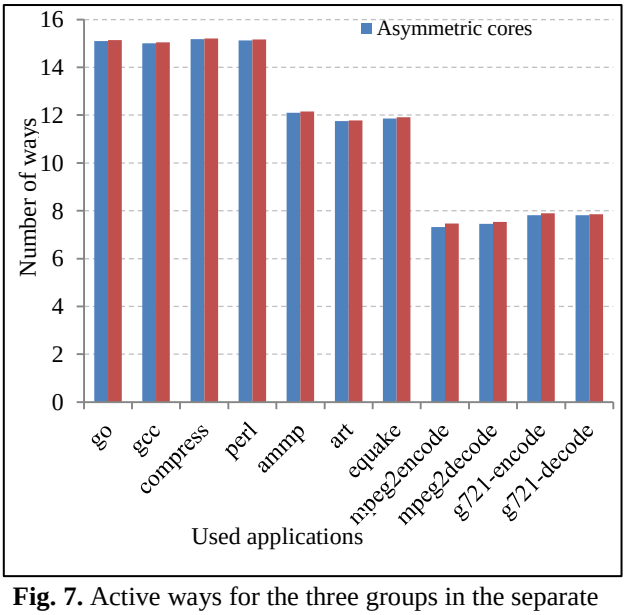


**Fig. 7.** Active ways for the three groups in the separate mode

#### 4.2.2. Examining the active ways in the merged mode

##### 4.2.2.1. The first merged method

In this method, the applications of the first group (SPECint95) and the second group (SPECfp2000) were used in a merged way. The applications *go* and *gcc* from

the first group and the applications *ammp* and *art* from the second group were used for execution. The results of this simulation are depicted in figure 8. As it can be observed from this figure, the proposed algorithm was able to enhance the number of active ways of the first group significantly; this increase is attributed to the fact that the proposed algorithm managed to get the ways unused by the second group applications from respective cores and put them at the disposal of the first group applications. Hence, not only the applications are executed at a high rate but also the cache memory is used optimally.

As noted above, the number of active ways for *gcc* increased from 15 to 17 ways. Regarding *ammp*, it can be argued that the algorithm gets surplus ways and allocates them to *gcc*.

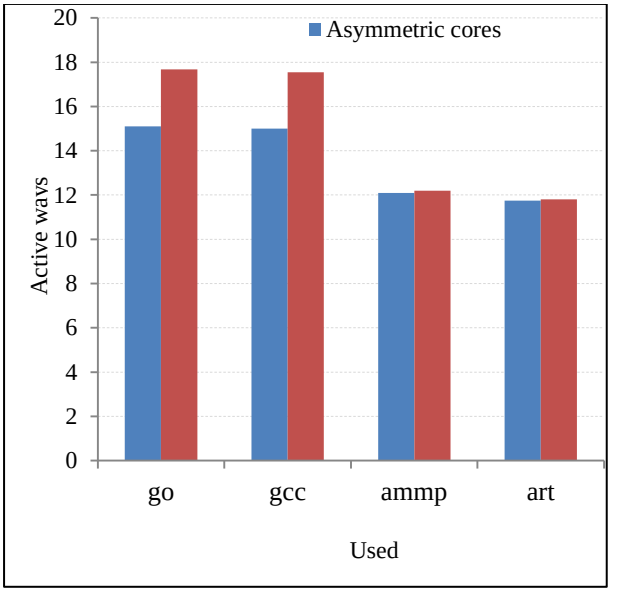


**Fig. 8.** Active ways in the first merged method

#### 4.2.2.2. The second merged method

In this method, the first group applications (SPECint95) and the third group applications (Media Bench) were used. The applications *go* and *gcc* from the first application and the applications *g721-encode* and *g721-decode* from the third application are used for execution. The results of this simulation are illustrated in figure 9.

