Asian Journal of Information Technology 8 (4-12): 94-103, 2009
ISSN: 1682-3915

Impact of Inter-Core Time Aggregation Scheduler on a Database Server Workload

Satoshi Yamada and Shigeru Kusakabe
Graduate School of Information Science and Electrical Engineering,
Kyushu University, 744, Motooka, Nishi-ku, Fukuoka, Japan

Abstract: In this study, we evaluate the effect of Inter-Core Time Aggregation Scheduler (ITAS) on a database server workload. ITAS is a kernel-level thread scheduler to utilize the locality of references between threads on Chip Multi-Processor (CMP) platforms. To detect the locality of reference between threads without the overhead of sampling the behavior of thread information, we focus on sibling threads, which are kernel-level threads sharing the same memory address space. We consider that we can utilize computing resources such as cache by dynamically aggregating sibling threads. We have investigated the effect of Time Aggregation Scheduler (TAS) on a single processing Core. TAS utilizes the locality of references and reduces the overhead due to context switching by executing sibling threads collectively in a group. We extend TAS into ITAS, which aggregates sibling threads on different Cores at the same time so that we can expect the effect of aggregation on CMPs. As the number of Cores increases, it is likely to run multiple multithreaded programs simultaneously to utilize all Cores on a chip. Thus, we consider that ITAS will be applicable to many situations. In this study, we show the effect of ITAS on a realistic workload of a multi-threaded database server running with a simple application server on a commodity CMP platform in terms of throughput, the number of cache misses and resource stalls and influence on the target program and other non-aggregated background programs. The experimental result indicates that ITAS enhances the performance of the database server without degrading that of non-aggregated background programs.

Key words: Thread scheduling, sibling threads, chip multi-processor parallelexecution, database server, cache

INTRODUCTION

In this study, we evaluate the effect of Inter-Core Time Aggregation Scheduler (ITAS) on a database server workload. ITAS is a kernel-level thread scheduler for Chip Multi-Processor (CMP) platforms. The basic idea of ITAS is to utilize locality of reference between threads. To utilize locality of reference between threads, we focus on Thread-Level Concurrency (TLC) and Parallelism (TLP) inside programs. To enhance the performance of programs with TLC and TLP, we previously proposed Time Aggregation Scheduler (TAS), a kernel-level thread scheduler implemented on the Linux kernel (Yamada and Kusakabe, 2008a). TAS tries to collectively execute sibling threads, which are kernel-level threads sharing the same memory address space, in sequence on a single processing Core. We expect two effects from TAS as:

- Reduction of the overhead switching memory address spaces
- Utilization of the locality of references between sibling threads and reduction of the capacity pressure on caches

We can always expect the first effect by aggregating sibling threads even though the overhead of switching memory address spaces is smaller compared to the time slice of common threads. We can expect the larger effect of utilizing cache from the second point if sibling threads share a certain amount of data to be accessed (working set).

Nowadays, multi-core processors like CMPs are widely spread as commodity platforms. CMPs have multiple Cores on a single chip and are able to execute multiple threads simultaneously. Thus, it is likely to run multiple multi-threaded programs simultaneously on a single node to utilize the Cores. It is common for Cores on a CMP to share computing resources such as L2 cache (Kundu *et al*., 2004) and it is well known that combinations of threads running simultaneously on different Cores affect the utilization of caches (Fedorova *et al*., 2005; Ogawa and Hiraki, 2005; Snavely *et al*., 2002). Therefore, we consider that it is not enough to run an independent TAS per Core in executing multiple multi-threaded programs simultaneously for the utilization of the locality of references between sibling threads. We extend TAS to

Corresponding Author: Satoshi Yamada, Graduate School of Information Science and Electrical Engineering, Kyushu University, 744, Motooka, Nishi-ku, Fukuoka, Japan

implement ITAS on the Linux kernel (Yamada and Kusakabe, 2008b), which executes sibling threads simultaneously on different Cores to utilize computing resources. The basic scheduling idea of ITAS is similar to that of gang scheduling (Feitelson and Rudolph, 1992). In gang scheduling, a single scheduler chooses related threads and executes them simultaneously on each Core.

The overhead of context switch can be heavy in gang scheduling because the scheduler has to synchronize all executed threads from different Cores. In ITAS, aggregations of sibling threads are accomplished by non-blocking communications between schedulers on different Cores. Thus, ITAS is able to schedule threads without the overhead due to synchronization of Cores.

ITAS may increase the overhead caused by the data contention or the data coherence problem by having threads executed on different Cores to access the same working set simultaneously. However, according to the previous researches on the analysis of the throughput on CMP platforms, the most influential factor is the lack of the capacity of cache (Chen *et al*., 2007; Chishti *et al*., 2005). Therefore, we expect that the effect of ITAS on the enhancement of the throughput is higher than the overhead.

