Development of a Thread Scheduler for Global Aggregation of Sibling Threads

Yamada, Satoshi
九州大学大学院システム情報科学府情報知能工学専攻

Kusakabe, Shigeru
九州大学大学院システム情報科学研究院情報知能工学部門

https://doi.org/10.15017/1654544

出版情報：九州大学大学院システム情報科学紀要. 13 (2), pp.69-74, 2008-09-26. 九州大学大学院システム情報科学研究院
バージョン：
権利関係：

九州大学大学院
システム情報科学紀要
第 13 巻　第 2 号　平成 20 年 9 月

Research Reports on Information Science and
Electrical Engineering of Kyushu University
Vol.13, No.2, September 2008

Development of a Thread Scheduler for Global Aggregation of Sibling Threads

Satoshi YAMADA* and Shigeru KUSAKABE**

(Received June 15, 2008)

Abstract: Chip-level multiprocessors (CMP) have multiple processing cores (Cores) and generally have their cache shared by each Core. On CMP, the combination of threads running simultaneously on different Cores as well as the order of threads running on one Core influences the utilization of the cache. We consider that an OS level thread scheduler for concurrent and parallel thread execution is the key to utilize the cache and reduce the memory accesses. Previously, we have developed a thread scheduler which recognizes the memory address space of each thread for concurrent execution and investigated its effect on a single processor environment. In this paper, we demonstrate the extension of our previous scheduler for parallel execution on CMP. Our scheduler is composed of the independent schedulers per Core and is able to let them cooperate with little cost. According to our investigation with Sysbench benchmark, this extension enhances the effect of our previous scheduler and results in the more reduction of the execution time.

Keywords: Thread scheduling, Multi-threaded application, Parallel execution, Cache misses, Chip Multi-Processing, Memory address space

1. Introduction

The memory access latency remains one of the major bottlenecks on CMP as well as in conventional single processors[1),4)]. CMP has multiple Cores and each Core generally shares a cache, mostly the level 2 cache (L2 cache). Therefore, the data contention occurs on the L2 cache and degrades the performance when simultaneously running threads on different Cores consume a large amount of different memory area. Therefore, the key to utilize the L2 cache is not only the order of threads executed on each Core but also the combination of threads executed simultaneously on different Cores on CMP[4)]. In general, the combination and the order of threads is controlled by a thread scheduler inside Operating System (OS). To utilize the L2 caches, an efficient OS level thread scheduler is necessary [1),4)].

Previously, we have developed a thread scheduler for the efficient concurrent execution on a commodity single processor[2)]. Our scheduler gives higher priority for threads sharing the same address space (sibling threads) and executes sibling threads in sequence. We expect our scheduling is effective because we presume that the sibling threads share a certain amount of memory area to be accessed (working set). If we sequentially execute the threads sharing their working set, we can expect that the previously executed threads leave their working set on the L2 cache, which will be accessed by the following threads. Therefore, the following threads can reduce the amount of working set to load from the memory. We call this scheduling as "time aggregation". We have confirmed the efficiency of the time aggregation scheduler in reducing the memory access frequencies with little cost.

In this paper, we extend the idea of the time aggregation to "global aggregation" and investigate its effect. The global aggregation scheduler executes the sibling threads nearly simultaneously on different Cores of CMP. As we presume that the sibling threads share the working set, we expect that the data contention is mitigated by the global aggregation and the L2 cache misses decrease. As a matter of course, accesses to the same working set from different Cores can cause the data conflicts and the data coherence problem, which could have the processor stall. However, we presume the utilization of the L2 cache to reduce the memory access is the top priority because of the previous research indication[1),3),4)].