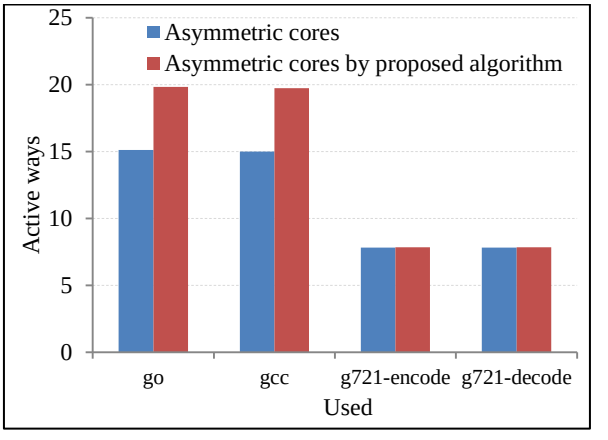


**Fig. 9.** Active ways in the second merged method

As shown in this figure, the proposed algorithm in the second merged method was able to enhance the number of active ways in the first group applications more than that of the first group application. This is attributed to the fact that the number of ways unused by the third group applications is more than that of the second group application. Thus, it can be pointed out that the proposed algorithm can dedicate more active ways to the first group applications. As a result, not only are the applications executed faster but also the cache memory is used optimally. Accordingly, the active ways for *gcc* increased from 15 ways to 20 ways. Figure 10 illustrates the *gcc* application under three states. This figure indicates that the proposed algorithm manages to enhance active ways of the *gcc* application.

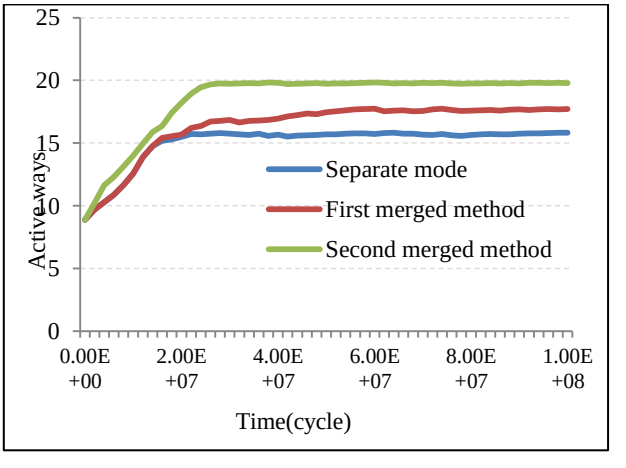


**Fig. 10.** Comparison of active ways in *gcc* application

### 4.3. The examination of miss rate in executing applications in the merged mode

#### 4.3.1. The first merged method

Noticing figure 11 indicates that the proposed algorithm was able to significantly reduce the miss rate

in the first group of applications. This is attributed to the fact that the proposed algorithm takes the cache not used by the cores of the second group applications and gives them to the first group applications. Consequently, the applications are executed faster and the cache is used optimally.

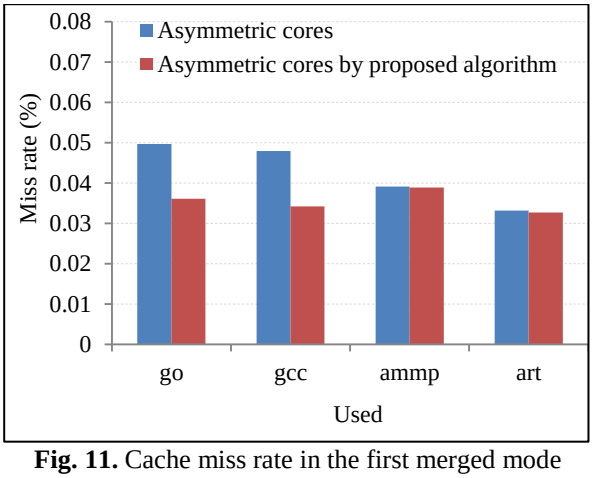


**Fig. 11.** Cache miss rate in the first merged mode

### 4.3.2. The second merged mode

As illustrated in figure 12, in the second merged mode, the proposed algorithm was able to reduce the miss rate of the first group applications more than that of the first merged mode. This is attributed to the fact that the unused cache by the third group applications is more than that of the second group applications. Hence, it can be mentioned that the proposed algorithm can allocate more cache to the first group applications.