We assume the use of ITAS, when we run multiple multithreaded and non-multi-threaded background programs concurrently. We expect the enhancement of the throughput of multi-threaded programs by aggregating their sibling threads. We also expect the enhancement of the throughput of the background programs by ITAS. As we mentioned, ITAS is implemented by modifying the Linux kernel. Linux has employed Completely Fair Scheduler (CFS) as its thread scheduler from the version 2.6.23. The scheduling policy of CFS is to let threads starting simultaneously with the same static priority (nice value) spend the same amount of CPU time. We do not modify this characteristic of CFS in implementing ITAS. Therefore, the quantum time of the background programs get longer as ITAS aggregates more sibling threads and keeps the background programs wait longer. Thus, the number of context switches of the background programs decreases and we can expect the enhancement of their throughput.

In this study, we evaluate the effect of ITAS on the throughput of a multi-threaded database server and other non-aggregated background programs. We use OLTP program of SysBench (2009) benchmark suites, which works as a simple application server and simulates various connections from database clients. We simultaneously run background programs, which are not aggregated by ITAS and investigate its throughput.

MATERIALS AND METHODS

Related research: Many kernel-level schedulers have been proposed to enhance the performance of thread execution on CMP platforms. Advantageous co-scheduling methods are achievable if a kernel can grasp the state of processor resources and how each thread uses the processor resources before scheduling the next thread. However, it requires a heavy load for the kernel to track all the states of processor resources and resource usage pattern of thread executions.

A major solution against this problem is to guess the preferable sequence of a thread execution from the previous behavior of the thread. Many researchers proposed to sample the information of the behavior of each thread during its execution and choose the desirable sequence of thread executions based on the sampled statistics (Fedorova *et al*., 2005; Ogawa and Hiraki, 2005; Snavely *et al*., 2002).

For example, Fedorova *et al*. (2005) calculates the size of the working set of each thread by sampling the behavior of the thread. They schedule the combinations of threads to let the sum of the working set of the combinations of threads fit within the capacity of the L2 cache. Ogawa and Hiraki (2005) sampled the number of the L2 cache misses during the execution of threads to obtain the affinity of each combination of threads.

They enhance the priority of the combinations of threads with higher affinity. These scheduling methods are advantageous in that they can handle any kind of programs. However, the overhead of dynamically sampling information can be large enough to degrade the performance. Especially, the overhead of sampling increases, when running many threads (Chandra *et al*., 2005) and optimal scheduling is difficult in many-core processors.

The basic scheduling idea of ITAS is similar to that of Chen *et al*. (2007). Their scheduling method is to run a group of threads sharing the working set simultaneously on different Cores of CMPs. To recognize the locality of references, they analyze the programs beforehand and statically schedule threads. In case of ITAS, the scheduler assumes the locality of references between sibling threads and dynamically aggregates them.

ITAS works only, when we execute multi-threaded programs. The rationale of the approach is that many modern programs are getting multi-threaded as CMP platforms widely spread. For example, Parsec benchmark (Bienia *et al*., 2008) is a multi-threaded benchmark to simulate emerging workloads and is intended to run on CMP platforms. Besides, many languages such as Java,

Perl, Ruby, Python, Erlang and compilers such as Open MP, MPI, Open 64 now support the development of programs using multiple kernel-level threads. We can also apply ITAS to running multiple hosted virtual machines like VM ware Server. In VMware Server, multiple sibling threads are created per one guest OS so that a guest OS can utilize multiple physical Cores. Thus, we expect that we will have more multi-threaded programs and more chances to apply ITAS.

Inter-core time aggregation scheduler: In this study, the outline of ITAS. We implement ITAS by modifying CFS in Linux 2.6.24. The implementation of ITAS is the extension of TAS (Yamada and Kusakabe, 2008a, b).

Completely fair scheduler: CFS is designed to accomplish fair usage of CPU between threads, which start at the same time with the same static priority. CFS counts CPU time consumed by each thread in nanoseconds to calculate the priority as vruntime. When a thread yields CPU, the additive vruntime is calculated from the CPU time and added to the vruntime of the thread. When, the thread gets ready, the thread is enqueued again with new vruntime. CFS sets higher priority for threads with less vruntime to equalize the amount of CPU consumption. The runqueue exists per Core and an independent scheduler works on each Core. CFS does not recognize the memory address space of each thread in scheduling.

Overview of time aggregation scheduler: TAS aggregates sibling threads and tries to execute them in sequence on a single Core. The basic idea of the implementation of TAS is to dynamically give a priority bonus to the sibling thread of the currently executed thread. As we mentioned, the priority of a thread is higher, when the vruntime of the thread is smaller. Therefore, the priority bonus for TAS works to reduce the vruntime of the sibling thread. The problem of fairness between threads or starvation is likely to happen by aggregating multi-threaded programs. Although, we can restrict the level of aggregation by tuning the priority bonus, it is hard for a user to specify the value because vruntime is too fine-grained. Therefore, we add a counter, agg_count, in each memory address space and each Core, which is used to count the number of the aggregation of the sibling threads per Core. TAS uses agg_count to control the aggregation. We show an example case of TAS (Fig. 1). Figure 1 shows the runqueue of CFS1. The circles (Fig. 1) represent threads. The number in a thread shows the vruntime of each thread. Threads are queued in the ascending order of vruntime in the runqueue. The links forming the queue are solid lines (Fig. 1). The pattern inside a thread represents the memory address space.

