Proposal and Evaluation of APIs for Utilizing Inter-Core Time Aggregation Scheduler

Satoshi Yamada and Shigeru Kusakabe

Graduate School of Information Science and Electrical Engineering,
Kyushu University, 744, Motooka, Nishi-ku, Fukuoka, Japan
satoshi@ale.csce.kyushu-u.ac.jp,
kusakabe@ait.kyushu-u.ac.jp

Abstract. This paper proposes and evaluates APIs for Inter-Core Time Aggregation Scheduler (IAS). IAS is a kernel-level thread scheduler to enhance performance of multi-threaded programs on multi-core processors. IAS combines time-multiplexing and space-multiplexing scheduling to utilize caches existing per processing core and shared between processing cores.

We present the effect of APIs in two aspects. Firstly, we show that we can effectively and easily set the aggregation strength in IAS based on the quantum time. Secondly, we show that we can gain the effect of space-multiplexing without setting processor affinity of each thread by grouping processing cores and running IAS per group. We implement IAS and its APIs by modifying a Linux kernel and present its effect on a commodity multi-core processor.

Keywords: Thread Scheduling, Multi-core Processor, Cache Sharing, Multi-threaded Program.

1 Introduction

In this paper, we show the proposal and the evaluation of APIs for Inter-Core Time Aggregation Scheduler (IAS). IAS is a kernel-level thread scheduler to enhance the performance of multi-threaded programs on a commodity multi-core processor. IAS combines time-multiplexing and space-multiplexing scheduler to utilize the caches existing per processing core (Core) and shared between Cores. The contributions of this paper is as follows:

- We show that we can effectively and easily set the aggregation strength in IAS based on the quantum time, which is a period of time that the thread uses CPU.
- We show that we can gain the effect of space-multiplexing without setting the processor affinity of each thread by grouping Cores and running IAS per group.

Nowadays, we have several kinds of multi-core processors, such as Simultaneous Multi-Threading (SMT), Chip Multi-Processing (CMP), and Chip

E. Frachtenberg and U. Schwiegelshohn (Eds.): JSSPP 2010, LNCS 6253, pp. 191–207, 2010.

Multi-Threading (CMT), where we can execute threads in parallel. One of the main differences between multi-core processors and conventional shared-memory multi-processors is that caches, typically L2 caches, are generally shared by Cores in multi-core processors. It is widely known that combinations of threads running simultaneously on different Cores affect the utilization of caches and the performance in a multi-core processor because Cores compete the shared cache with each other[1,2,3]. To utilize the shared cache in a multi-core processor, we propose a thread scheduling mechanism which focuses on multi-threaded programs.

In this paper, a multi-threaded program means a program which executes multiple kernel-level threads sharing the same memory address space in parallel. In Linux, for example, we can implement multi-threaded programs with POSIX library, Java, Perl, MPI, OpenMP, and Open64 because a thread in these languages and compilers systems corresponds to a native thread in the kernel. The rationale of focusing on only multi-threaded programs is that many modern programs, especially commercial programs, are getting multi-threaded as multi-core processors widely spread. For example, database servers and Web servers, such as MySQL and Apache HTTP Server, are multi-threaded to handle multiple client connections efficiently. The modern benchmark programs such as DaCapo Benchmarks[4] and Parsec Benchmark[5] also employ multi-threading to simulate popular and emerging workloads. We expect that we will have more multi-threaded programs and more chances to apply our scheduling mechanism.

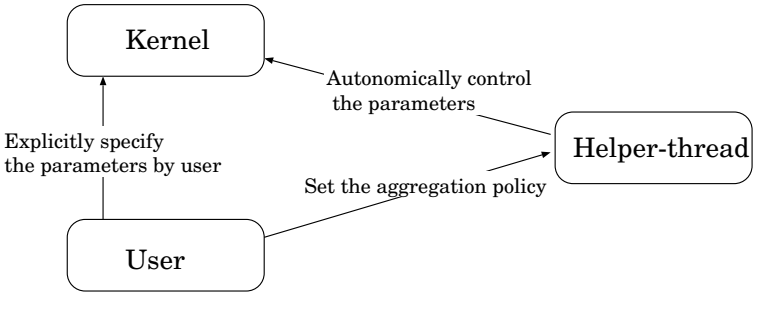

Fig. 1. The overview of the scheduling mechanism. We divide the functions related to scheduling into three domains, which enables the dynamic and flexible control of thread aggregation. In this paper, we focus on User domain.

We show the overview of our scheduling mechanism in Fig. 1. The scheduling mechanism is made up of three domains, Kernel, User, and Helper-thread. Kernel domain provides a basic scheduling mechanism and implemented as IAS. User and Helper-thread are domains which control the parameters for Kernel domain. User domain provides the interfaces to control the parameters explicitly assuming that users are aware of the characteristics of the workloads. Helper-thread domain analyzes the characteristics of the currently executed workloads, detect the degradation of the performance of multi-threaded programs, and controls

the parameters autonomically. Thus, our scheduling mechanism can be applied to the characteristics of the workloads without modifying and re-building Kernel. In this paper, we focus on User domain, and present the APIs to control the behavior of IAS. The detailed design and implementation of Helper-thread domain is our future work.

