Effect of Inter-Core Aggregation Scheduler on a Memory Intensive Benchmark

Yamada, Satoshi
Department of Computer Science and Communication Engineering, Graduate School of Information Science and Electrical Engineering, Kyushu University : Graduate Student

Kusakabe, Shigeru
Department of Advanced Information Technology, Faculty of Information Science and Electrical Engineering, Kyushu University

https://doi.org/10.15017/1517959

出版情報：九州大学大学院システム情報科学紀要. 14 (2), pp.47-52, 2009-09-25. 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14巻 第2号 平成21年9月

Research Reports on Information Science and
Electrical Engineering of Kyushu University
Vol.14, No.2, September 2009

Effect of Inter-Core Aggregation Scheduler on a Memory Intensive Benchmark

Satoshi YAMADA* and Shigeru KUSAKABE**

(Received June 12, 2009)

Abstract: This paper investigates the effect of Inter-Core Aggregation Scheduler (IAS) with `memory` program of SysBench. IAS is a kernel-level thread scheduler which executes sibling threads, threads sharing the memory address space, simultaneously on different processing cores (Cores) of a Chip-Multi Processor. IAS is a promising scheduler to reduce the capacity pressure on the shared L2 cache and enhance the performance. Previously, we showed that IAS enhanced the performance in running a Web application benchmark. To clarify the relationship between the characteristic of thread behavior and the effect of IAS, we estimate its effect on `memory` benchmark, a multi-threaded memory intensive benchmark, by simulating the thread execution of IAS. We show that the effect of IAS approximately follows our estimation and is predictable especially when we cannot expect the positive effect of IAS. We also show that the effect of IAS is related to the size of the data set of `memory` program and the size of the shared L2 cache.

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

1. Introduction

This paper investigates the efficiency of Inter-Core Aggregation Scheduler (IAS) with `memory` program of SysBench[9], a memory intensive benchmark, with various memory access parameters. IAS is a kernel-level thread scheduler which executes sibling threads, threads sharing the memory address space, simultaneously on different processing cores (Cores) of a Chip-Multi Processor (CMP). IAS is a promising scheduler to reduce the capacity pressure on the shared L2 cache and enhance the performance by utilizing the locality of reference between threads.

The memory access latency remains one of the major bottlenecks on CMPs as well as in conventional single processors[4]. In CMP platforms, co-scheduling of threads running simultaneously on different Cores affects the utilization of caches and its performance because CMP has multiple Cores and each Core generally shares caches. In major commodity platforms, co-scheduling of threads is controlled by a kernel-level thread scheduler of Operating System. Therefore, it is popular research area to propose alternative co-scheduling methods on CMP by developing kernel-level thread scheduler[1),6),8)].

IAS dynamically aggregates sibling threads and executes them simultaneously on different Cores based on the assumption that sibling threads share a certain amount of data set. We expect two effect of utilizing the locality of reference between sibling threads from IAS. One effect is that IAS increases the possibility that co-scheduled threads share their data set, thus, decreases the capacity pressure on the cache. Another effect is that IAS decreases the overhead of context switching by leaving the data on the cache, which succeeding threads access. Especially in x86 architectures, TLB entries are flushed in every switch of memory address spaces, therefore we can expect re-use of TLB entries by aggregating sibling threads. IAS may increase the overhead caused by the data contention or the data coherence problem by promoting threads executed on different Cores to access the same working set simultaneously. However, according to the previous research on the analysis of the performance of CMP, the L2 cache misses caused by the lack of capacity is the most influential[5].

Previously we implemented IAS based on Completely Fair Scheduler (CFS) in Linux[3]. We investigated the effect of IAS with RUBiS benchmark, a Web application benchmark program running multi-threaded database server and Web server[2]. IAS reduced the L2 cache misses and page faults and enhanced the performance of a Web application. In addition, the effect of IAS is influenced by the priority bonus which control the strength of the aggregation of sibling threads.