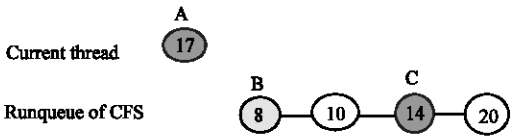

Fig. 1: Example case of TAS. A circle represents a thread and the pattern inside the circle expresses its memory address space. If there exists a sibling thread (thread C), TAS considers the thread as the candidate for the next thread

The currently executed thread A has been dequeued from the runqueue. After executing thread A, CFS selects thread B as the next thread. On the other hand, TAS considers thread C as another candidate. We set agg_bonus, the priority bonus for aggregation and agg_limit, the limit number of aggregation, in advance. TAS follows the processes below to schedule the next thread (Vrun_B means the vruntime of thread B).

TP1: The scheduler refers to agg_count of the memory address space of the current thread (thread A). If Eq. 1 is true, the scheduler goes to TP2. Otherwise, the scheduler chooses the thread with highest priority (thread B) as the next thread, sets agg_count of the memory address space of thread A as 0 and goes to TP4.

$$agg\_count<agg\_limit \quad (1)$$

TP2: If Eq. 2 is true, the scheduler chooses the sibling thread (thread C) of thread A as the next thread and goes to TP3. Otherwise, the scheduler chooses thread B as the next thread, sets agg_count of the memory address space of thread A as 0 and goes to TP4.

$$Vrun\_B>Vrun\_C - agg\_bonus \quad (2)$$

TP3: The scheduler increments agg_count in the memory address space of thread A and goes to TP4.

TP4: The scheduler finishes scheduling.

In case of (Fig. 1), if we set the agg_bonus equal to or >7, TAS will select thread C as the next thread. Otherwise, TAS selects thread B. We can change the value of agg_bonus and agg_limit manually using a system call we have implemented.

Overview of inter-core time aggregation scheduler: We extend TAS to implement ITAS. First, we run independent TAS per Core and assigns the role of master or slave to each Core. When, the scheduler on a master Core finds a chance of Time Aggregation, it sets a pointer, ia_mm, to the memory address space of the currently executed thread. Otherwise, ia_mm is NULL. Only a master Core can manipulate ia_mm and slave Cores can only refer to ia_mm. When ia_mm is set to an actual memory address space by a master Core, the schedulers on slave Cores look for the sibling threads sharing the memory address space, which ia_mm points to, from their own runqueue. If there exist sibling threads, the schedulers on slave Cores consider the threads as the candidates for the next threads with the priority bonus. We show an example case of ITAS on a dual-core processor (Fig. 2) and describe the scheduling processes in detail.

IP1: The scheduler on each Core looks for the candidate thread of Time Aggregation (thread C and G). If running on the master Core, the scheduler goes to IP2. If running on slave Core, the scheduler looks for the candidate thread of Inter-Core Aggregation (thread H) and goes to IP6.

IP2: If thread C exists, the scheduler refers to agg_count from the memory address space of currently running thread (thread A) and goes to IP3. Otherwise, the scheduler goes to IP5.

IP3: If both Eq. 1 and 2 are true, the scheduler chooses thread C as the next thread, increments agg_count of the memory address space of thread E and goes to IP4. Otherwise, the scheduler goes to IP5.

IP4: The scheduler sets the address of memory address space of thread C to ia_mm and goes to IP7.

IP5: The scheduler chooses a thread with the highest priority (thread B) as the next thread, sets agg_count of the memory address space of thread E as 0, sets ia_mm as NULL and goes to IP7.

IP6: If both thread G and thread H exist, the scheduler goes to IP6.1. If only thread G exists and thread H does not, the scheduler goes to IP6.2. If only thread H exists and thread G does not, the scheduler goes to IP6.3. If either thread G or thread H does not exist, the scheduler chooses the thread with the highest priority (thread F) as the next thread and goes to IP7.

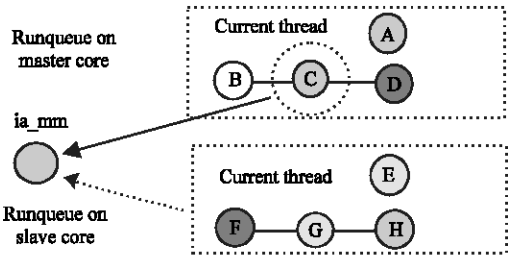

Fig. 2: Example case of ITAS. When sibling threads (circles with the same pattern) are aggregated in the master Core by TAS, the memory address space of the sibling thread is registered in ia_mm. The scheduler on lave Cores looks for threads sharing the same memory address space with ia_mm and considers the thread as the candidate for the next thread

IP6.1: If Eq. 3 and 4 are both true, the scheduler chooses thread H as the next thread, sets agg_count of the memory address space of thread E as 0 and goes to IP7. Otherwise, the scheduler goes to IP6.2.