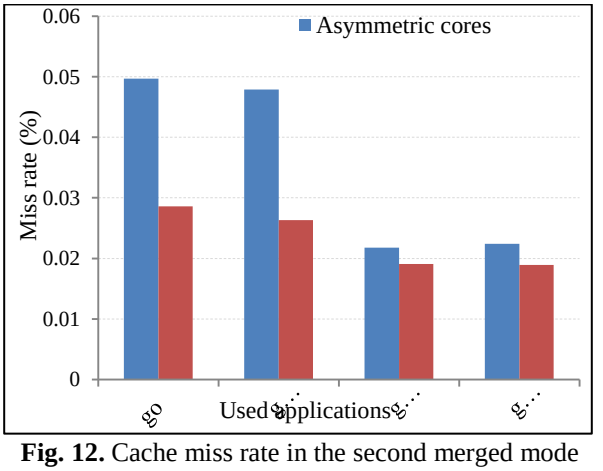


**Fig. 12.** Cache miss rate in the second merged mode

Now, the results obtained from the proposed algorithm are compared with those of other studies. Inasmuch as the proposed algorithm is software-based and partitions the cache dynamically among the cores, studies should be selected for comparison which has the same features. In other words, the selected studies should be fundamentally comparable with the proposed algorithm in the present study. In [22], the cache miss rate was mentioned with respect to the effective size of the cache. The method proposed in [22] was able to reduce the cache miss rate. For comparing the results of the algorithm proposed in the present study with those related to [22], art, mesa and swim applications were considered. It was observed that the algorithm proposed in this paper reduces the cache miss rate better than the method used in [22]; because the proposed algorithm uses the cache space more effectively. In other words, the proposed algorithm selects cache for each core with regard to the application which is executed and uses the free space of cache optimally. As depicted in figure 13, the swim application has less locality feature; as a result, the proposed algorithm operates better and brings about more significant reduction. Furthermore, due to the accurate selection of the page to be removed and due to preventing instantaneous fluctuations in the algorithm, the curve obtained by the proposed algorithm in this paper is more uniform and consistent than the curve obtained in [22]. Art application, as shown in figure 14, has more locality feature. Accordingly, there is not much difference between the two methods. However, mesa application, shown in figure 15, is somewhere between the swim and art applications in terms of the locality feature and has less changes.