This paper investigates the effect of IAS with `memory` program, a simple memory intensive benchmark, to clarify the relationship between the char-

* Department of Computer Science and Communication Engineering, Graduate Student
** Department of Advanced Information Technology

acteristic of thread behavior and the effect of IAS. For this purpose, we preliminary estimate the effect of IAS by simulating the thread execution in CFS and IAS, and compare with the real effect.

The rest of the paper is organized as follows. Section 2. explains IAS. Section 3. explains `memory` program and the estimation of the effect in CFS and IAS. Section 4. compares the estimated effect and the real effect of IAS. Section 5. introduces several related works and clarifies the position of our research. Section 6. concludes the paper.

2. Inter-Core Aggregation Scheduler

We implemented IAS by modifying Completely Fair Scheduler (CFS) in Linux 2.6.24[3]. The implementation of IAS is based on Time Aggregation Scheduler (TAS) that we have implemented for single Core platforms.

We explain CFS in Section 2.1, TAS in Section 2.2, and IAS in Section 2.3. In this paper, we only explain the outline of each scheduler with **Fig. 1**. **Figure 1** represents threads in executed order on dual Core machine. Each circle expresses a thread and the patterns inside threads represents the memory address spaces. The numbers represents the Core ID, and threads are executed on either Core. Detailed explanation is shown in our previous paper[3].

2.1 Completely Fair Scheduler

In scheduling threads, CFS just chooses a thread of the highest priority. CFS counts the process time of each thread, total CPU time consumed by the thread, in nanoseconds and calculates the priority as `vruntime` based on `nice` value and the process time of the thread. When a thread is executed, `vruntime` of the thread is increased according to its increased process time. CFS sets higher priority for threads with less `vruntime` to accomplish fair usage of CPU between threads which start at the same time with the same `nice` value. 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.

2.2 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 priority bonus (PB) to the sibling threads of the currently executed thread. As we mentioned, the priority of a thread becomes higher when the `vruntime` of the thread becomes smaller. Therefore, PB for TAS works to reduce the `vruntime` of the sibling thread. If we set PB higher, we can expect more aggregations of sibling threads. We can interactively change PB with a system call we implement.

2.3 Overview of Inter-Core Aggregation Scheduler

The basic idea of IAS is to let one Core (`slave`) follow the time aggregation of another Core (`master`). We consider a case that `master` is Core 0 and `slave` is Core 1 as we show in **Fig. 1**. In **Fig. 1**, We run independent TAS per Core first. When the scheduler on `master` finds a chance of time aggregation of 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` is able to modify `ia_mm` and `slave` only refer to `ia_mm`. When `ia_mm` is set to an actual memory address space by `master`, `slave` looks for the sibling threads sharing the memory address space of `ia_mm` from their own runqueue. If there exists sibling threads, the schedulers consider the threads as the candidates for the next threads with the PB for IAS. Thus, IAS can execute sibling threads nearly simultaneously on different Cores.

In this case, we use one `ia_mm` for dual Core platform. However, we can change the number of `ia_mm` and specify the role of each Core by issuing a system call we implement as well as changing the value of PB. We consider that tuning IAS by the system call above has potential to utilize many-Core processors having more complex memory hierarchy. The investigation of tuning IAS with multiple `ia_mm` is our future work.

3. `memory` program and the Estimation of the Effect in CFS and IAS

This section explains `memory` program and the evaluation method of the effect of IAS. We also estimate the effect of IAS by simulating the execution of `memory` program in CFS and IAS. We explain `memory` program in Section 3.1 and the evaluation method in Section 3.2. We show the estimation of the effect in each scheduler in Section 3.3.

3.1 `memory` program in SysBench

We show the outline of `memory` program in **Fig. 2**. `memory` program creates a specified number of threads and let them repeat accessing a specified