$$Vrun\_F > Vrun\_H - agg\_bonus \quad (3)$$

$$Vrun\_G > Vrun\_H - agg\_bonus \quad (4)$$

IP6.2: The scheduler refers to agg_count of thread E. If both Eq. 1 and 5 are true, the scheduler chooses thread G as the next thread, increments agg_bonus of the memory address space of thread E and goes to IP7. Otherwise, the scheduler chooses thread F as the next thread, sets agg_count as 0 and goes to IP7.

$$Vrun\_F > Vrun\_G - agg\_bonus \quad (5)$$

IP6.3: If Eq. 4 is true, the scheduler chooses thread H as the next thread. Otherwise, the scheduler chooses thread F as the next thread. After choosing the next thread, the scheduler sets agg_count of the memory address space of thread E as 0 and goes to IP7.

IP7: The scheduler finishes scheduling. We implement ITAS to enable multiple master, slave Cores and ia_mm to exist. In this study, we use quad-core processors for the evaluation and assign the master to Core 0 and slave to other Cores with single ia_mm.

When we use many-core platforms, it is possible to change the allocation of master and slave by a system call we have implemented. We have also implemented another system call to change agg_bonus and agg_limit.

Preliminary evaluation of ITAS: In this study, we conduct a preliminary evaluation to show the relationship between the effect of ITAS and the locality of references between sibling threads.

We use a memory intensive workload because the effect of ITAS is related to the memory usage of each thread. The specification of the experimental platform is shown in Table 1. We use memory program of SysBench (2009). SysBench (2009) comprises several programs and each program investigates the performance of the specific factor in Online Transaction Processing (OLTP) workloads. We show the outline of memory program (Fig. 3). In memory program, a specified number of threads are created and each thread repeats sequential access to a specified amount (block_size) of memory area. Threads from a single invocation of memory program share a single memory address space. One memory access is expressed as an arrow (Fig. 3). Sibling threads of memory program repeat memory accesses until the total access size of every thread exceeds a specified size (total_size). We can specify the types of memory access to a specified amount (block_size) of working set. The type is read, reading a value from a specified address, or write, writing a value to a specified address. We can choose if each thread accesses to the same working set (global) or independent working set (local). In case of global mode, the size of working set of a single memory program is block_size. In case of local mode, the size of working set of a memory program is the product of block_size and the number of threads.

To clarify the effect of relationship between ITAS and block_size, we modify memory program to let all sibling threads yield CPU after accessing block_size of data. Default memory program lets sibling threads iterate the memory access until it expires the quantum time. Therefore, the memory access to the working set from the second iteration is likely to hit the data on the cache loaded by the first iteration. The modification stops the iteration of memory access and clarifies the effect of ITAS to utilize the locality of references between sibling threads. We investigate the effect of ITAS in both global and local mode. We do not focus on read access because hardware-prefetching of cache works efficiently with memory program and the effect of ITAS is not so clear as write access. We fix total_size as 1000GB, the number of threads as 100 and the number of memory program as 10. On the other hand, we change block_size from 16-16 MB to investigate the relation of the effect of ITAS and the size of the working set. We show the result, when we set agg_bonus as 50 millions, which is high enough to see the maximum effect of ITAS based on the measurements. We also set agg_limit as 100 based on the number of threads.

Table 1: Specification of the experimental platform

Processor	Intel Core 2 Quad 2.66 GHz
L2 caches size/Latency	4 MB/14 cycles
Memory size/Latency	1 GB/189 cycles
Operating system/Kernel	Fedora 7/Linux 2.6.24

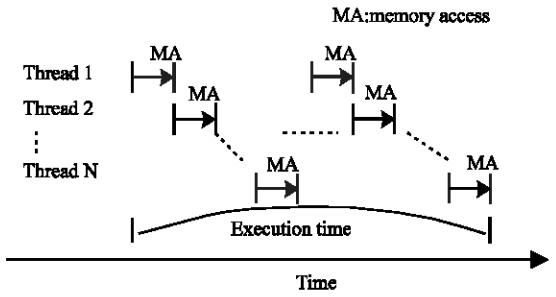

Fig. 3: Outline of memory program in SysBench (2009). Each arrow expresses one memory access. The memory access is repeated until the total access size of every thread exceeds a specified size. We evaluate the execution time to accomplish all the memory accesses

We show the result of the ratio of the execution time in both global and local mode (Fig. 4). We also show the result of the ratio of the L2 cache misses ([local,global]_L2) and the resource stalls ([local,global]_RS) in each access mode (Fig. 5).

In global mode, we see the larger reduction of the execution time, as well as L2 cache misses and resource stalls, especially when the size of access data is between 512 KB and 4 MB. We consider that the largest effect of ITAS is related to the L2 cache size of the platform. When block_size is <512 KB, the possibility that the previously loaded data is still left in the L2 cache increases also in CFS, therefore, we do not see the clear effect of ITAS. When block_size is larger than 4 MB, the memory access size is too large for a previously executed thread to leave data on caches, which the next thread accesses.