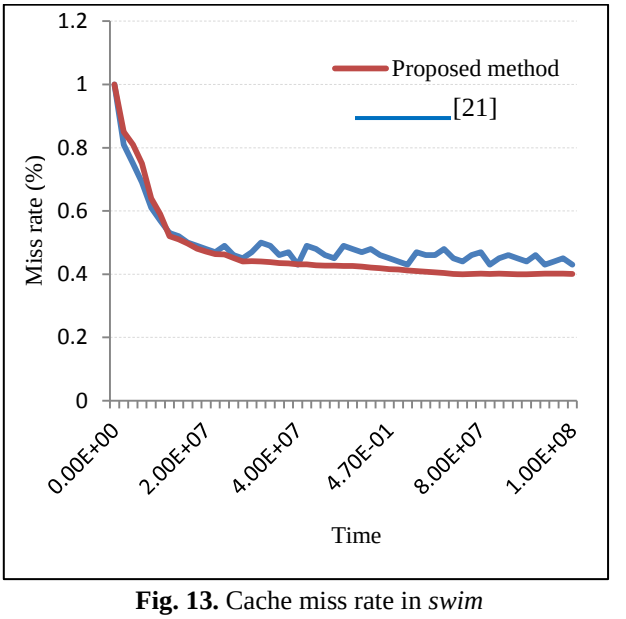


**Fig. 13.** Cache miss rate in *swim*

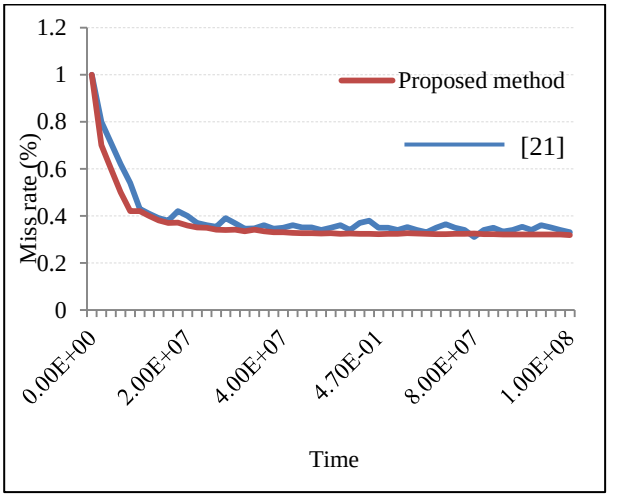


**Fig. 14.** Cache miss rate in *mesa*

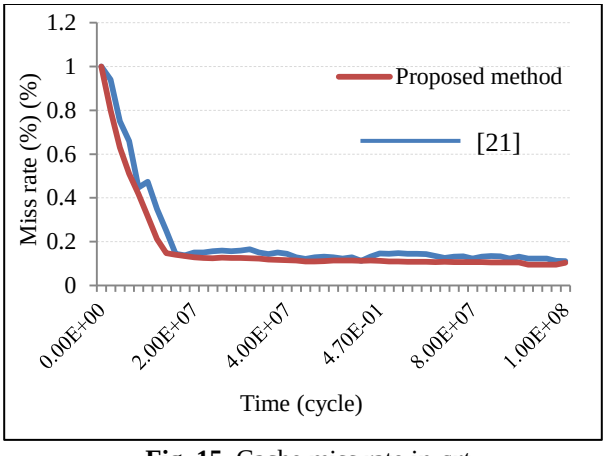


**Fig. 15.** Cache miss rate in *art*

## 5. CONCLUSION

In the architecture of multicore processors, cache memory is considered to be the efficiency bottleneck. In this paper, having introduced the concept of cache re-access, the researchers proposed a dynamic algorithm for partitioning and segmenting cache memory in multi-core processors. The simulation results revealed that the proposed algorithm significantly improved the execution of applications and applications, especially applications with little locality feature. Having examined the active and inactive ways in cache memory, the researchers concluded that increasing the active ways results in the enhancement of cache efficiency. Also, IPC increases. Furthermore, cache miss rate in different applications was investigated; it was observed that the proposed algorithm reduces the cache miss rate more than similar methods. It was pointed out in the paper that the study of asymmetric multi-core processors is a novel research issue which is of high significance. The present study addressed the research gap on cache which is deemed to be a bridge between the main memory and core. Nevertheless, more research should be conducted on this novel and under-researched issue. Recently, as a result of the extensive development and utilization of mobile phones, portable computers, etc. the critical need for saving consumption energy is increasingly underscored. One direction for further research in this area is the investigation of the power consumption of the processors.

## REFERENCES

[1] F. J. Pollack, **"New micro architecture challenges in the coming generations of CMOS process technologies"**, *in Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, 1999, pp. 2.

[2] G. E. Moore, **"Cramming more components onto integrated circuits"**, *Electronics*, Vol. 86, No. 1, 1998, pp. 82-85.

[3] C. McNairy and R. Bhatia, **"Montecito: a dual-core, dual-thread titanium processor",** *IEEE Micro*, Vol. 25, No. 2, 2005, pp. 10–20.

[4] S. Naffziger, B. Stackhouse, T. Grutkowski, D. Josephson, J. Desai, E. Alon and M. Horowitz, **"The implementation of a 2-core, multi-threaded titanium family processor",** *IEEE Journal of Solid-state circuits*, Vol. 41, No. 1,2005, pp. 197–209.

[5] A. Carbine and D. Feltham, **"Pentium pro processor design for test and debug"**, *IEEE Design &Test of Computer*, Vol. 15, No. 3, 1998, pp. 77–82.

[6] J. W. Langston and X.He, **"Multi-core Processors and caching: A Survey"**, http://blogs.cae.tntech.edu/jwlangston21/files/2008/08/multi-core-processors-and-caching-a-survey-ieee-format.pdf

[7] V. Romanchenko, **"Evaluation of the multi-core processor architecture Intel core: Conroe, Kentsfield"**, in *Digital-Daily.com*, 2006.

[8] V. P. Heuring and H. F. Jordan, **"Computer Systems Design and Architecture",** *Prentice Hall*, 2004.

[9] J. L. Hennessy, D. A. Patterson, **"Computer architecture: a quantitative approach"**, *Morgan Kaufmann Publishers*, 2007.

[10] D. Tam, R. Azimi, L. Soares and M. Stumm, **"Managing shared L2 caches on multi-core systems in software"**, in *Workshop on the Interaction between Operating Systems and Computer Architecture*, 2007, pp. 26-33.

[11] F. Guo and Y. Solihin, **"An analytical model for cache replacement policy performance"**, *ACM SIGMETRICS Performance Evaluation Review*, Vol. 34, No. 1, 2006, pp. 228-229.