IAS is a kernel-level thread scheduler for commodity platforms with multi-core processors and implemented by modifying the scheduler of Linux kernel. IAS dynamically aggregates sibling threads, kernel-level threads sharing the same memory address space, and executes them simultaneously on different Cores based on the assumption that sibling threads share a certain amount of working set, the memory area to be accessed by threads. The benefit of IAS is to increase the possibility that co-scheduled threads share their working set and decrease the capacity pressure on the cache. IAS may increase the simultaneous access to the working set, where only transactional access is permitted with locks and semaphores, and cause frequent stalls. However, according to the researches on the analysis of the performance of CMP, the L2 cache misses caused by the insufficiency of capacity are the most influential[3,6]. Therefore, we expect the enhancement of the performance by IAS.

Previously, we investigated the effect of IAS with several multi-threaded benchmark programs and clarified two problems for the effective use of IAS[7,8]. The first problem is the aggregation strength. The effect of IAS depends on the characteristic of programs and platforms such as the size of shared working set between sibling threads of the programs and that of the shared cache size of the platforms. In case sibling threads share a working set, strong aggregations of sibling threads are likely to enhance the performance. On the other hand, IAS can degrade the performance when the workload is I/O intensive and the aggregation of sibling threads results in poor utilization of CPU. For this reason, we should control if we aggregate sibling threads of a program or not, and the aggregation strength. The second problem is the groups of Cores to execute IAS. IAS aggregates sibling threads on the group of Cores specified in the kernel. Previously, we have evaluated the effect of IAS on a dual-core processor. In the dual-core processor, we can make only a single group of Cores. Nowadays, the number of Cores has increased and the structure of the memory hierarchy tends to become complex like Intel Core i7. In such platforms, aggregations of sibling threads with a single group of Cores may increase the overhead of communications between Cores because we assume that sibling threads share a certain amount of working set. Setting processor affinities and assigning Cores to every different program like a conventional space-multiplexing may decrease the communication between Cores. However, it is another difficult issue to optimally set the processor affinity of each thread. We consider that setting multiple groups of Cores to run IAS can reduce the overhead of communications between Cores and we should have an interface to control the groups.

In this paper, we propose and evaluate APIs for IAS to settle the problems mentioned above. We show that we can effectively and easily set the aggregation strength in IAS based on the quantum time of the previously executed thread.

We also show that we can gain the effect of space-multiplexing without setting the processor affinity of each thread by splitting Cores into several groups and running IAS per group.

The rest of this paper is organized as follows. Section 2 explains the implementation and preliminary evaluation of IAS. Section 3 explains the proposal of APIs. Section 4 presents the evaluation of the effectiveness of APIs. Section 5 introduces related works and clarifies our research position. We conclude in Section 6.

2 Implementation and Evaluation of Inter-Core Time Aggregation Scheduler (IAS)

In this Section, we explain the implementation and the evaluation of IAS. We implement IAS by modifying Completely Fair Scheduler (CFS) in Linux 2.6.24 because we assume the use of IAS on commodity processors. IAS ignores the inversion of the priority of each thread in SCHED_NORMAL class, which is non-real-time thread in Linux, and dynamically aggregates sibling threads. We explain the scheduling mechanism of CFS for threads of SCHED_NORMAL class in Section 2.1 and IAS in 2.2. In Section 2.3, we show the preliminary evaluation of IAS on a commodity processor. Based on the preliminary evaluation, we show the problems of running IAS and necessity of effective APIs.

2.1 Completely Fair Scheduler (CFS)

CFS is the standard thread scheduler employed in Linux since its version 2.6.23. CFS is designed to equally distribute CPU time to threads with the same static priority. CFS counts the quantum time of each thread in nanoseconds and calculates the priority as *vruntime* based on the quantum time and `nice` value. When a thread is dispatched by the scheduler, the additional *vruntime* value is calculated from the quantum time and added to the current *vruntime* of the thread. CFS sets higher priority for threads with less *vruntime* to accomplish the fair usage of CPU between threads which start at the same time with the same `nice` value. The runqueues and independent schedulers exist per Core. The load balancer in CFS equalizes the sum of `weight`, which is a value corresponding to `nice` value and defined in the kernel, between runqueues. CFS does not recognize the memory address space of each thread both in scheduling and load balancing.

2.2 Overview of Inter-Core Time Aggregation Scheduler (IAS)

IAS implements two scheduling policies at the same time. The first scheduling policy is the time aggregation, which executes sibling threads in a row on a single Core. The second scheduling policy is the inter-core aggregation, which simultaneously executes sibling threads on different Cores. In this section, we firstly explain Time Aggregation Scheduler (TAS), which is the implementation of the time aggregation. Then, we explain the extension of TAS to adopt the inter-core aggregation.

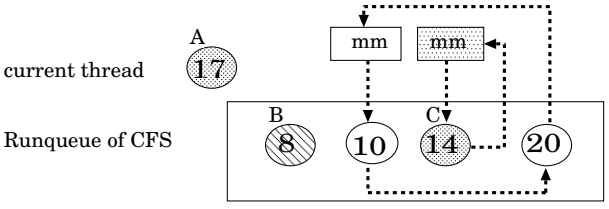

Fig. 2. Example case of TAS. A circle represents a thread and the pattern inside the circle expresses its memory address space. TAS looks for the sibling thread of the current thread from the list of the sibling threads. If there exists a sibling thread (thread C in this case), TAS considers the thread as the candidate for the next thread.