As we mention above, our scheduler focuses on the memory address space of each thread. Therefore, our scheduler only works for applications using multiple OS level threads and it seems that the range of application is limited. However, many modern applications have become multi-threaded in

* Department of Computer Science and Communication Engineering, Graduate Student

** Department of Computer Science and Communication Engineering

accordance with the spread of Simultaneous Multi-Threaded (SMT) and CMP. In addition, many languages such as Java, Perl, Ruby, Python, and Erlang now support the development of multi-threaded applications. Moreover, we have the compiler support to develop multi-threaded applications such as OpenMP, MPI, and Open64. Therefore, we expect we will have more multi-threaded applications in the future and our scheduler will be effective in more general cases.

The rest of the paper is organized as follows. Section 2. introduces several related works and clarifies the position of our research. Section 3. explains the implementation of our scheduler. Section 4. evaluates the efficiency of our scheduler with "memory" program in Sysbench benchmark suite[5]. Section 5. concludes the paper.

2. Related Works

The utilization of many mechanisms, such as caches, registers, pipelines, etc., influences the performance of a parallel execution. An advantageous thread scheduling is achievable if the OS could obtain the information of how each thread uses those mechanisms before scheduling the next thread. In practice, the detailed behavior of a thread is platform specific and the OS cannot obtain the information beforehand. As we introduce later, it is popular to sample the information of each thread during its execution to guess the succeeding behavior of the threads. However, sampling too much information can be significantly large cost and degrade the performance. Thus, an efficient thread scheduling is a matter of balance between the cost of sampling information and the effect from the scheduling.

The approach of Fedorova[1] is to reduce the overflow of the L2 cache. They sample the size of actual memory space consumed by each thread during its execution. According to this sampled information, the OS selects the group of threads whose sum of the memory consumption size would be within the L2 cache size. Our approach also aims to reduce the L2 cache misses. However, our approach does not sample the information of each thread during its execution. Our scheduler only recognizes the memory address spaces, hence the mechanism of our scheduling is simple and the cost of the scheduling is small.

The approach of Ogawa[4] also focuses on the L2 cache misses on SMT. They define the "affinity" of threads as how much working set each thread shares with other threads. This "affinity" is calculated by counting the actual L2 cache misses during the execution of each thread. The OS scheduler selects the combination of threads with high "affinity" and execute them simultaneously on SMT. Although their thread scheduler is similar to ours in that it is effective when one thread shares the working set with others, our scheduler does not sample the information of the threads during their execution.

The basic idea of our scheduler is similar to that of Chen[3]. Their scheduling idea is to run the threads sharing the same working set simultaneously on different Cores of CMP. To recognize if a thread is sharing the same working set with others, they analyze the applications and schedule threads statically. They logically certify the efficiency of their scheduling and also demonstrate the effect on their simulator. The difference from our work is that our scheduler works dynamically and does not require statical analysis of applications. Moreover, our research investigates the efficiency of our scheduler on a real system with CMP.

3. Thread Scheduler for Global Aggregation of Sibling Threads

We implement our scheduler by modifying Completely Fair Scheduler (CFS) in Linux. We explain CFS first in Section 3.1. Next, we show the implementation of our scheduler in Section 3.2.

3.1 Completely Fair Scheduler

The thread scheduler in Linux is altered from kernel 2.6.23 and named as Completely Fair Scheduler. One of the most distinct changes from the previous scheduler is the policy of setting the priority for each thread. The previous Linux scheduler, so called vanilla scheduler, sets the static priority based on the nice value and calculates the additive dynamic priority from the sleeping time of each thread. In CFS, the scheduler counts the execution time of each thread in nanoseconds and calculates the priority as "vruntime" based on the execution time. CFS sets the higher priority for the threads with less vruntime to accomplish the fair usage of CPU among the threads. The runqueue of CFS is composed of Red-black tree, where each node represents the thread and the value of each node represents the vruntime of each thread. The leftmost node in the runqueue has the smallest vruntime and should be scheduled next by CFS. The vruntime of a thread is updated during its execution and the structure of the runqueue is updated when a thread is enqueued. This runqueue exists per Core and independent schedulers work on different Cores.

