An Approach for Scalability-Bottleneck Solution

Identification and Elimination of Scalability Bottlenecks in a DBMS

Takashi Horikawa
NEC Service Platform Laboratories
1753, Shimonumabe, Nakahara-Ku,
Kawasaki 211-8666, Japan
t-horikawa@aj.jp.nec.com

ABSTRACT

ACID-compliant DBMSs are said to be difficult to scale their performance by using many more processors, which means that they are difficult to enjoy the benefits of recent many-core systems that wide-spread use of multi-core processors has made practicable. Since DBMSs are indispensable in most of IT systems, scalability issues should be addressed to fulfill the demand of handling large quantity of data. This paper proposes a viable approach for solving scalability issue, in which lock-related bottleneck will be identified from event trace based measurements and scalability will be improved by replacing the bottleneck-lock with fine-grained locks. This paper also describes a case study on the application of the proposed method, in which the scalability of a many-core system in executing DBT-1 transactions with MySQL adopting the InnoDB storage engine has been successfully improved. Since applying the proposed method produced the increase in maximum throughput of the 16-CPU system by 1.6 times, the method is promising, as long as lock-related bottlenecks are of concern.

Categories and Subject Descriptors

C.4 [**Computer Systems Organizations**]: Performance of Systems—*design studies, measurement techniques, performance attributes*; D.2.8 [**Software Engineering**]: Metrics

General Terms

Experimentation, Measurement, Performance

Keywords

Scalability, Database Management System, Bottleneck, Critical Section, Benchmark Program, Performance Tuning, Event-Trace, Many-core System, and Symmetric Multiprocessing

1. INTRODUCTION

Thanks to widespread use of multi-core processors, we can easily use many CPU-cores in IT systems (many-core systems), which mitigates the shortage of CPU resource and enables us to bring performance improvement in the system. Not all types of IT systems, however, are easy to reap the benefit of multi-core processors. Some types of systems such as WEB servers and application servers have high scalability, which is already exploited with scale-out architecture, which means adding a new server machine that will take a share of the workload to the system, to achieve higher performance. Other types of systems such as ACID[1]-compliant relational database systems (DBMSs) are said to be poor in scalability, which means that their performance cannot be easily improved by using many more processors, which is unfavorable for many-core systems that wide-spread use of multi-core processors has made practicable. Since DBMSs are indispensable in most of IT systems, scalability issues should be addressed to fulfill the demand of handling large quantity of data.

Deep understanding of performance characteristics is crucial for performance improvement of many-core systems, as they differ from traditional systems in that traditional multi-processor techniques are no longer efficient for many-core systems. Since many-core systems are rich in CPU resource, their bottlenecks tend to be not CPU resource but some element other than CPU, called scalability bottleneck. This means that system performance cannot be improved by adding CPUs to the system. Many-core systems are much more likely to have this tendency than traditional systems, and thus, higher emphasis is placed on scalability attribute of IT systems then ever before.

In order to address scalability issues a well-formed methodology is required, particularly for traditional ACID-compliant relational database systems, because they are crucial in most of IT systems. Although the BASE[2] consistency model is used to enhance DBMS scalability, some mission critical systems still requires ACID consistency model and/or there are a lot of needs for enhancing system performance by upgrading system platform (OS and/or middleware) without modifying existing application code that works with a traditional DBMS.

Based on the above considerations, this paper focuses on the scalability of a traditional ACID-compliant relational database system. Specifically, this paper proposes an event-tracing technique based method to identify scalability bottleneck and applies the method to an open-source DBMS in executing a benchmark program that models business transactions in the real-world. The proposed method has succeeded to discover that the scalability bottleneck was, as expected, the critical section enforced with the lock.

This paper also tries to improve the scalability of the system by replacing the bottleneck-lock with fine-grained locks, in which the granularity of shared data protected with the bottleneck-lock will be broken down. Since the throughput performance of the system has

ICPE'11, March 14–16, 2011, Karlsruhe, Germany.

[1]Atomicity, Consistency, Isolation, Durability
[2]Basically Available, Soft state, Eventual consistency

successfully improved, the author believes that the tuning method used in the experiment is a possible countermeasure against scalability bottlenecks.