In local mode, we can expect the effect only from time aggregation of ITAS because there is no locality of references between sibling threads. In the experiment, if sibling threads of a memory address space are equally distributed among Cores, there are 25 sibling threads enqueued on each Core. As we set agg_limit as 100, a single thread can be executed 4 times before switching to threads of other memory address spaces if the assumption above is correct. However, block_size has to be small to utilize the loaded data on caches because the data of one thread can be purged by sibling threads during the aggregation. Figure 4 can see the largest effect, when block_size is 32 KB on the execution time, where the total

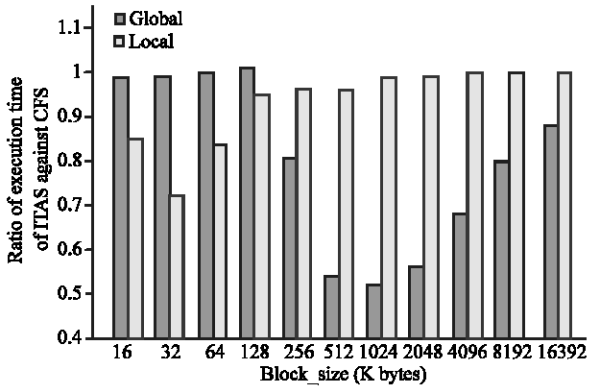

Fig. 4: The comparison of the execution time of ITAS against CFS in global and local mode. The effect of ITAS is related to the total size of the working set and the size of L2 cache

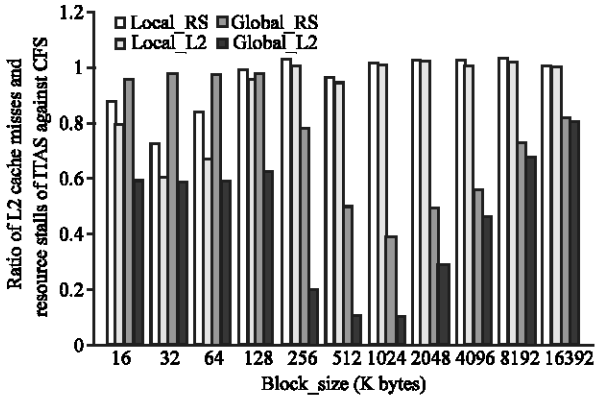

Fig. 5: The comparison of the L2 cache misses ([local,global]_L2) and the resource stalls ([local,global]_RS) of ITAS against CFS in global and local mode. The effect of ITAS is related to the total size of the working set and the size of L2 cache

size of the working set of one memory program is 3.2 MB. When we increase block_size >32 KB, the total size of the working set is larger than the capacity of the L2 cache and the effect of ITAS becomes smaller. We can also see the reduction of L2 cache misses and resource stalls only when block_size is small.

Thus, we show that the effect of ITAS depends on the locality of reference between sibling threads and the size of the working set. According to the previous researches on the analysis of commercial programs such as Web servers and database servers, sibling threads of programs in the real world have high possibility to share a certain amount of working set (Chishti *et al*., 2005: Ziemba and Upadhyaya, 2008). Thus, we expect that ITAS is widely effective in the real world.

Evaluation with multi-threaded database server: In this study, we show the evaluation of ITAS with a multithreaded database server. We have two purposes of the evaluation. The first purpose is to investigate how widely ITAS can be effective in executing a more realistic workload. We showed that the effect of ITAS is related to the characteristic of memory access of each thread and the capacity of caches in a memory intensive benchmark. In this study, we investigate the relationship between the characteristic of the memory access of database server and the effect of ITAS. The second purpose is to investigate the effect of ITAS on the non-aggregated background programs, which are concurrently executed with the multithreaded programs but are not aggregated by ITAS. Thus, we assume an execution of a multi-threaded database server with background batch programs and evaluate the influence on those programs by measuring its throughput. We explain OLTP program, a benchmark program to measure the throughput of a database server.

OLTP program of SysBench: We use OLTP program of SysBench (2009) as a client program for the evaluation of ITAS. The program issues dynamic requests to a database server and plays a role of a simple application server. We consider OLTP program is a proper tool to simulate realistic workloads and investigate the performance of thread scheduler because it is known to trigger the discussion of the relationship between the performance and the thread scheduler on CMP platforms (Linux-Kernel Archive, 2009). We change the database sizes by specifying the number of lines and the size of a line of the database is 190B. If we use a database with 50 thousands lines, the size of the database is about 9.5 MB. We have various options to control workloads of OLTP program. For example, we can generate multiple client threads so as to simulate simultaneous accesses from multiple clients. Other than client threads, we can specify the size of database, the database engines, the instruction sets of each request and so on.