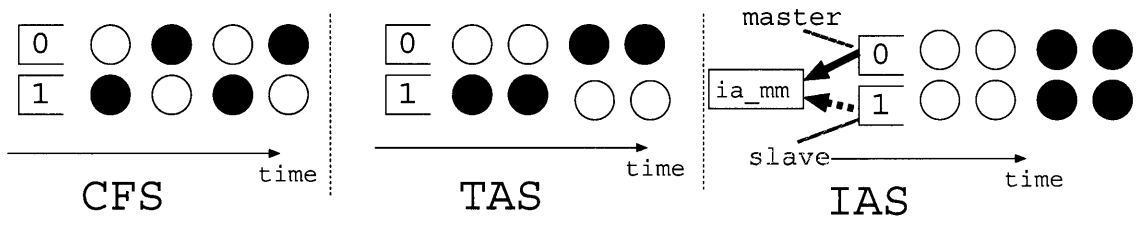

Fig. 1 Outline of CFS, TAS, and IAS.

amount (`block_size`) of memory area. Threads of a single `memory` program share the memory address space. The access of each thread is expressed as arrow in **Fig. 2**. The memory access is repeated until the total access size of every thread exceeds a specified size(`total_size`). SysBench provides two types of memory access, `read`, reading a value from a specified address, and `write`, writing a value to a specified address, until one thread accesses to a specified amount of memory. SysBench also provides two access modes, `seq`, the address is incremented sequentially, and `rnd`, the address is modified randomly. We can choose if each thread accesses to the same data set (`global`) or independent data set (`local`). In case of `global` mode, the size of data set of a `memory` program is `block_size`. In case of `local` mode, the size of data set of a `memory` program is the product of `block_size` and the number of threads.

3.2 Method of Evaluation

To investigate the effect of IAS, we run multiple `memory` programs simultaneously to let threads of different memory address spaces mingled in the runqueues and measure the average execution time. To make the effect of IAS clear, we modify two parts of `memory` program as follows.

1. Let the parent thread to yield CPU after creating a thread.
2. Let a thread yield CPU after accessing `block_size` of data.

The first modification is to let sibling threads dispersed and threads of different memory address spaces mingled in the runqueue. By default, the parent thread continues to create threads until it expires the quantum time and a block of sibling threads are queued, therefore, the effect of IAS may not be clear. The second modification is to clarify the effect of IAS and `block_size`. The default `memory` program lets one thread iterate the memory access until it expires the quantum time. Therefore, a thread accesses the data set from the second iteration is likely to hit the data on the cache loaded by the first iteration of the thread. We consider the second modification makes the effect of IAS clear to utilize the locality of references between threads.

We investigate the effect of IAS in both `read` and `write` and both `global` and `local`. In `local` mode, we cannot expect that IAS utilizes the shared data set of sibling threads because each thread does not share the data set. However, sibling threads which yield CPU can be executed before the data set loaded on the cache by previous execution is purged from the cache by threads of different memory address spaces as a result of the aggregation of sibling threads. We only measure `seq` mode and the effect in `rnd` mode is future work. 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` in from 16KB to 10MB in `global` and from 163B to 100KB in `local` to investigate the relation of the effect of IAS and the size of the data set´. We set PB as 50 millions which is large enough to aggregate more than 99% of the sibling threads during the experiments based on our preliminary measurements.

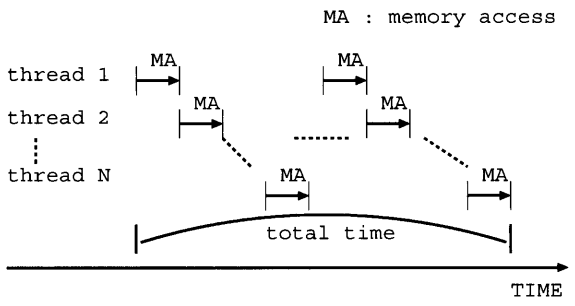

Fig. 2 Outline of `memory` program in SysBench.

3.3 Estimation of the Effect of IAS

We estimate the effect of the simultaneous execution of multiple `memory` programs mentioned in Section 3.2. To estimate the execution in CFS and

IAS, we make two assumptions as follows.

- Threads of different memory address spaces are always executed simultaneously on different Cores in CFS.
- Sibling threads are always executed simultaneously on different Cores in IAS.