Implementation of Time Aggregation Scheduler (TAS). 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 in Section 2.1, the priority of a thread is higher when *vruntime* of the thread is smaller. Therefore, the priority bonus for TAS works to reduce *vruntime* of the sibling thread. To implement this idea in CFS, we add a flag to `task_struct`, the structure to maintain the states of a thread in Linux, to recognize if the thread has the sibling threads or not. When a thread creates its sibling thread, TAS sets the flag in `task_struct` and inserts the thread into the list of its sibling threads. The list of the sibling threads exists per Core and sorted in the ascending order of *vruntime*. We show an example case of the time aggregation in Fig. 2.

Fig. 2 shows the runqueue of CFS[1] and the additional links of sibling threads for the time aggregation. The circles in Fig. 2 represent threads and the rectangle containing threads represents a runqueue. The number in a thread shows *vruntime* of the thread. Each thread in Fig. 2 owns *vruntime* of around 10 to 20 for ease of explanation, however, it is common for threads in Linux to own *vruntime* in the millions and the billions calculated from their quantum time counted in nano seconds. Threads are queued in the ascending order of *vruntime* and shown from the left in the runqueue in Fig. 2. The patterns inside the threads represent the memory address spaces. Our scheduler links the sibling threads in the ascending order of their *vruntime*. The links between sibling threads are dashed lines in Fig. 2. We add a member, `mm_sibling`, to the structure of the memory address space, `mm_struct`, in Linux. The links of the sibling threads begin with `mm_sibling` (represented as `mm` in Fig. 2). The currently executed thread A has been dequeued from the runqueue. After executing thread A, CFS selects thread B as the next thread. TAS checks if the flag for the sibling threads is set in thread A. TAS recognizes that the flag is set and looks for the sibling thread from the list of the sibling thread starting from the `mm_sibling` of thread A. TAS finds thread C from the list and considers thread C as another candidate.

[1] The runqueue of CFS has a structure of Red-black tree. We express the runqueue as a list for ease of explanation.

We set the priority bonus for aggregating sibling threads in advance and our scheduler evaluates the expression below.

$$B.vruntime > C.vruntime - priority\_bonus \quad (1)$$

In this paper, we express *vruntime* of a thread as *thread_ID.vruntime* like *B.vruntime*. If expression (1) is true, then TAS will select thread C. If we set the priority bonus equal to or larger than 7 in Fig. 2, TAS will select thread C as the next thread. Otherwise, TAS selects thread B. Thus, TAS is able to aggregate sibling threads while considering the priority of each thread. Also, the scheduling algorithm of TAS is $O(1)$ because the link of sibling threads is sorted in ascending order of *vruntime*.

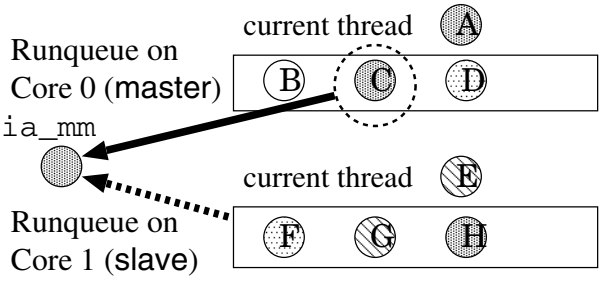

Fig. 3. Example case of IAS. When sibling threads (circles with the same pattern) are aggregated in Core 0 by TAS, the memory address space of the sibling thread is registered in `ia_mm`. The scheduler on Core 1 recognizes for threads sharing the same memory address space by looking at `ia_mm` and considers the thread as the candidate for the next thread.

Extension of TAS to add the inter-core aggregation. We extend TAS to adopt the inter-core aggregation to implement IAS. First of all, we run independent TAS per Core and assign each Core a role of `master` or `slave` Core. IAS lets every Core cooperatively aggregate sibling threads by making `slave` Cores follow the aggregation on `master` Core. When the scheduler on `master` Core finds a chance of aggregating sibling threads, it sets a pointer, `ia_mm`, to the memory address space of the currently executed thread. Otherwise, `ia_mm` is NULL. Only `master` Core can manipulate `ia_mm` while `slave` Cores only refer to `ia_mm`. When `ia_mm` is set to an actual memory address space, the schedulers on `slave` Cores look for the sibling threads sharing the memory address space, which `ia_mm` points to, in their own runqueue. If there exists sibling threads, the schedulers consider the threads as the candidates for the next thread to be scheduled with the priority bonus.