Contributions of this paper are: 1) it proposes a method for identifying the scalability bottlenecks based on an event-tracing technique, 2) it illustrates the availability of the method through an experiment in which one of the most popular open-sourced ACID-compliant relational database systems was measured and analyzed, and 3) it demonstrates the possibility of eliminating the bottleneck by breaking down the granularity of the bottleneck-lock and by adopting an atomic instruction of the CPU. The proposed methods should be a meaningful first step for establishing a well-formed procedure to address scalability issues.

The rest of this paper is organized as follow. Section 2 shows the background and related work of this study. Section 3 describes the method used in this study to identify scalability bottlenecks and applies the method to an experimental system that executed the DBT-1 benchmark program on MySQL DBMS. Section 4 applies the scalability-bottleneck analysis results to tune the bottlenecks and illustrates the performance improvement. Section 5 discusses the results of the experiments, followed by conclusions described in Section 6.

2. BACKGROUND AND RELATED WORK

This section briefly introduces the basics of scalability bottlenecks and describes related works that deal with the bottleneck and/or its related subjects.

2.1 Definition of scalability

It was advocated that the scalability, a desirable attribute of an IT systems, has two different aspects; structural scalability, the ability of a system to expand in a chosen dimension without major modifications to its architecture, and load scalability, the ability of a system to perform gracefully as the offered traffic increases [13].

This paper focuses on the structural scalability, especially that concerning the number of processors. The structural scalability has been get attention recently with accompanying the wide-spread use of many-core processors. The load scalability, on the other hand, is fundamental property of IT systems from the very beginning of computer systems.

The author considered that the evaluation of the structural scalability can be conducted through the comparison of the load-scalability characteristics among various systems which differ in the number of CPUs.

2.2 Scalability bottleneck in DBMSs

An ACID-compliant DBMS has to implement some concurrency-control to ensure the ACID properties, making a DBMS thread unable to work independently with other threads. In other words, DBMS threads are processing their work with interacting with each other. One type of interactions is that a DBMS thread has to wait a processing result of another DBMS thread. Another type of interactions between DBMS threads are contentions for some shared data. To implement those concurrency controls, mutual-exclusion enforced by means of a lock is usually used to arbitrate the contentions. When a thread tries to acquire a lock that has been acquired by another thread, the former thread has to wait until the latter thread releases the lock.

Suppose an extreme case in which every DBMS thread needs to acquire a particular lock to process its work. Since only one thread can acquire the lock at a time, the other threads have to wait the lock to be released, which means that only one thread can carry forward the useful work and the other threads cannot. In such a situation, only one CPU can contribute to beneficial work, and thus there is no performance gain by adding CPUs to the system. This is a typical appearance of scalability bottleneck.

2.3 Related Work on Scalability Bottleneck

As IT systems commonly adopt multi-core CPUs, scalability gets emphasized and studied from various points of view including experiments to investigate the performance characteristics of many-core systems, principles in data-management methodology, system architectures, and concurrency-control strategies.

2.3.1 Measurement result

There are published performance measurements of MySQL executing DBT-1 benchmark program in the WEB site of IPA (Information-technology Promotion Agency, Japan). Figures 1 and 2 shows the variation of the throughput (Y-axis) as a function of the load (X-axis) for the DBMS server with four CPUs and eight CPUs respectively.

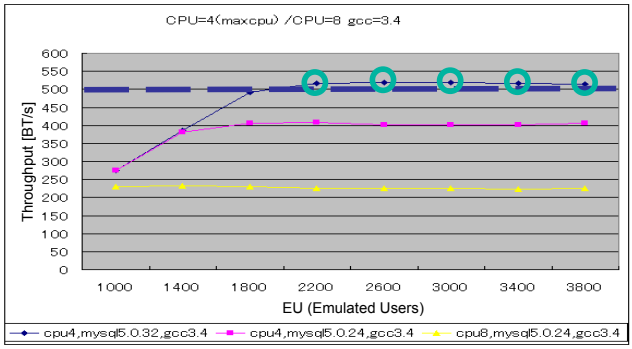


Figure 1: Throughput of four-CPU case [6] measured with the DBT-1 benchmark program. The load intensity was changed by changing the EU (Emulated Users) parameter.