CFS does not recognize the memory address space of each thread because its efficiency is not well investigated yet. We assume we will have more multi-threaded applications as we mention above. We have confirmed that considering the memory address space can be effective in executing multi-threaded applications. Therefore, we decide to modify CFS to recognize the memory address space. We show the implementation of our scheduler in Section 3.2.

3.2 Implementation

First, we show the implementation of the time aggregation in Section 3.2.1. Next, we show the extension to achieve the global aggregation in Section 3.2.2.

3.2.1 Time Aggregation

The basic idea of implementing the time aggregation in CFS is similar to our previous implementation in vanilla scheduler[2]. First, we insert a flag in the thread structure to recognize if the thread has the sibling threads or not. When a thread creates a sibling thread, the OS sets the flag and link the thread with the list of its sibling threads. In the previous implementation of the time aggregation in vanilla scheduler, we just link the enqueued sibling thread at the end of the list. In case of CFS, we create the list of sibling threads according to the order of the vruntime of each thread. We show the example of our implementation in **Fig. 1**.

Figure 1 shows the runqueue of CFS and the additional links of the sibling threads for the time aggregation. The circles in **Fig. 1** represents the threads. Each thread in the runqueue is linked with the solid lines. The numbers in the threads show the vruntime. The currently executed thread A is dequeued from the runqueue. Notice the vruntime of thread A is increased during its execution and is now more than that of the leftmost node. The pattern of the node represents the memory address space. The color of red or black in Red-black tree is ignored. We link the sibling threads in the order of their vruntime and the link is shown with the dotted lines. The link of the sibling threads begins with the structure of the memory address space, mm_struct. When we enqueue a thread X with vruntime of 17, we look for the point to link the thread with the list of its sibling threads. In case of thread X, it has the same memory address space with thread D and E. According to its vruntime, thread X is linked between thread D and E.

After executing thread A, CFS chooses thread B as the next thread. Our scheduler for the time aggregation recognizes thread C as another candidate. We set the bonus vruntime for the time aggregation (time_bonus) in advance and calculate the expression below ($Vrun\_Z$ shows the vruntime of thread Z).

$$Vrun\_B \geq Vrun\_C - time\_bonus \quad (1)$$

If the expression (1) is true, then we choose thread C. If we set the bonus vruntime more than 6, we choose thread C as the next thread. Otherwise, we choose thread B. We can also tune the vruntime manually using a system call we implement.

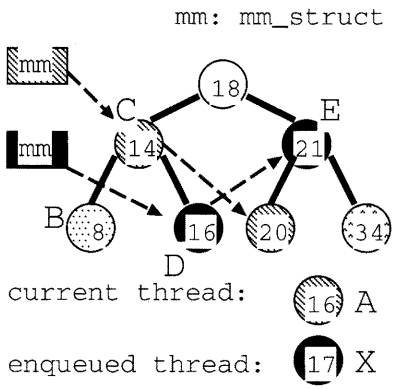

Fig. 1 Implementing the time aggregation in CFS.

3.2.2 Global Aggregation

We extend the idea of the time aggregation to accomplish the global aggregation. First, we run independent schedulers per Core as CFS does and each scheduler runs for the time aggregation. When the scheduler on Core 0 finds the chance of the time aggregation, it sets the pointer (global_mm) to its memory address space of those sibling threads. Otherwise, global_mm is NULL. Only one global_mm exists per OS. The schedulers on the other Cores can only refer to global_mm. When global_mm is set, the other schedulers look for the sibling threads with the memory address space of global_mm from their own runqueue. If there exists a sibling thread, the scheduler considers the thread as the third candidate with bonus vruntime. We show the example in **Fig. 2**.

Figure 2 shows the example on a dual core platform. Thread A is running on Core 0 and thread E is running on Core 1. Threads enqueued into each runqueue is shown in the order of their vruntime from the left. In other words, thread B on Core 0 and thread F on Core 1 are to be scheduled by CFS.