We show an example case of IAS on a platform of a dual-core processor in Fig. 3. In Fig. 3, the circles represent threads and squares represent runqueues on each Core. The pattern inside the thread represents the memory address space and three threads are waiting in the runqueue on each Core. While we

omit *vruntime* values in Fig. 3, threads are enqueued into each runqueue in the ascending order of their *vruntime* from the left. Thread A is running on `master` Core and thread E is running on `slave` Core. We also omit the links between sibling threads in Fig.3. Thread B on `master` Core and thread F on `slave` Core are to be scheduled next to thread A and thread E in case of CFS. After executing thread A on `master` Core, thread C is also the candidate to be scheduled next in TAS because thread C is the sibling thread of thread A. If thread C satisfies expression (1), the scheduler on `master` Core sets the memory address space of thread C to `ia_mm` (solid arrow in Fig. 3). On `slave` Core, the scheduler checks `ia_mm` in scheduling (dashed arrow in Fig. 3). After executing thread E, thread F, G, and H are the candidates because thread G is a sibling thread of thread E and thread H is a sibling thread sharing the memory address space set in `ia_mm`. To execute sibling threads simultaneously on different Cores, IAS raises the priority of the thread sharing the memory address space, which `ia_mm` points to, with the priority bonus. Thread H has the priority bonus against thread F and thread G. If thread H satisfies both expression (2) and (3), thread H will be scheduled after thread E.

$$F.vruntime > H.vruntime - priority\_bonus \tag{2}$$

$$G.vruntime > H.vruntime - priority\_bonus \tag{3}$$

Following the steps above, IAS can execute sibling threads nearly simultaneously on different Cores while considering the priority of each thread. When thread H does not satisfy expression (2) and (3), IAS behaves as TAS. If expression (4) is satisfied, thread G will be the next thread. If expression (4) is not satisfied, thread F will be scheduled.

$$F.vruntime > G.vruntime - priority\_bonus \tag{4}$$

IAS uses the link of sibling threads, which we use for the time aggregation, to search for the sibling threads. The scheduling cost of IAS is also $O(1)$ because the sibling threads are sorted in ascending order in the link.

2.3 Preliminary Evaluation of Inter-Core Time Aggregation Scheduler (IAS)

In this section, we show the preliminary evaluation of IAS in terms of its overhead against CFS. We also show the effectiveness of IAS on RUBiS benchmark[9], which is a benchmark program to measure the performance of a Web application server running a multi-threaded HTTP server and a database server simultaneously. Firstly, we show that the overhead of IAS is small compared to CFS. Then, we show that the effect of IAS depends largely on the value of the priority bonuses[8], indicating that an easy and effective way of controlling the priority bonus is necessary.

Overhead of IAS. We evaluate the additional overhead of IAS compared to CFS. The following tasks are causes of the overhead of IAS.

- Setting the flag of sibling threads in the added member of `task_struct`
- Setting link between sibling threads in the runqueues
- Considering sibling threads in scheduling

We implement a benchmark, which measures the execution time of creating and joining multiple sibling threads, to evaluate the total additional overhead of IAS. The created sibling threads just join with the parent thread. Also, we set the priority bonus for IAS as 0 to schedule threads according to the priority of CFS. We compare the execution time in CFS and IAS and measure the sum of the listed overhead.

According to our measurements, we see the increase of the execution time in IAS by 1% in creating and joining 500K sibling threads. In case the aggregation of sibling threads degrades the performance, we only have to set the priority bonus as 0.

Table 1. Result of RUBiS benchmark

Kernel	CFS	IAS		
		1M*vruntime*	10M*vruntime*	100M*vruntime*
Completed Sessions	230	259 (1.12)	301 (1.30)	265 (1.15)
Response Time (ms)	62,556	48,760 (0.77)	43,090 (0.68)	53,230 (0.85)

Effect of IAS on RUBiS benchmark. We show the effect of IAS in running RUBiS benchmark in Table 1. RUBiS is a benchmark application which simulates the workload of ebay.com and evaluates the performance of a Web application consisting of a HTTP and a database server. RUBiS sends simultaneous requests from multiple clients to the Web application server and evaluates the throughput (Completed Sessions) and the average response time (Response Time) of each request. Both HTTP (Apache HTTP server 2.2.8) and database servers (MySQL 5.0.45) are multithreaded, therefore, IAS aggregates threads of both servers. We use RUBiS benchmark because each thread of these transaction-oriented applications is likely to share the working set rather than scientific application benchmarks[10,11]. We change the value of the priority bonus and compare the result with CFS. The numbers in the parentheses indicates the ratio of the result in IAS against CFS.

In Table 1, we see the increase of the throughput and the reduction of the response time in IAS compared to CFS, indicating IAS is effective in enhancing the performance of a Web application server. We also see that the effect varies as we change the priority bonus and we have to set the priority bonus around 10 millions to maximize the effect. When we set the priority bonus as high as 100 millions *vruntime*, IAS aggregates too many sibling threads of one server and let the sibling threads of another server wait too long. The result shows that we have to tune the priority bonus to accomplish the better performance in running multiple multi-threaded programs.

2.4 Problems of IAS

Based on the preliminary evaluation in Section 2.3, we consider two problems in running IAS as shown below.

- Control of the priority bonus
- Allocation of `master/slave` role

Firstly, the effectiveness of IAS depends on the characteristic of each program. In case IAS degrades the total performance by the aggregation of some programs, users should have an interface to set the priority bonus as 0 or tell the kernel not to aggregate the sibling threads of those programs. Even when IAS enhances the total performance by aggregating the sibling threads of some programs, the priority bonus should be given in proper strength to maintain some degree of fairness of CPU usage between threads. Assuming users are aware of the characteristics of each program in advance, it is still difficult to properly give the priority bonus in *vruntime*. As we mention in Section 2.1, *vruntime* is calculated in the order of nano seconds and too fine-grained for users to control the behavior of the scheduler. We consider that users should have an interface to control the aggregation strength other than specifying the priority bonus in *vruntime*.