We consider that the first assumption is accomplished by the first modification of `memory` program described in Section 3.2. We consider that the second assumption is satisfied by increasing PB. We show the comparison of the thread execution in each scheduler in **Fig. 3**. **Figure 3** represents the simultaneous execution of four `memory` programs. As we show in **Fig. 3**, CFS executes threads of different memory address spaces (Mixed execution). On the other hand, IAS simultaneously executes sibling threads except when the switch of the threads of different memory address spaces happens.

Based on the assumption above, we consider that we can estimate the effect of IAS compared to CFS by comparing the execution time below:

1. Run X `memory` programs, each of which creates one thread, and measure the elapsed time for all threads to access Y bytes
2. Create X sibling threads and measure the elapsed time for all threads to access Y bytes

The first case simulates thread execution in CFS and the second case simulates thread execution in IAS. By comparing the execution time of each case, we estimate the effect of IAS against CFS on the execution time.

In case of simulating `global` mode, the total size of the data set is decided by X and `block_size` in case 1. and only by `block_size` in case 2.To set the total `block_size` as equal to the measurement in Section 3.2, we set X as 10. We set Y as 5000GB, which is large enough to ignore the difference of overhead of creating threads or processes and focus on the difference of the data set. We change `block_size` from 16KB to 10MB as we do in Section 3.2.

In case of simulating `local` mode, the total size of the data set is decided by X and `block_size` in both case 1. and 2. as we mentioned in Section 3.1. We set X as 10 as we do in simulating `global` case. We choose `block_size` from 163B to 100KB to correspond to the total size of data set of the measurements in Section 3.2.

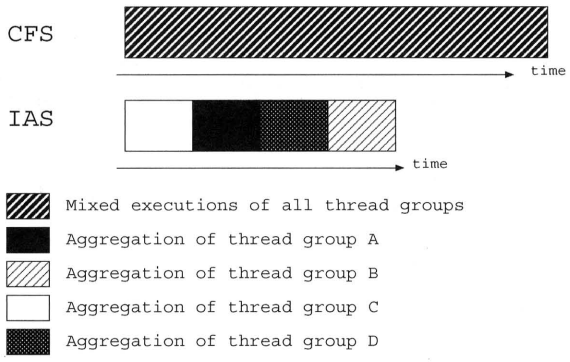

Fig. 3 Comparison of executing `memory` program in each scheduler.

4. Results of the Estimated Effect and the Real Effect of IAS

We execute all the experiments explained in Section 3. in the platform shown in **Table 1**. Intel Core 2 Duo is a dual Core processor and each Core shares the L2 cache.

We show the result in `read` and `write` operation of `global` mode in **Fig. 4** and **Fig. 5** respectively. In each figure, we show the estimated and the real effect of IAS as a ratio of the execution time between CFS. As we can see in **Fig. 4**, the difference between the estimated effect and the real effect is as little as a few percent. The effect on both the estimated and the real effect on `read` is at most 10 % of the reduction of the execution time, which is not as large as the effect in `write` in **Fig. 5**. We consider that the smaller effect in `read` comes from data prefetching of hardware. Data prefetching works efficiently with the sequential read of data, which reduces the effect of IAS. We see slightly larger effect in smaller sizes of `block_size` in `read` because the effect of prefetching on the total elapsed time is smaller. On the other hand, we see the larger reduction of the execution time in `write` especially when the size of access data is between 512KB and 4MB. As the estimated effect implies, we see the largest effect when the size of access data is 2MB. We consider that the largest effect of IAS is related to the L2 cache size of the platform. When `block_size` is smaller, the possibility that the previously loaded data set is still left in the L2 cache increases in CFS, therefore, we do not see the clear effect of IAS. When `block_size` is bigger, even the aggregation of sibling threads purges the data set from the L2 cache, therefore, we see that the effect becomes small.

We show the result in `read` and `write` operation of `local` mode in **Fig. 6** and **Fig. 7** respectively.