After executing thread A on Core 0, thread B and thread C are the candidates to be scheduled next because thread C is a sibling thread of thread A. If thread C is scheduled, the scheduler on Core 0 sets global_mm to the memory address space of thread A and C (solid arrow). On Core 1, the scheduler checks the global_mm in scheduling (dotted arrow). After executing thread E, thread F, G, and I are the candidates because thread G is a sibling thread of thread E and thread I is a sibling thread of thread C. Thread I has the bonus vruntime against thread F (global_bonus) and thread G (global_time_bonus). If thread I satisfies the two equations below, thread I is scheduled after thread E.

$$Vrun\_F \geq Vrun\_I - global\_bonus \quad (2)$$

$$Vrun\_G \geq Vrun\_I - global\_time\_bonus \quad (3)$$

Thus, our global aggregation scheduler can execute the sibling threads nearly simultaneously on different Cores.

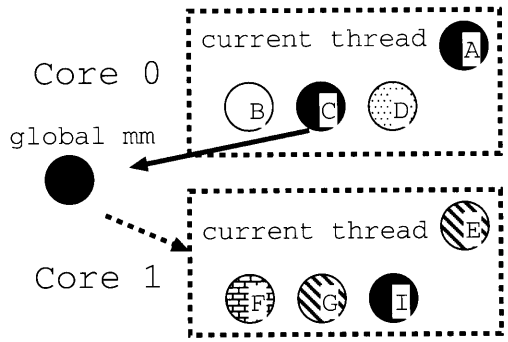

Fig. 2 Implementing the global aggregation in CFS.

4. Experimental Evaluation

We evaluate the effect of our scheduler with "memory" program in Sysbench benchmark suite[5]. The specification of our experimental platform is shown in **Table 1**. When we run "memory" programs, we apply several combinations of bonus vruntime. We express the combinations of bonus vruntime as (time_bonus, global_time_bonus, global_bonus). We try (0, 0, 0), (4K, 0, 0), (4K, 4K, 4K), (5M, 0, 0), and (5M, 5M, 5M). The analysis of using different values for global_time_bonus and global_bonus is our future work.

In this Section, we explain "memory" first. In Section 4.2, we show the result in terms of the execution time, the fairness, and the L2 cache misses.

Table 1 Specification of our experimental platform.

Processor	Intel Core 2 Duo
L2 Cache Size / Latency	2 MB / 14 ns
Memory Size / Latency	1 GB / 149 ns

4.1 "memory" program in Sysbench

In "memory" program, a specified number of threads are created to access a specified amount of memory. Sysbench provides two types of memory access, "read", reading a value from an address, and "write, writing a value to an address. We show the outline of "memory" program in **Fig. 3**. Each arrow represents the memory access of each thread. This access is repeated until the total access size of every thread exceeds a specified size. We measures the execution time for the threads to access this specified size of memory.

In this paper, we create 100 threads and let them access the memory area of 50GB in total. We experiment both "read" and "write" accesses. The default "memory" program lets one thread execute until it expires its quantum time, which makes it hard to understand the relationship with memory access size and the effect of our scheduler. Therefore, we modify "memory" program to let a thread yield after accessing a specified memory block size. When a thread yields, its vruntime is recalculated and the thread is enqueued again. We use 32KB, 512KB, 1.5MB, 2MB, and 4MB for the memory block size.

The threads in one "memory" program share the same address space. To show the effect of our scheduler clearly, we run multiple "memory" programs simultaneously. In this paper, we show the result when we run 10 "memory" programs. We run this experiment 10 times and calculate the average execution time. In addition, we compare the maximum time of running "memory" program to investigate the influence on the fairness. We also measure the L2 cache misses using the performance monitoring counter of the processor.