Secondly, users should have an interface to allocate multiple groups of `master/slave` flexibly. Nowadays, we have multi-Core processors with complex memory hierarchy. For example, Intel Core 2 Quad has four Cores. Each Core has own L1 data/instruction cache and a single L2 cache is shared between two Cores, while no cache is shared between all Cores. In this case, aggregating sibling threads with a single `ia_mm` may increase the overhead of communication between Cores not sharing the same L2 cache. We consider that users should have an interface to allocate multiple groups of `master/slave` Cores.

3 APIs for Inter-Core Time Aggregation Scheduler (IAS)

In this section, we propose the APIs for IAS, `set_ias_agg` and `set_ias_alloc`, which deal with the problems described in Section 2.4. In Section 3.1, we explain `set_ias_agg`, which controls the strength of aggregation of sibling threads. In Section 3.2, we explain `set_ias_alloc`, which controls the assignment of `master/slave`.

3.1 `set_ias_agg`

There are five arguments passed to `set_ias_agg` as shown below.

- *pid*
- *agg*
- *bonus_type*
- *bonus_value*
- *limit*

We specify the process ID to control the aggregation of its sibling threads by *pid*. At the implementation level, `set_ias_agg` sets the values of *agg*, *bonus_type*, *bonus_value*, and *limit* to the members added in the data structure of memory address space of thread *pid*. The values stored in the members in the memory address spaces are the parameters for IAS to make scheduling decisions. We explain each member below.

agg must be 0 or 1. If *agg* is 0, IAS does not aggregate sibling threads of *pid*. If *agg* is 1, IAS aggregates sibling threads of *pid*. The kernel initializes the values of *agg* as 0 and IAS does not aggregate any threads by default. Users should set *agg* as 1 only when they judge that the aggregation of the sibling threads is effective.

IAS provides two ways to specify the priority bonus with *bonus_type* and *bonus_value*. *bonus_type* takes 0 or 1. If *bonus_type* is 0, IAS gives the priority bonus in *vruntime* specified in `bonus_value`. In this case, `bonus_value` ranges from 0 to over 18,446,744,073G *vruntime*[2]. If *bonus_type* is 1, IAS gives the priority bonus by multiplying the quantum time of the previously executed thread by *bonus_value*. There are four reasons to utilize the quantum time of previously executed thread. Firstly, the change of the additional *vruntime* influences the order of threads in the runqueues. We assume that parallel tasks are equally assigned to sibling threads during their execution. In this case, the difference of *vruntime* between sibling threads are less than the quantum time of previously executed thread. For this reason, we consider that setting the quantum time as the criterion of the priority bonus is reasonable. Secondly, the quantum time changes dynamically according to the workload, therefore, it is hard for users to statically guess the effective priority bonus. Thirdly, it is easy to evaluate the quantum time because CFS tracks it for the calculation of *vruntime*. Fourthly, it is easier to make a guideline of using IAS between different programs. As we mention, the range of `bonus_value` is too wide to properly decide the effective priority bonus to enhance the throughput while keeping a certain fairness between different programs. For these reasons, we consider that using the quantum time provides a reasonable way of the abstraction.

Users can also restrict the number of sibling threads successively scheduled per aggregation on a single Core by specifying the value of *limit*. IAS counts the number of sibling threads successively selected on a single Core. When the count exceeds the *limit*, IAS does not give the priority bonus to sibling threads and resets the count.

3.2 `set_ias_alloc`

`set_ias_alloc` assigns `master`/`slave` roles to each Core. The arguments passed to `set_ias_alloc` are numbers which specify the role of each Core. We assume the use of `set_ias_alloc` from command lines because the allocation of `master`/`slave` influences the execution of all threads in the system and we need to observe the impact while interactively running programs.

[2] *vruntime* has the type of `unsigned long long` and we assume to use 32 bit kernel here.

Table 2. The correspondence between the number and its role in `set_ias_alloc`

Number in `ias_job_alloc[]`	Correspondent Role
0	master_0
1	slave_0
2	master_1
3	slave_1
4	master_2
5	slave_2
6	master_3
7	slave_3

IAS controls the role of each Core by using an array `ias_job_alloc[]`, which we defined inside the kernel. The index of `ias_job_alloc[]` corresponds to the ID of Core starting with 0. For example, the role of Core 2 is stored in `ias_job_alloc[2]`. So far, IAS is able to deal with octa-core processors and the role of each Core is specified with numbers from 0 to 7. We show the correspondence between the numbers and its role in Table 2. In Table 2, Cores on `slave_0` follow the aggregation of Core on `master_0`. Following command sets two inter-core aggregation groups on a quad-core processor, one inter-core aggregation group consists of Core 0 and 1 and another group Core 2 and 3.

```
$ set_ias_alloc 0 1 2 3
\widehat{}
```

4 Evaluation of APIs with `memory` Program in SysBench

In this section, we evaluate the effectiveness of APIs with `memory` program in SysBench[14]. In Section 4.1, we explain `memory` program, our experimental platform, and the method of the evaluation. In Section 4.2, we explain the result and show that our API is effective in utilizing IAS.