The specification of the experimental platform is shown in Table 1. In Table 2, we show the options specified for OLTP program. We use InnoDB of MySQL for the database engine. MySQL creates one server thread per one client connection and each server thread shares the same memory address space. Therefore, ITAS is applicable in executing OLTP workloads of MySQL. InnoDB supports transactional executions and is the default database engine in OLTP program. We test 8 database sizes by changing the number of lines. We select the database sizes around 10,000 lines, which are the default value in the OLTP program. To simulate the

Table 2: Options specified for OLTP program in SysBench (2009)

Database engine	MySQL InnoDB
Database size (thousand lines)	1, 5, 10, 20, 30, 40, 50
Number of threads	100
Test mode	Advanced
Access point distribution	Uniform, gaussian

accesses from multiple clients, we specify 100 client threads. Each client thread repeats a set of transactions registered in the advanced mode, which is the default test mode and contains 9 kinds of transactions to simulate real workloads. We can specify the access strides by controlling the access points of each thread to follow a specific distribution. For example, if we select the uniform distribution, each thread accesses uniformly within the range of the specified database size. We test both the uniform and the gaussian distribution. We let the threads to repeat the transactions until the total counts of the transactions exceed 10,000 times, which is the default value in OLTP program. We measure the execution time and the CPU time of oltp program. During the execution, the database threads dominate >99% of the execution time. We investigate the effect on the database server by comparing the execution time. On the other hand, we investigate the effect on the client threads of OLTP program by measuring its CPU time. We also count the L2 cache misses and the resource stalls during the execution by the performance monitoring counters of the processor. To investigate the effect of ITAS on the throughput of the non-aggregated background programs, we run single-thread loop programs with OLTP program. The loop programs only execute infinite loop touching a certain amount of memory and counts the number of the executed loops. In this study, we run 4 loop programs, where each program is bound to different Cores and touching 1 MB of memory area to fully consume L2 cache. We run the loop programs with GNU's time command to measure the CPU time of the loop programs during the execution of OLTP program. We calculate the throughput of the loop programs by dividing the number of the iterations by their CPU time during the execution of OLTP program. We show the result, when agg_bonus is 50 millions. When agg_bonus is 50 millions, ITAS aggregates >99% of chances of aggregation, therefore, we consider it is large enough. We also set agg_limit as 100, which is the number of client connections. We show the comparison of CFS to ITAS in executing multithreaded database workloads (Fig. 6). Figure 6 shows the order of the threads executed on the platform. Each circle expresses a thread and the pattern inside one circle shows its memory address space. As we mentioned, MySQL database server and OLTP program creates multiple sibling threads. The background threads include threads of the loop program. As shown in Fig. 6, ITAS aggregates more sibling threads simultaneously on different Cores. The scheduling order of the background programs tends to be postponed by ITAS.

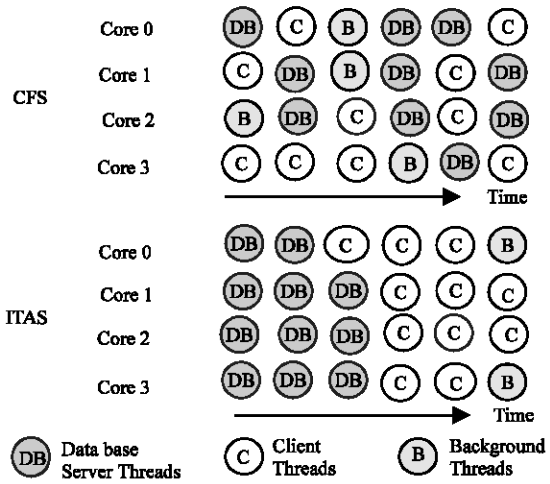

Fig. 6: Comparison of thread selection in CFS and ITAS in executing multi-threaded database workloads. ITAS tries to execute more sibling threads (circles with the same pattern) simultaneously on each Core

RESULTS AND DISCUSSION

Influence on OLTP program: We compare the execution time ([uniform,gaussian]_DB) and CPU time of OLTP program ([uniform,gaussian]_CLI) between ITAS and CFS (Fig. 7). We show the ratios of the execution time and CPU time in ITAS against CFS in each access distribution.

As shown in Fig. 7, ITAS reduces the execution time in most of the parameters and we consider that ITAS is effective in executing database server threads. The effect on the execution time on the uniform distribution becomes small, when the database size is 30,000 lines or more. We consider the effect on the execution time is related with the increase of the number of the L2 cache misses and the resource stalls shown in Fig. 8.

When we increase the database size >20,000 lines, the number of the L2 cache misses and the resource stalls start to increase. The database size of 30,000 lines is about 5.7 MB, which is 1.7 MB larger than the size of the L2 cache on our platform. Thus, it is reasonable to see the increase of the L2 cache misses and resource stalls, when we set the database size as 30,000 lines or more in the uniform distribution. As we can see, the effect on the reduction of the L2 cache misses, the resource stalls and the execution time by ITAS are relatively large, when the database size is 20,000 lines and less. On the