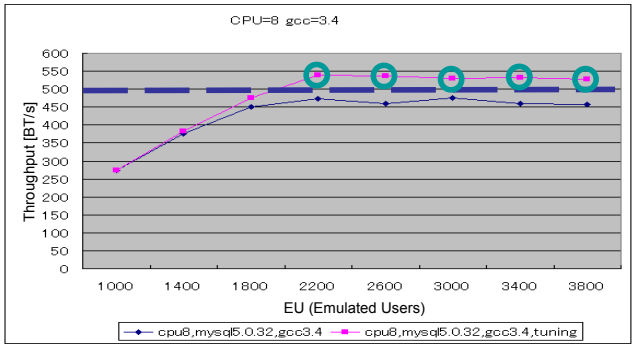


Figure 2: Throughput of eight-CPU case [7].

These results shows that additional four CPUs (from four to eight CPUs) seldom produced performance gain; throughputs were saturated at about 500 [BT[3]/s] for both systems with four and eight CPUs using MySQL version 5.0.32. With taking into account the CPU utilization result for eight-CPU system (Figure 3), the bottleneck of the eight-CPU system was something other than CPU resource; it's an appearance of a scalability bottleneck.

Newer version of MySQL (5.4 and later) is said to be designed to deliver significant performance and scalability improvements [8].

[3]Business Transactions

This paper examines and analyzes the performance of MySQL 5.5.2-m2 using the InnoDB storage engine in executing the DBT-1 benchmark program and tries to tune its performance.

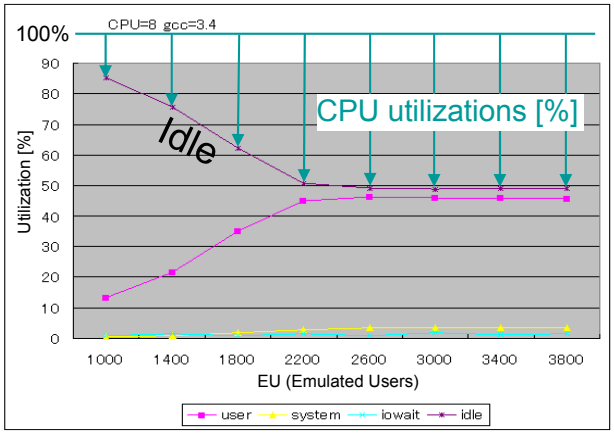


Figure 3: CPU utilization of eight-CPU case [7]. Since CPU utilizations are shown in its constitutions (user, system, and iowait) in the graph, you can grasp them with '100% − idle' as shown in the graph.

2.3.2 *Critical section in DBMS*

Critical sections in DBMS have attracted attention for a long time and very early works include what exhibits the convoy phenomenon [11], which is the forming of a queue of lock-waiter processes in a DBMS. Recently the critical-section related bottlenecks have been re-examined as the most significant obstacle for system scalability [20].

Another work [21] was conducted to improve scalability of a particular DBMS by revising its architecture and synchronization mechanism used in it. Although the work achieved an excellent improvement in scalability, it was a pity that the DBMS was not widely used and benchmark transactions used in the experiments seemed to be rather simple and small. Also the work focused on rather load scalability than structural scalability as little attention was paid to the variation in the system behavior due to the change in the number of hardware contexts (i.e. CPUs). On the other hand, this study focused on the structural scalability through the performance measurement of the system executing MySQL and DBT-1 benchmark program.

2.3.3 *Lock mechanism*

Performance evaluation study on various algorithms used in spin-lock mechanism was conducted with simple and artificial benchmark and concluded that software queuing and inserting a delay between the accesses of lock word according to a variant of Ethernet backoff have good performance [10]. In that research, the number of competing threads for the bottleneck-lock was less than the number of processors and the threads did not sleep (no blocking). The other cases were out of the scope of the research.

A study on the strategy of the choice between spinning and blocking was carried out through a combination of analysis and simulation [12]. They pointed out that predictability of lock holding times is significant and concluded that their results may be used for heuristically determining how long threads should be spinning to wait the lock-release because of the uncertainty of lock holding times. As lock holding times are difficult to know about realistic programs, heuristic algorithms using lock holding times are out of the scope of this study.

As to lock algorithms in DBMS workload, performance evaluation and comparison study of several lock mechanisms were conduced for a particular DBMS [19]. The focus of the study was the variation in the performance with the change in the synchronization primitives used in the system, and was not the way of mitigating the critical section forming the bottleneck enforced with the lock.