4.1 `memory` Program and Experimental Platform

SysBench benchmark suites is a collection of benchmark programs to evaluate the performance of workloads related to Online Transaction Processing. `memory` program in SysBench focuses on the performance of sequential reads from or writes to a memory block. `memory` program creates sibling threads and lets them repeat accessing a specified size of shared or unique memory block until the total accessed size exceeds a user-specified size. There are several metrics in `memory` program such as the average time of each data access and the total elapsed time. We can control `memory` program through the parameters such as the number of threads, the size of the memory block, and the total access size.

We show the parameters used for the evaluation in Table 3. In the following explanation, we show the used parameters in the parentheses. We execute 10

Table 3. Parameters for evaluating `memory` program

Parameter		Specified value
-num-threads		100
-memory-oper		write
-memory-scope		global
-memory-block-size	`set_ias_agg`	4(MB)
	`set_ias_alloc`	1, 2, 4, 6, 8, 10 12, 14, 16(MB)
-memory-total-size	`set_ias_agg`	10 (GB)
	`set_ias_alloc`	5, 10, 15 20, 25(GB)

Table 4. Specification of our experimental platform

Processor	Intel Core 2 Quad
L2 Cache Size / Latency	3MB×2 / 5.6 ns
Memory Size / Latency	1.8GB / 74.4 ns
OS / kernel	CentOS 5.3 / Linux 2.6.24

`memory` programs simultaneously to mingle threads of different memory address spaces. We let each program create 100 sibling threads (–num-threads=100) and let sibling threads access the shared memory block (-memory-scope=global) to focus on the effect of utilizing the locality between sibling threads. Each thread writes to the memory block sequentially (–memory-oper=write). We can control the size of the memory block (–memory-block-size) and the total access size (–memory-total-size). We use different values for –memory-block-size and –memory-total-size in the evaluation of each API and explain them in the evaluation method below.

We also show our experimental platform in Table 4. Intel Core 2 Quad is quad-core processor and has two L2 caches, each of which is shared by two Cores.

We measure the total elapsed time and the number of resource stalls (RESOURCE_STALLS.ANY[15]), and compare the results in CFS and IAS with APIs. We show the method of the evaluation in each API below.

Evaluation Method of `set_ias_agg`. In the evaluation of `set_ias_agg`, we focus on the function of setting the value of the priority bonus. We use a single value for the –memory-block-size and the –memory-total-size as shown in Table 3 and set the same parameter to ten `memory` programs. We compare the results of the different methods of setting the priority bonus. We directly specify it in *vruntime* or calculate by multiplying the quantum time of the previously executed thread. In case of setting `bonus_type` as 0, we try wide range of `bonus_value` from 1K to 10M *vruntime* because it is difficult to previously guess the effective value. In case of setting `bonus_type` as 1, we multiply the quantum time by 1 to 5.

Evaluation Method of `set_ias_alloc`. In the evaluation of `set_ias_alloc`, we use five different –memory-total-size values for ten `memory` programs as shown in Table 3, assuming a situation when a user executes several different programs. We also change –memory-block-size to investigate the relationship between the size of the shared working set and the effect of IAS in setting multiple `master/slave` groups.

We prepare three cases, where we set different `master/slave` groups and processor affinity of the threads, and compare their results. The first case is to use a single `master/slave` group, where Core 0 is `master_0` and other three Cores are `slave_0`, and not to set the processor affinity to any threads (Case 1). We set two `master/slave` groups, where Core 0 is `master_0` and Core 1 is `slave_0` while Core 2 is `master_1` and Core 3 is `slave_1`, in the second and the third case (Case 2 and Case 3). The difference between Case 2 and Case 3 is the setting of the processor affinity of threads. In Case 2, we do not specify the processor affinity of threads and threads can be executed in every Core. In Case 3, we divide `memory` programs into two groups as programs of the same total size are split into different `master/slave` group. For example, sibling threads of a `memory` program with –memory-total-size of 10GB are executed on Core 0 and Core 1 while sibling threads of another `memory` program with –memory-total-size of 10GB are executed on Core 2 and Core 3. By specifying the processor affinity as described above, we can divide the workload equally into two Core groups with different L2 caches and restrict the overhead of communication between Cores. We expect the optimal performance in Case 3 and evaluate how close the result in Case 1 and 2 will be. We set the priority bonus as 50M *vruntime* based on our previous experiment[7].

4.2 Results

In this section, we firstly show the results of the evaluation of `set_ias_agg`. Succeedingly, we show the results of the evaluation of `set_ias_alloc`.

Results of the evaluation of `set_ias_agg`. We show the result of the evaluation of `set_ias_agg` in Fig. 4. In Fig. 4, we show the ratio of the execution time in IAS against CFS (lines), and the absolute value of the resource stalls (bars) in each parameter. In Fig. 4, we express each parameter as `d_[1,2,3,4,5]` when we set `bonus_type` as 1, and `s_[1K,10K,100K,1M,10M]` when we set `bonus_type` as 0. We see that the reduction of the execution time and the resource stalls becomes larger as we increase the value of the parameter when we set `bonus_type` as 1. On the other hand, we see little effect of IAS when `bonus_value` is from 1K to 100K when we set `bonus_type` as 0. When we set `bonus_value` higher than 1M *vruntime*, we see the effect becomes larger. We consider that `bonus_value` below 1M *vruntime* is too small in this experiment because the average additional *vruntime*, which we measure simultaneously during the experiment, is 33M.