[12] H. Kannan, F. Guo, L. Zhao, R. Illikkal, R. Iyer, D. Newell, Y. Solihin and C.Kozyrakis, **"From chaos to QoS: case studies in CMP resource management"**, in *ACM SIGARCH computer Architecture News*, Vol. 35, No. 1, 2007, pp. 21-30.

[13] M. Qureshi and Y. Patt, **"Utility-based cache partitioning: a low overhead, high-performance, runtime mechanism to partition shared caches"**, in *Micro 39*, 2006, pp. 422-432.

[14] S. Cho and L. Jin, **"Managing distributed, shared L2 caches through OS-level page allocation",** in *Micro 39*, 2006, pp. 455-468.

[15] L. Jin and S. Cho, **"Better than the two: exceeding private and shared caches via two-dimensional page coloring"**, in *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2007.

[16] A. Asaduzzaman, F. N. Sibai, and M. Rani, **"Impact of level-2 cache sharing on the performance and power requirements of homogeneous multi-core embedded systems"**, *Microprocessors and Microsystems, Embedded Hardware Design*, Vol. 33, No. 5,2009, pp. 388-397.

[17] R. Manikantan, K. R. Govindarajan, **"Nucache: an efficient multi-core cache organization based on next-use distance"**, *in the proc of the 17th International Computer Architecture*, 2011, pp. 243-253.

[18] M. D. Hill and A. J. Smith, **"Evaluating associativity in cpu caches",** *IEEE Transactions on Computer's,* Vol. 38, No. 12, 1989, pp. 1612–1630.

[19] D. Chandra, F. Guo, S. Kim and Y. Solihin, **"Predicting inter-thread cache contention on a chip multi-processor architecture"**, In *HPCA*, 2005, pp. 340–351.

[20] R. Iyer, **"CQOS: A framework for enabling QoS in shared caches of CMP platforms"**, in Proc. *Annual International Conference on Supercomputing*, 2004, pp. 257–266.

[21] C. Xu, X. Cheny, R. P. Dicky and Z. M. Mao, **"Cache contention and application performance prediction for multi-core systems"**, in *Performance Analysis of Systems & Software (ISPASS)*, 2010, pp.76-86.

[22] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm, **"Rapid MRC: approximating L2 miss rate curves on commodity systems for online optimizations"**, *ACM SIGARCH Computer Architecture News*, Vol. 37, No. 1. ACM, 2009, pp. 121–132.

[23] D. Kaseridis, M. F. Iqbal and L. K. John, **"Cache friendliness-aware management of shared last-level cachesfor high performance multi-core systems"**, *IEEE transactions on computers*, Vol. 63, 2014, pp. 874-887.

[24] N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S. K. Reinhardt, **"The m5 simulator: Modeling networked systems"**, *IEEE Micro*, Vol. 26, No. 4,2006, pp. 52–60.

[25] T. Austin, E. Larson and D. Ernst, **"Simple scalar: an infrastructure for computer system modeling"**, *IEEE Computer*, Vol. 35, No. 2, 2002, pp. 59–67.

[26] Compaq. Alpha 21264 Microprocessor Hardware Reference Manual. Technical report, Compaq Computer Corporation, 1999.

[27] The Standard Performance Evaluation Corporation. http://www.spec.org/.

[28] C. Lee, M. Potkonjakand W. H. M. Smith, **"Media bench: a tool for evaluating and synthesizing multimedia and communications systems"**, In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Micro architecture*, 1997, pp. 330–335.

[29] S. M. Khan, A. R. Alameldeen, C. Wilkerson, J. Kulkarni and D. A. Jiménez, **"Improving multi-core performance using mixed-cell cache architecture,"** *IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 119-130.

[30] W. Zang and A. G. Ross, **"A single-pass cache simulation methodology for two-level unified caches"**, *IEEE International Symposium on Performance Analysis of Systems & Software*, Vol. 0, 2012, pp. 168-177.

[31] A. Asaduzzaman, V. R. Suryanarayana, F. N. Sibai, **"On level-1 cache locking for high performance low-power real-time multi-core systems"**, *Computers and electrical engineering*, Vol. 39, 2013, pp. 1333-1345.

[32] I. Kotra, **"Performance and power aware cache memory architectures"**, *Ph.D. thesis, Department of Computer and Mathematical Sciences*, TOHOKU University, Sendai, Japan, 2009.