4.2 Result

We show the result on the average execution time (average) and the maximum execution time (max) of "memory" program in **Fig. 4**. In **Fig. 4**, the X axis shows the memory block size and the access type. The Y axis shows the ratio between CFS.

First of all, we can see the increase of the cost of our scheduler is at most 1 % when we see the average of (0, 0, 0) in most of the cases. Therefore, we consider the cost of our scheduler is little.

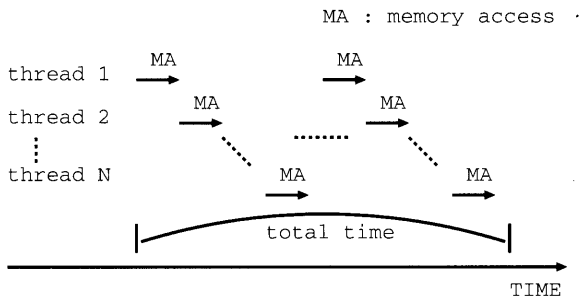

Fig. 3 Outline of "memory" program in Sysbench.

When the bonus vruntime are (4K, 0, 0) and (4K, 4K, 4K), their results are almost the same as that of CFS. When we compare (4K, 0, 0) and (4K, 4K, 4K), the average in (4K, 4K, 4K) is always the same or less than that in (4K, 0, 0). Therefore, we can say that the global aggregation can enhance the effect of the time aggregation.

When we increase the bonus, the effect of the global aggregation on the average becomes clear. When we set the bonus as (5M, 0, 0), we can see the effect of the time aggregation especially in 32KB and 512KB. However, when we set the memory block size as 1.5MB and more, the effect gradually declines. We consider this is because the L2 cache is overflowed by the access from each Core and the memory access size of around 1.5MB is the boundary that employing only the time aggregation could be effective. On the other hand, when we set the bonus as (5M, 5M, 5M), we can see more reduction of the average even in 2MB and 4MB of the memory block size. We consider that the chance of sharing the same working set on the different Cores increases by the global aggregation, which results in the more reduction of the average. Furthermore, we can see these positive effect in both "read" and "write" access. From this results, we expect that our scheduler can be effective in broad range of applications.

To discuss the fairness, we compare the max. The max is not changed or decreased in most of the cases. Therefore, we consider that the possibility that our scheduler disturbs the fairness is low.

We see the exceptionally large increase of the average in (0, 0, 0) and the max when we apply "write" access for 1.5MB. However, we do not find any rational explanation about this increase and we consider this is a platform specific result.

We show the result of the L2 cache misses in **Fig. 5**. In **Fig. 5**, the X axis shows the memory block size and the access type. The Y axis shows the actual L2 cache misses. We see the totally big decline of the L2 cache misses in 512KB. We consider this is a processor specific result and is not related to our scheduler.

First of all, the increase of the L2 cache misses of bonus (0, 0, 0) is little. Therefore, we can say that the cost of our scheduler is small. In case of bonus (4K, 0, 0) and (4K, 4K, 4K), the effect is unstable. We consider this unstable result comes from the short of the bonus value because we see more stable effect in (5M, 0, 0) and (5M, 5M, 5M). As we can see, the L2 cache misses in (5M, 5M, 5M) is the smallest in every case we measure. Even when the bonus (5M, 0, 0) can not result in distinct reduction of the L2 cache misses in 4M, the global aggregation reduces the L2 cache misses by 14 - 19 %. Thus, we can say that the global aggregation scheduler enhances the effect of the time aggregation and is an effective thread scheduler on CMP.

5. Conclusion

In this paper, we propose the global aggregation of the sibling threads and investigate its effect. According to our measurements with "memory" program in Sysbench, we show the global aggregation is effective in reducing the execution time on CMP. In addition, the global aggregation does not cause the significant problems on the fairness, rather it reduces the execution time of all applications in most of the cases.