We conclude that we can set the priority bonus easily and effectively by setting the priority bonus based on the quantum time rather than specifying in *vruntime*.

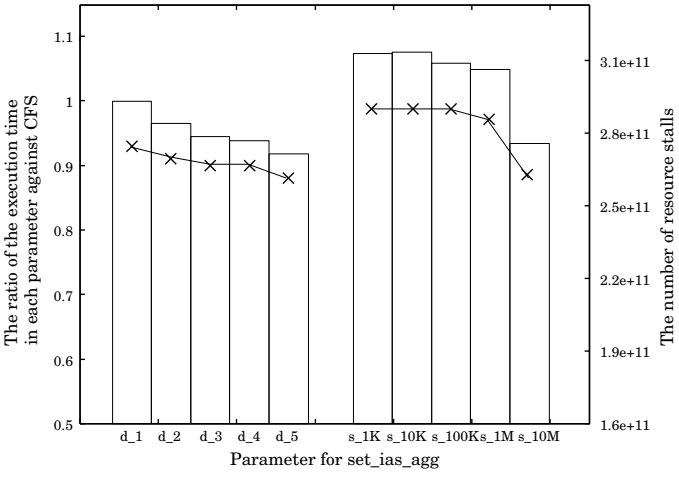

Fig. 4. The effect on the execution time (lines) and the resource stalls (bars) in using `set_ias_agg`

Results of the evaluation of `set_ias_alloc`. We show the result of the evaluation of `set_ias_alloc` in Fig. 5. In Fig. 5, we show the ratio of the execution time in IAS against CFS in Case 1, 2, and 3. We can see the effect in Case 2 and 3 are larger than that in Case 1. We consider that the result shows the effect of space-multiplexing, which reduces the overhead of communication between Cores in Case 2 and 3. We also consider that the effect will be larger in many-core processors with deeper memory hierarchy.

When we compare Case 2 and Case 3, Case 3 seems advantageous only when the memory block is less than 6MB. In other parameters, the effects in Case 2

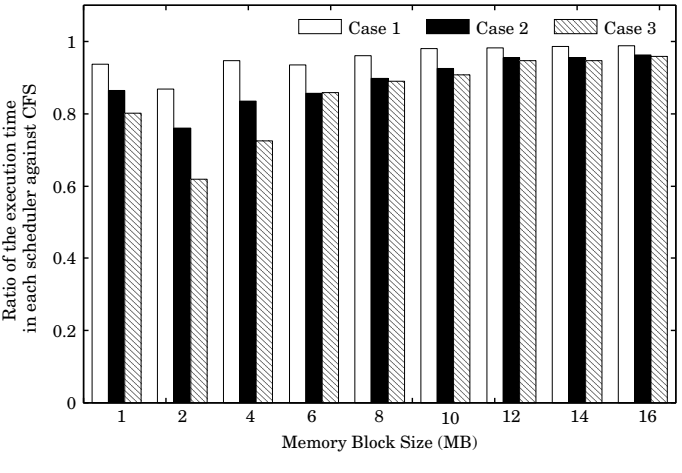

Fig. 5. The effect on the execution time in setting multiple `ia_mm` with `set_ias_agg`

and 3 are almost the same while Case 3 is the optimal setting as we described in Section 4.1. By considering users do not have to set the processor affinity, Case 2 becomes more advantageous as working set gets larger. We conclude that we can gain the effect of reducing the overhead of communication between Cores by setting multiple `master/slave` Cores with `set_ias_alloc`.

5 Related Research

As caches are generally shared between Cores in multi-core processors, many thread-level schedulers have been proposed to utilize the caches. Many researches proposed to split the thread execution into the sampling phase and the scheduling phase[16,17,18]. In the sampling phase, the kernel samples the information of each thread execution. In the scheduling phase, the kernel schedules the combination of threads to execute them simultaneously between different Cores based on the information obtained in the sampling phase. For example, Fedorova[18] calculates the size of the working set of each thread by tracing its behavior in the sampling phase. They schedule the combinations of threads to let the sum of the working set fit within the capacity of the L2 cache. The benefit of this sampling and scheduling approach is that we can apply this method to any case of thread execution in theory. The problem of this approach is the overhead of sampling information, especially when running many threads, as IAS supposes[12,19]. Moreover, the complexity of optimal co-scheduling in multi-core processor, where a cache is shared between all Cores and the number of Cores is more than 2, is NP-complete[2]. We focus on a more realistic approach. Even though IAS does not intend to schedule threads optimally, IAS only focuses on the memory address space of each thread and its overhead is little as we see in Section 2.3.

The basic idea of our approach is similar to that of Chen[3] in that their scheduling algorithm executes threads sharing the working set simultaneously on different Cores to utilize the shared cache. Chen also proposes a compiler to control the granularity of threads to fit with the caches of the processor. Chen's approach is applicable to fine-grained multi-threaded programs, which contains DAGs inside, and shows that their scheduling method can enhance the throughput by carefully tuning the granularity of threads by their compiler. The difference between IAS and Chen's approach is that IAS is intended to work for multi-programmed execution while Chen only considers single-programmed execution. IAS does not detect the size of the working set shared between sibling threads while Chen's approach does not consider the influence from other programs. We consider that we can enhance the performance of broader range of multi-threaded programs by mixing IAS and Chen's approach.