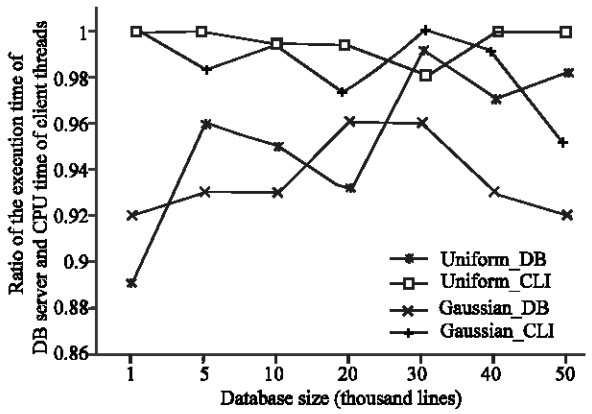

Fig. 7: Comparison of execution time of OLTP program ([uniform,gaussian]_DB) and CPU time ([uniform, gaussian]_CLI) of client programs between CFS and ITAS in each database size and distribution of access stride. We see the reduction of the execution time of DB server threads. We also see slight reduction of CPU time in client threads

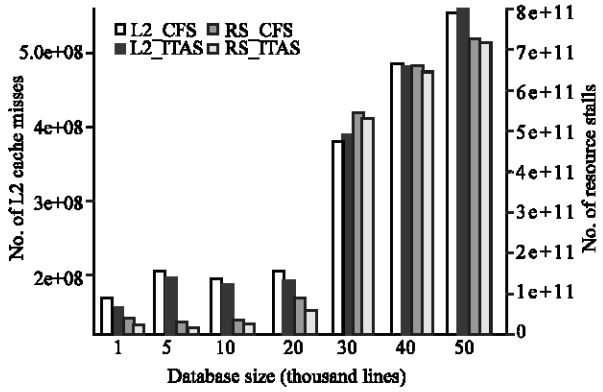

Fig. 8: Comparison of the L2 cache misses and the resource stalls in each database size in uniform distribution. We see the reduction of the L2 cache misses and the resource stalls, whens the size of the database server is 20,000 lines and below

other hand, the effect on the reduction of the L2 cache misses and the execution time is little, when the database size is 30,000 lines or more. In case that the database size is 30,000 lines or more, the possibility that each thread shares the working set is small. Therefore, we consider the effect of ITAS gets small in larger database sizes in the uniform distribution.

In the gaussian distribution, we see the reduction of the execution time in every parameter. As shown in Fig. 9, ITAS also reduces the L2 cache misses and the resource stalls. In the gaussian distribution, the access

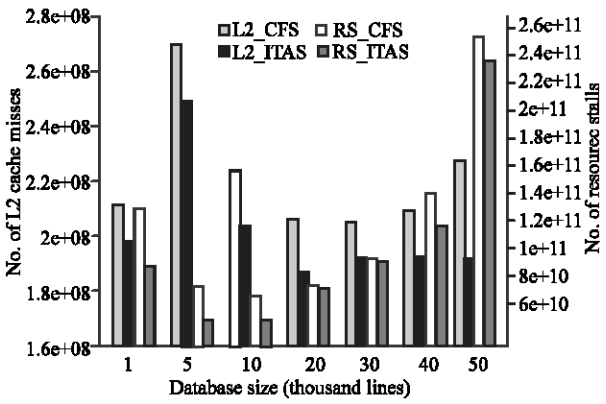

Fig. 9: Comparison of the L2 cache misses and the resource stalls in each database size in gaussian distribution. We see the reduction of the L2 cache misses and the resource stalls in every database size

point is biased and locality of the reference is likely to emerge. Thus, the number of the L2 cache misses and the resource stalls are smaller compared to those in the uniform distribution. The reduction of the CPU time of OLTP program is at most 5% as shown in Fig. 7. In OLTP program, each client thread just repeats issuing requests to the database server. The working set of client thread itself is small and each thread does not share the large working set. Although, ITAS does not degrade the performance of the client threads, its effect is small.

Influence on the background programs: This study demonstrates the effect of ITAS on the loop program. Figure 10 shows the ratio of the throughput of the loop program in ITAS to the throughput in CFS. As shown in Fig. 10, the enhancement of the throughput of the loop program in most of the parameters. We consider that the enhancement of the throughput comes from the characteristics of CFS. CFS tries to equalize the cumulative CPU time of threads, which start at the same time with the same nice value. If one thread spends less CPU time than other threads, CFS will give longer quantum time to the thread. We did not modify this characteristic of CFS in implementing ITAS. We consider that the loop program obtains longer quantum time instead of obtaining smaller quantum time frequently because ITAS aggregates sibling threads and lets the loop program wait longer than CFS. We consider acquiring the longer quantum time results in the decrease of context switches and the enhancement of the throughput of the loop program. Thus, even if ITAS aggregates multithreaded programs, ITAS does not