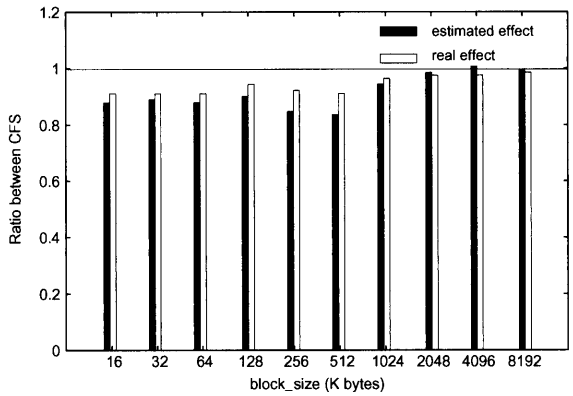

Fig. 4 The estimated and real effect of IAS against CFS in `global` and `read`.

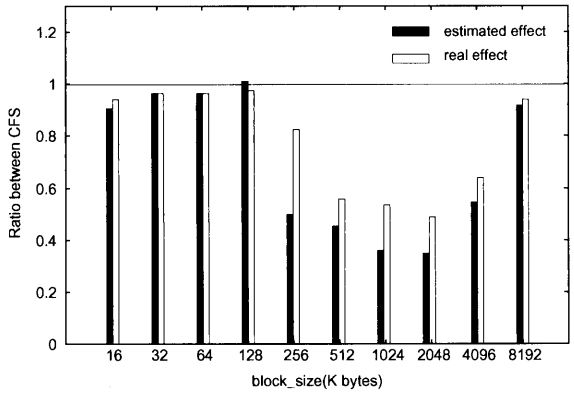

Fig. 5 The estimated and real effect of IAS against CFS in `global` and `write`.

We can see the similar effect to `global` mode in that the effect in `read` is smaller than `write` and that the effect becomes small in increasing `block_size` in both `read` and `write`. As we mentioned, the total size of the data set in one `memory` program is the product of each `block_size` and the number of threads created. In `write`, we can see the largest effect when `block_size` is 20KB and the size of the data set is about 2MB, corresponding to the size of L2 cache. We consider that the effect becomes smaller in other `block_size` because of the same reason mentioned in `global` mode. In case of `local` mode, the effect in `read` is larger than in `global` mode. We consider that the effect of data prefetch is smaller because the `block_size` is smaller in `local` mode, therefore, IAS results in larger effect.

To support the effect we show above, we count L2 cache misses, DTLB misses, and ITLB misses during the execution when the total size of the data set is 2MB in `write` and `global`. We show the result in **Table 2**. We can see the large reduction of each event as the execution time.

Finally, in most of `block_size` parameters, the real effect of IAS is worse than the estimated effect because "Mixed execution" of threads did not always occur in CFS. However, we can approximately estimate the effect of IAS, and especially the cases when we cannot expect the positive effect of IAS.

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

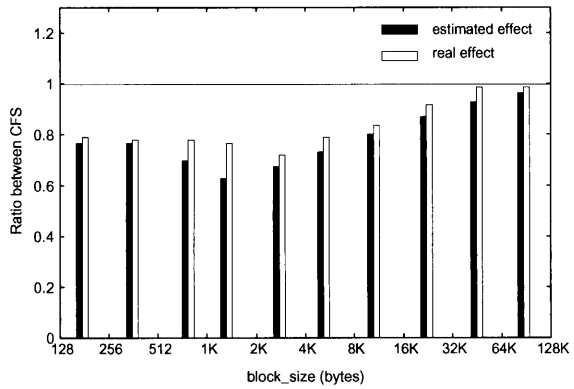

Fig. 6 The estimated and real effect of IAS against CFS in `local` and `read`.

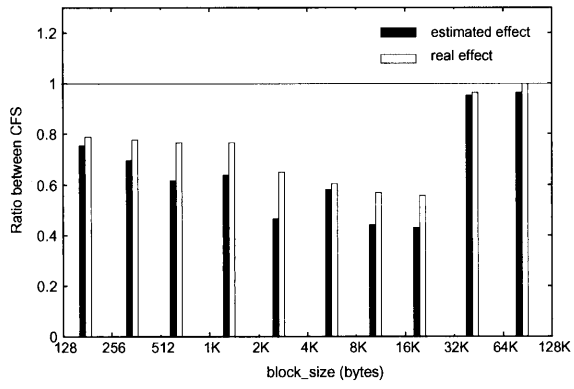

Fig. 7 The estimated and real effect of IAS against CFS in `local` and `write`.