Ziemba also focuses on the locality of references between sibling threads and investigates the effect of space-multiplexing with a Web application server[20]. Ziemba sets different processor affinities for threads of HTTP and application servers in executing SPECweb benchmark[21]. Ziemba presents their aggregation is effective and enhances the performance of the Web application server, indicating the locality of references between sibling threads. However, Ziemba

mentions that it is difficult to statically analyze applications and optimally set processor affinities. In this paper, we present that we can gain the effect of space-multiplexing without setting processor affinities in each thread.

6 Conclusion

This paper proposes and evaluates APIs for IAS, which is a kernel-level thread scheduler to enhance the performance of multi-threaded programs. We have proposed IAS, which dynamically aggregates sibling threads in $O(1)$ to utilize the cache shared between Cores. In this paper, we present two APIs, `set_ias_agg`, which controls the aggregation of sibling threads, and `set_ias_alloc`, which controls `master/slave` groups. The effectiveness of our API is described in two aspects. Firstly, we show that we can effectively and easily set the aggregation strength in IAS based on the quantum time of the previously executed thread by using API `set_ias_agg`. Secondly, we show that we can gain the effect of space-multiplexing by grouping Cores and running IAS per group without setting the processor affinity of each thread by using API `set_ias_alloc`.

Our future work includes the investigation of the effect of IAS with more general benchmark applications. We consider that IAS is especially effective in benchmark applications, which runs multiple multi-threaded programs simultaneously such as SPECweb[21]. We also investigate the effectiveness of Helper-thread mentioned in Section 1. Even though we can set the priority bonus easily with `set_ias_agg`, we still have to set the parameter manually. We will develop Helper-thread mechanism to detect the degradation of multi-threaded programs and automatically tune the priority bonuses to enhance the effect of IAS. In addition, we will develop scheduling strategies to control the behavior of Helper-thread such as the frequency of sampling thread information and the granularity of parameter changes.

References

1. Kim, S., et al.: Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, pp. 111–122 (2004)
2. Jiang, Y., et al.: Analysis and Approximation of Optimal Co-Scheduling on Chip Multiprocessors. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 220–229 (2008)
3. Chen, S., et al.: Scheduling Threads for Constructive Cache Sharing on CMPs. In: Proceedings of 19th ACM symposium on Parallel Algorithms and Architectures, pp. 105–115 (2007)
4. DaCapo benchmark suite, `http://dacapobench.org/`
5. The PARSEC Benchmark Suite, `http://parsec.cs.princeton.edu/`
6. Chishti, Z., et al.: Optimizing Replication, Communication, and Capacity Allocation in CMPs. In: Proceedings of the 32nd International Symposium on Computer Architecture, pp. 357–368 (2005)

7. Yamada, S., et al.: Development of a Thread Scheduler for Global Aggregation of Sibling Threads. Research Reports on Information Science and Electrical Engineering of Kyushu University 1(2), 69–74 (2008)
8. Yamada, S., et al.: Impact of Priority Bonuses of Inter-Core Aggregation Scheduler on a Commodity CMP Platform. In: Workshop on Managed Many-Core Systems (MMCS) co-located with ASPLOS (2009), http://www.cercs.gatech.edu/mmcs09/program.htm
9. RUBiS: Rice University Bidding System, http://rubis.ow2.org/
10. Keeton, K., et al.: Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. In: Proceedings of the 25th Annual International Symposium on Computer Architecture, pp. 15–26 (1998)
11. Redstone, J., et al.: An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture. In: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 245–256 (2000)
12. DeVuyst, M., et al.: Exploiting Unbalanced Thread Scheduling for Energy and Performance on a CMP of SMT Processors. In: Proceedings of 20th IEEE International Parallel & Distributed Processing Symposium (2006)
13. Yamada, S., et al.: Effect of Context Aware Scheduler on TLB. In: Workshop on Multi-Threaded Architectures and Applications, Published in CD (2008)
14. SysBench: a system performance benchmark, http://sysbench.sourceforge.net/
15. Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3B: System Programming Guide, Part 2, http://www.intel.com/products/processor/manuals/index.htm
16. Parekh, S., et al.: Thread-Sensitive Scheduling for SMT Processors, Technical report, Dept. of Computer Science and Engineering, University of Washington (2000)
17. Snavely, A., et al.: Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor. In: Proceedings of International Conference on Measurement and Modeling of Computer Systems, pp. 66–76 (2002)
18. Fedorova, A., et al.: Performance of Multithreaded Chip Multiprocessors and Implications for Operating System Design. In: Proceedings of USENIX 2005 Annual Technical Conference, pp. 395–398 (2005)
19. Chandra, D., et al.: Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In: Proceedings of 11th International Symposium on High-Performance Computer Architecture, pp. 340–351 (2005)
20. Ziemba, S., et al.: Analyzing the Effectiveness of Multicore Scheduling Using Performance Counters. In: Proceedings of Workshop on the Interaction between Operating Systems and Computer Architecture (2008)
21. SPECweb, http://www.spec.org/web2009/