Our future work includes the investigation of efficiency in more practical applications, such as multithreaded web servers and database servers. Applying to the large distributed system, such as Hadoop is also included.

References

1) Alexandra Fedorova, et al., Throughput-Oriented Scheduling On Chip Multithreading Systems, *Technical Report TR-17-04, Division of Engineering and Applied Sciences, Harvard University*, 2004.
2) Satoshi Yamada, Shigeru Kusakabe, "Effect of Context Aware Scheduler on TLB", *Proc. of Workshop on Multi-Threaded Architectures and Applications, published in CD*, 2008.
3) Shimin Chen et al., "Scheduling Threads for Constructive Cache Sharing on CMPs" *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pp. 105–115, 2007.
4) Shugo Ogawa, Kei Hiraki, "A Speedup Technique with Scheduler Using Process Execution Information" *Vol.46 No.SIG 12 (ACS 11)*, pp. 161–169, 2005
5) "SysBench: a system performance benchmark", http://sysbench.sourceforge.net/

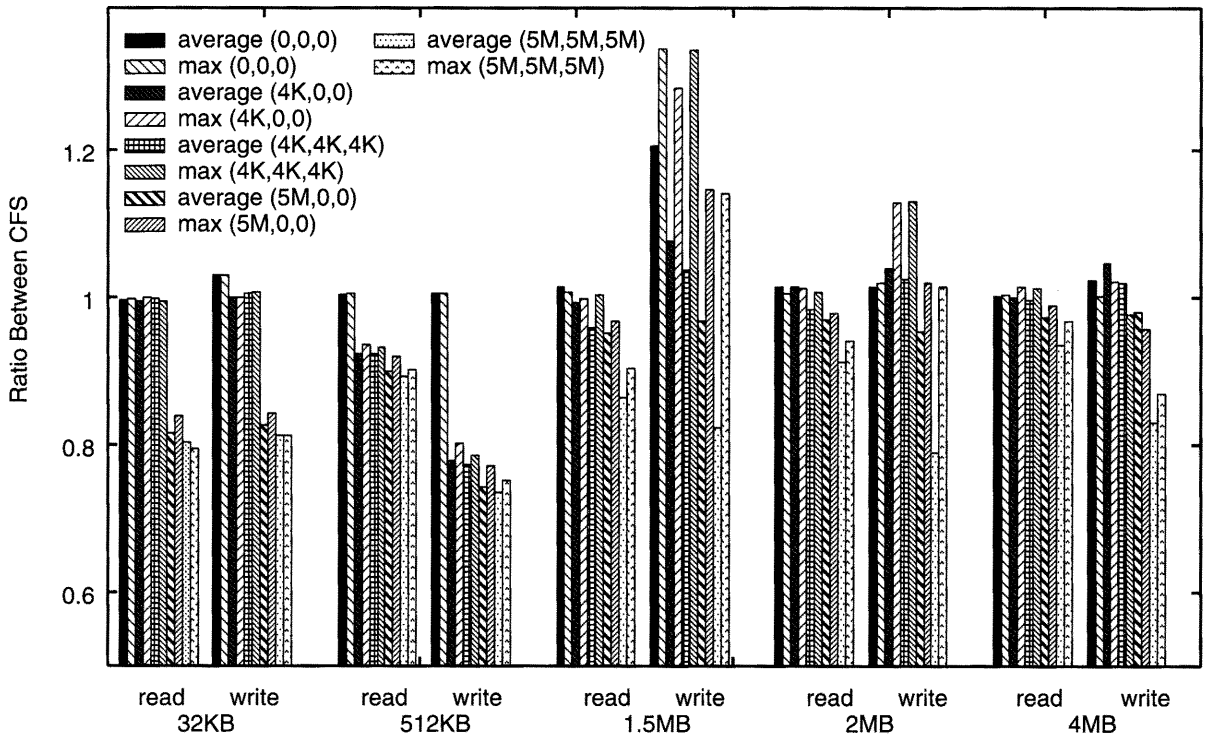

Fig. 4 Result of Executing 10 Sysbench.