The aim of this study is to establish a well-formed methodology to mitigate lock-related bottlenecks with a view to program modification by replacing the bottleneck-lock with fine-grained locks,

2.4 Previous Research of the Author

2.4.1 *Performance of original DBMS*

The author has studied about the performance of MySQL 5.5.2-m2 using the InnoDB storage engine executing the DBT-1 benchmark program [2] on multi-processor systems [16]. The target was chosen for the following reasons; MySQL is one of the most popular traditional ACID-compliant DBMSs and DBT-1, a fair usage implementation of the TPC-W [9] specification, is expected to generate realistic transactions to DBMSs.

To investigate the change in the performance characteristic due to the change in the number of CPUs, the performance measurements were conducted for the systems with 4, 8, 12, and 16 CPUs enabled. The load intensity (Emulated Users, EU) was changed from 400 to 4800 for each case. The measurement results are illustrated in Figure 4, in which each line corresponds to one of the system configurations and shows the variation of a couple of performance indices measured, the throughput (y-axis) and CPU utilization (x-axis). The plots in each line were obtained by changing the EU, which means that each line was drawn with treating the EU as the parameter.

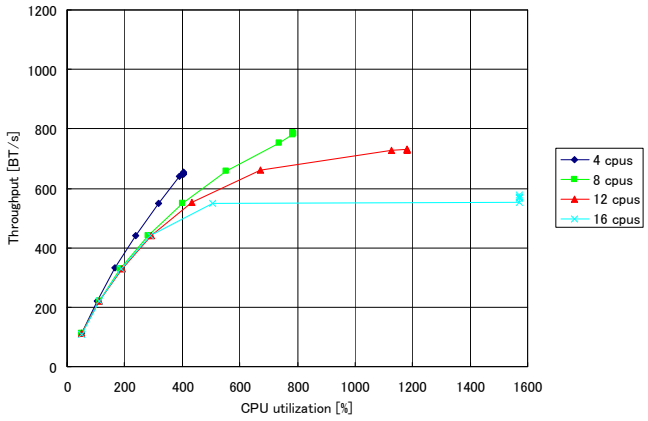


Figure 4: Throughput (y-axis) vs. CPU utilization (x-axis) of the original system. CPU utilization of 100% means that just one CPU is fully loaded, thus, CPU utilization of a system having 16 CPUs, for example, will be at some point between 0% and 1600%.

The graph indicates that there was an scalability bottleneck; CPU utilizations of the 16-CPU system in the region of throughput saturation was sharply-increased (from about 500% to 1600%) without appropriate improvement in the throughput. The author has named this phenomena "CPU jump phenomena".

2.4.2 *Performance analysis*

The cause of the performance degradation occurred in adding CPUs was identified through the measurement using CPU built-in performance counters; there were two kinds of bottlenecks which were in software and hardware layers and their interaction was the cause. The two kinds of bottleneck were:

- at the software layer the MySQL threads competed for the critical section enforced with a particular bottleneck-lock and
- at the hardware layer the CPUs competed for the shared bus through which each CPU accesses the main memory on cache miss etc.

The shared-bus conflict was probably caused by the increase in the bus transaction frequency and the increase was supposed to be caused by the increase in the conflict with the bottleneck-lock. The many processors the system has, the many thread come to access the lock word, and thus the heavier access of the shared-bus occurs. This illustration is consistent with the observation that the performance degraded along with the increase in the number of the processors.

By considering the interaction between these two conflicts, the author concluded that the chain of causality was occurred. That is:

1. The contention for the shared-bus resulted in the increase in the CPI (Clock Cycles Per Instruction).
2. The increase in the CPI resulted in slowing the execution of the critical section that was the bottleneck at the software layer, making increase in the time needed to finish the critical section.
3. The increase in the critical-section execution time of caused the decrease in the throughput of the target system.

2.4.3 *Performance tuning*

A key point of the analysis is that the scalability bottleneck was caused by the shared-bus conflict that was due to the increase in the shared-bus transaction frequency. This suggested the way to improve the performance; reducing the shared-bus conflict is the prime key factor. Since the frequency increased with the increase in the number of spinning threads that repeatedly accessed the lock-word to obtain the lock, the conflict was expected to be reduced by maintaining the number of spinning threads within a particular number.