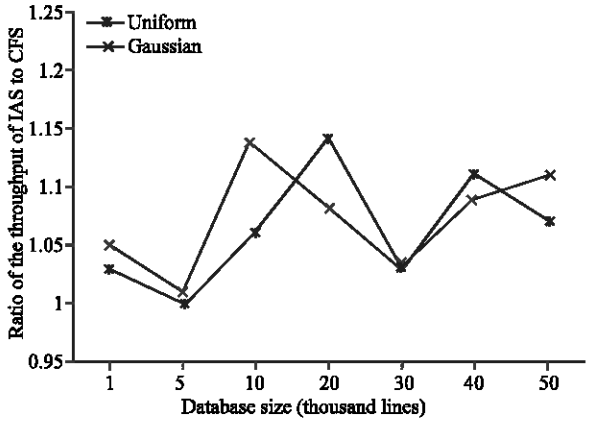

Fig. 10: Effect on the throughput of the loop program. Even though the loop program in not multithreaded program, the throughput of the loop program increases in most of the parameters

disturb the throughput of the background programs. In this study, we show that the effect of ITAS on a database server workload is influenced by the access distribution of client threads and the size of the database. To actually utilize ITAS in the real world, it is necessary to tune the priority bonus and the limitation of the aggregation to enhance the throughput of the system and keep the fairness between threads in a preferable level. For example, if sibling threads share working set as large as the size of the L2 cache, we can expect the enhancement of the total performance by setting agg_bonus high. On the other hand, if sibling threads have heavily I/O intensive workload, the ratio of CPU usage may fall and ITAS may degrade the total performance. Manual tuning of agg_bonus is difficult. To settle this issue, we are now developing API and a helper thread mechanism to automatically tune agg_bonus according to the programs to run.

CONCLUSION

In this study, we investigate the effect of ITAS on a multithreaded database server workload. ITAS is a kernel-level thread scheduler for commodity CMP platforms. ITAS executes sibling threads, kernel-level threads sharing the same memory address space, simultaneously on different Cores to utilize locality of reference between threads and reduce cache misses. We investigate the effect of ITAS on a multithreaded database server workload running with non-multithreaded background batch programs on a commodity CMP platform. We show the experimental result, which indicates that ITAS enhances the performance of the database server without degrading that of non-aggregated background programs. We consider that the result is promising and applicable to wider situations because nowadays it is common to run multiple-multi-threaded programs concurrently. The future research includes the measurement in using multiple virtual machines. In some virtual machines like VMware Server, each thread in guest OS is mapped to multiple kernel-level threads in host OS. In case of Linux host OS, one virtual machine is run by multiple sibling threads. Therefore, we can expect the effect of ITAS in running multiple virtual machines. We also investigate both static and automatic ways of tuning the priority bonuses to enhance the effect of ITAS. We now implement API for developers to statically specify the strength of aggregation based on the characteristic of programs. We also use helper threads to monitor the aggregation of sibling threads inside the kernel and dynamically tune the priority bonuses.

REFERENCES

Bienia, C., S. Kumar, J.P. Singh and K. Li, 2008. The parsec benchmark suite. Characterization and Architectural Implications in Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp: 72-81.

Chandra, D., F. Guo, S. Kim and Y. Solihin, 2005. Predicting inter-thread cache contention on a chip multi-processor architecture. In: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, pp: 340-351.

Chen, S., P.B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G.E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T.C. Mowry and C. Wilkerson, 2007. Scheduling threads for constructive cache sharing on Cmps. In: Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures, pp: 105-115.

Chishti, Z., M.D. Powell and T.N. Vijaykumar, 2005. Optimizing replication, communication and capacity allocation in Cmps. In: Proceedings of the Thirty-Second International Symposium on Computer Architecture, pp: 357-368.

Fedorova, A., M. Seltzer, C. Small and D. Nussbaum, 2005. Performance of multithreaded chip multiprocessors and implications for operating system design. In: USENIX Annual Technical Conference.

Feitelson, D.G. and L. Rudolph, 1992. Gang scheduling performance benefits for fine-grain synchronization. J. Parallel Distributed Comput., 16: 306-318.

Kundu, P., M. Annavaram, T. Diep and J. Shen, 2004. A case for shared instruction cache on chip multiprocessors running OLTP. In ACM SIGARCH Comput. Arch. News., 32: 11-18.

Linux-Kernel Archive, 2009. SMP performance degradatioin with SysBench. http://www.ussg.iu.edu/hypermail/linux/kernel/0702.3/0299.html.

Ogawa, S. and K. Hiraki, 2005. A speedup technique with scheduler using process execution information. ACS 11, 46 (4): 161-169.

SysBench, 2009. A system performance benchmark. http://sysbench.sourceforge.net.

Snavely, A., D.M. Tullsen and G. Voelker, 2002. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In: Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pp: 66-76.

Yamada, S. and S. Kusakabe, 2008a. Development of a thread scheduler for global aggregation of sibling threads. Res. Rep. Inform. Sci. Elect. Eng. Kyushu Univ., 1: 69-74.

Yamada, S. and S. Kusakabe, 2008b. Effect of context aware scheduler on tlb. In: Proceedings of Workshop on Multi-Threaded Architectures and Applications (Proceedings of IEEE International Parallel and Distributed Processing Symposium), Published in CD.

Ziemba, V.S.P.S. and G. Upadhyaya, 2008. Analyzing the effectiveness of multicore scheduling using performance counters. In: Workshop on Interaction Between Operating Systems and Computer Architecture.