5. Related Works

The basic idea of our scheduler is similar to that of Chen[5]. Their scheduling idea is to run the threads sharing the working set simultaneously on different Cores of CMP. They analyze and modify applications to select the appropriate thread granularity for the applications and schedule the threads

Table 2 Effect on memory related events when the size of the total data set is 1MB (thousand times).

Scheduler	L2 cache misses	DTLB misses	ITLB misses
CFS	29,504	23,386	973
IAS	5,468	17,480	155

statically. They logically certify the efficiency of their scheduling and also demonstrate the effect on their simulator.

However, in most of the commodity platforms, co-scheduling of threads are managed by Operating System. Therefore, many kernel-level thread scheduler for CMP and SMT have been proposed[1),6),8)]. One popular approach is to repeat the phases of sampling thread information and scheduling based on the information. Taking this approach enables the kernel to guess the succeeding behavior of the threads and appropriate co-scheduling. However, the cost of sampling information from each thread can be too large as a practical solution, especially when the OS is loaded with many threads[7)].

Our scheduler just focuses on the memory address space of each thread, therefore, the overhead of scheduling is small. The strength and the number of the aggregations is tunable through the system call we implemented. The drawback of IAS is that IAS works only when we execute multi-threaded programs. However, we consider that many modern programs, especially commercial programs, are getting multi-threaded. Actually, many programs have been multi-threaded as CMP and SMT platforms spread. For example, database servers and Web servers, such as MySQL and Apache HTTP Server, use multiple sibling threads to handle multiple client connections efficiently. Besides, many languages such as Java, Perl, Ruby, Python, Erlang, etc., now support the development of programs using multiple sibling threads. We also have the compiler support to develop programs using sibling threads such as OpenMP, MPI, and Open64. Therefore, we expect that we will have more multi-threaded programs and more chances to apply IAS.

6. Conclusion

This paper investigates the effect of Inter-Core Aggregation Scheduler (IAS) with `memory` program of SysBench. IAS is a kernel-level thread scheduler which tries to execute threads sharing the memory address space simultaneously on different processing cores of a Chip-Multi Processor. To clarify the relationship between the characteristic of thread behavior and the effect of IAS, we estimate the effect of IAS on `memory` benchmark. We show that the real effect of IAS approximately follows our estimation. Our estimation especially effective when we judge the cases when we cannot expect the positive effect of IAS. We also show that the effect of IAS is related to the L2 cache size.

Our future work includes the investigation of the effect of multiple Inter-Core aggregations by using multiple ia_mm on processors with more Cores such as Intel i7. We will also investigate the efficiency of IAS in more practical benchmarks such as SPEC jApp or SPEC WEB. We will also investigate the efficiency of helper threads which automatically tune the priority bonus for IAS.

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, et al., "Impact of priority bonuses of Inter-Core Aggregation Scheduler on a commodity CMP platform", MMCS, 2009.
3) Satoshi Yamada and Shigeru Kusakabe, "Development of a Thread Scheduler for Global Aggregation of Sibling Threads", Research Reports on Information Science and Electrical Engineering of Kyushu University, Vol.1, No.2, pp.69–74, 2008.
4) Naoto Fukumoto, et al., "Effect of Data Prefetching on Chip MultiProcessor", IPSJ SIG Technical Report, 2007-ARC-173, pp.19–24, 2007.
5) 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.
6) Shugo Ogawa, Kei Hiraki, "A Speedup Technique with Scheduler Using Process Execution Information" *Vol.46 No.SIG 12 (ACS 11)*, pp. 161–169, 2005
7) D. Chandra, Fei Guo, Seongbeom Kim, Yan Solihin, "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture" Proceedings of the 11th International Symposium on High-Performance Computer Architecture, pp. 340–351, 2005.
8) A. Snavely, et al., "Symbiotic jobscheduling with priorities for a simultaneous multithreading processor", In Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, pp.66-76, 2002
9) "SysBench: a system performance benchmark", http://sysbench.sourceforge.net/