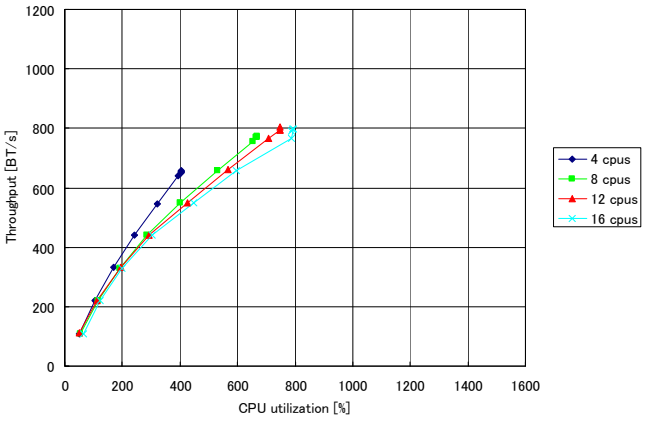


Figure 5: Throughput vs. CPU utilization of the baseline system.

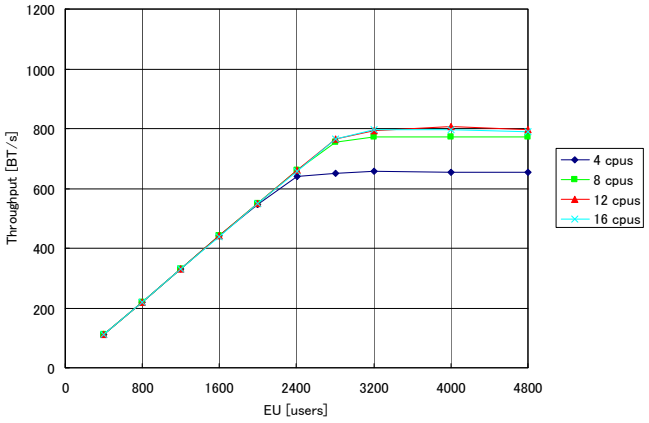


Figure 6: Throughput vs. EU (Emulated Users) of the baseline system.

A lock mechanism that carries out the policy has been developed as a combination of a library and LKM (Loadable Kernel Module), and applied to the bottleneck-lock in MySQL. The performance improvement achieved by the newly developed lock mechanism was measured with the DBT-1 benchmark program. The results are shown in Figures 5 and 6. This paper treats the tuned system as baseline system.

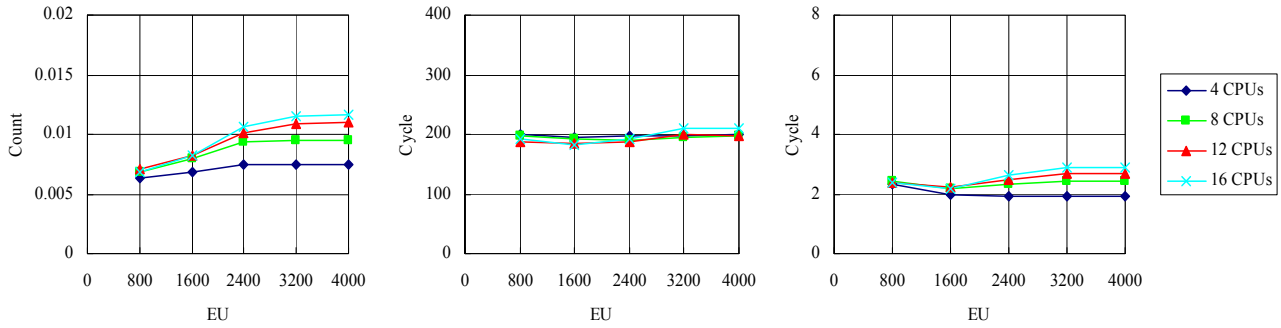
As can be seen in Figure 5, both throughput and CPU utilization characteristics, especially for the 16-CPU cases, were improved; there was neither CPU jump nor throughput inversion. While it is true that the reform of the lock mechanism and its resulting performance improvements are meaningful, the degree of the improvements is by no means satisfactory as the throughput achieved with the 16-CPU system was almost the same as that of the eight-CPU system. Since the CPU utilizations of 12- and 16-CPU cases were not saturated, the obstacle for system performance improvement seemed to be a critical section-related bottleneck that the modification of the lock mechanism highlighted.