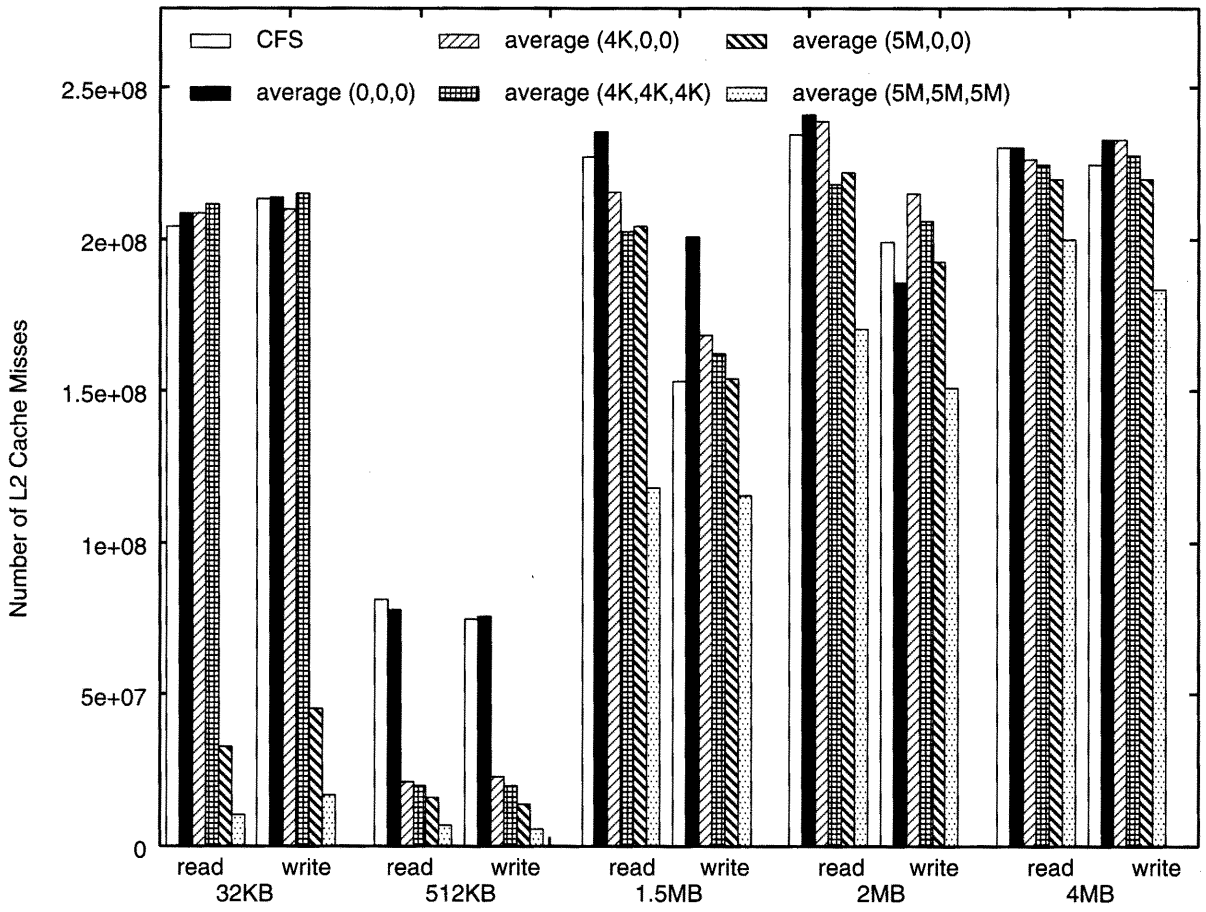

Fig. 5 Result of the L2 cache misses.