This result suggested that the effect of the lock algorithm innovation had a limit in performance improvement and solution of the critical section-related bottleneck was required to achieve further performance improvement. It seemed to contradict the suggestion by Johnson et. al. [20] that proper use of synchronization primitives can be effective to maximize performance and keep critical sections off the critical path in database engines.

2.4.4 *Hardware activities of the tuned system*

Figure 7 exhibits the typical performance indices of the tuned system at the hardware layer. They were calculated from the following elemental indices obtained from multiple measurements with the oprofile tool [5] on the linux. They were: the number of instructions retired (I), Clock cycles when not halted (C), outstanding cacheable data read bus requests duration (D), and the number of any completed bus transactions (B). The performance indices shown in the graphs are: (a) the number of bus transactions per instruction is B/I, (b) the clock cycles per bus transaction is D/B, and (c) the clock cycles per instruction is C/I.

It should be noted here that the indices in the graphs (b) and (c) were virtually constant with the change in the load intensity (EU). These results will be used in Section 4.3.3 to identify the bottleneck at the hardware layer with taking activities at the hardware level into consideration.



(a) Number of Bus Transactions per Instruction

(b) Clock Cycles per Bus Transaction

(c) Clock Cycles per Instruction

Figure 7: Performance indices at the hardware layer (baseline system).

3. BOTTLENECK IDENTIFICATION

Generally speaking, the first step of performance tuning is identifying the bottleneck in the target system, thus, this study also began with identifying the scalability bottleneck. This section illustrates the target system, measurement method and tool, and results of the measurements.

3.1 System Configuration

The measured system and the benchmark program used in the scalability evaluation was the same as those used in the previous study (Section 2.4) of the author. The baseline employed in this paper was MySQL 5.5.2-m2 with the modification for performance tuning described in Section 2.4.3.

The software components of the target system were as follows. 1) Benchmark program was DBT-1 [2], 2) DBMS was MySQL [4] version 5.5.2-m2 (development release) using InnoDB storage engine, and 3) operating system was Linux [3], more specifically CentOS 5.4 [1] with the kernel of version 2.6.18-164.6.1.el5. The hardware consisted of four Intel E7310 CPUs driven by a 1.6 Giga-Hz clock, Intel 7300 chipset, DDR2-667 memory of 14 Giga-Byte, and three 72Giga-Byte hard-drives forming one RAID 0 drive. Each CPU-chip has four processor-cores and 2 Mega-Byte of internal cache memory, which makes target system having total of 16 processors (CPUs).

An execution of DBT-1 benchmark program produces a throughput value in BT per second under the given amount of load in EU for a particular configuration of the target system. Several executions were required to figure out the load scalability characteristics. The sets of benchmark executions were repeated with varying the number of processors in the target system to evaluate the structural scalability.

The CPU affinity mechanism provided with the taskset command was used to enable the specified CPU(s) and disable the other CPU(s), rather than by specifying kernel boot parameter maxcpus. Although the use of taskset command cannot completely disable the CPUs assumed to be nonexistent in the benchmark execution, the author thinks it gives a good approximation of a system having the specified number of CPUs because almost all portion of the program execution was on the DBMS and load generator programs. On the other hand, the use of maxcpus parameter tends to enable processor-cores equally among CPU-chips, resulting in a situation in which each enabled processor-core has more CPU-cache than that in the assumed situation. This difference can affect benchmark results.

3.2 Measurement Method and Tool

Critical sections enforced by locks are seemed to be the most probable cause of the scalability bottleneck in the target system and thus measurements to quantify the lock-waiting times have been conducted. The fundamental concept of lock-waiting time is similar as that of thread waiting time [18], but details are different in some ways; 1) waiting times are accumulated on a resource to resource basis and 2) the periods of busy wait (spin wait) are included in the lock-waiting time.

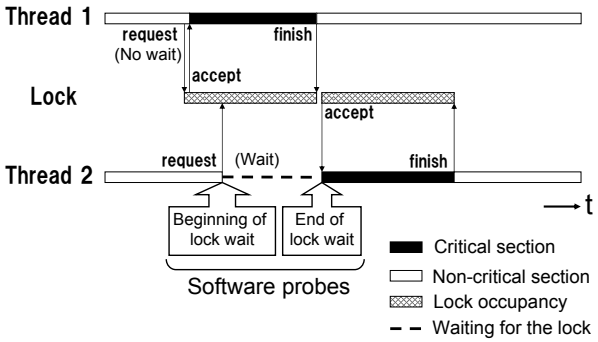


Figure 8: Software probes to mark the boundaries of lock-waiting periods were implemented in the application program, in addition to the mandatory kernel probes.

Based on this idea, measurements were conducted with an event-trace based measurement tool [14, 15] capable of obtaining specific events that relate Linux kernel activities. To detect the lock-waiting period, application-probes (Figure 8) were implemented in MySQL and used in the measurements in addition to the tool's default events. More specifically, the application-probes were placed at the beginning and end of the lock-waiting procedure in following functions in the innoDB storage engine:

- mutex_enter_func()
- rw_lock_s_lock_func()
- rw_lock_x_lock_func()

Two types of measurement were conducted to identify the bottleneck of the system; one was to identify which lock was the bottleneck and the other was to identify which critical section enforced

by the bottleneck-lock was the bottleneck. In the former measurement, the application probes output the line-number information that indicates where the lock-initialization was invoked (*cline* member variable in the lock structure) as an ID of the lock. The application probes used in the latter measurement output the line-number information that indicates where the lock-request was invoked (a parameter for the lock function) as an ID of the critical section.

Analysis of the obtained event trace gives us the detail of the lock-waiting times of the MySOL threads during the execution of the benchmark transactions. The bottleneck-lock can be identified from the lock-waiting times accumulated on a lock to lock basis; the lock on which threads were mostly spent their execution time to wait the lock acquisition should be the bottleneck. In a similar way, the critical section (CS) that formed the bottleneck can be identified from the result of lock-waiting times accumulated on a CS to CS basis.

3.3 Results of the Baseline System

3.3.1 Lock to lock basis

Figure 9 shows the lock-waiting times normalized as a per-transaction basis obtained from the baseline system in 16-CPU configuration. The top five members are separately displayed and the other members, which accounted for just a fraction of a percent of the total lock-waiting time, are displayed as the sum of them.

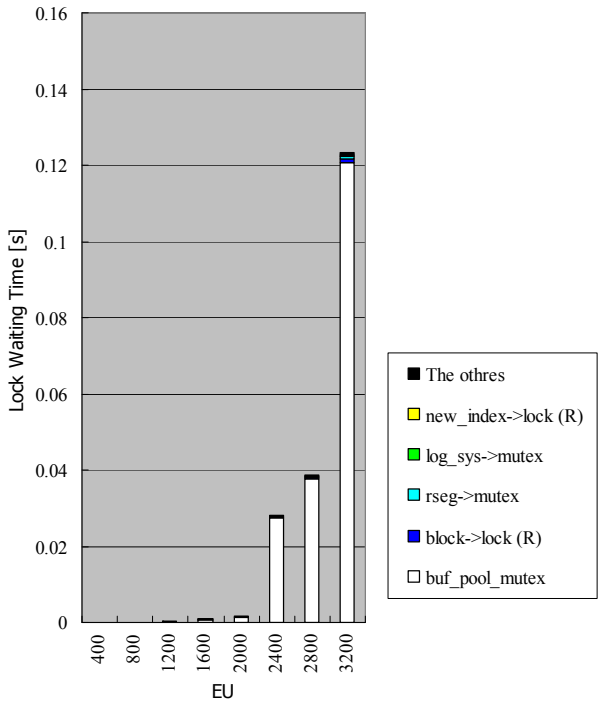


Figure 9: Variation of lock-waiting times detail for a 16-CPU case of the baseline system in a lock to lock basis.

As can be seen from the figure, total waiting time became noticeable in the region where the load intensities (EU) were more than 2000, and waiting times for `buf_pool_mutex` were the major portion of the lock-waiting times in that region. The results of total lock-waiting times are consistent with Figure 6 in that throughputs were saturated in the region where total lock-waiting times were prominent. The results indicate that the `buf_pool_mutex` was the bottleneck of the system, meaning that performance tuning to improve scalability should be focused on the critical sections enforced by the `buf_pool_mutex`.

3.3.2 CS to CS basis

In order to pinpoint the bottleneck critical section, detail of the waiting times on the critical sections enforced by the `buf_pool_mutex` was investigated in a CS to CS basis through the measurement based on event-tracing. The results for the baseline system in 16-CPU configuration in a CS to CS basis are shown in Figure 10.

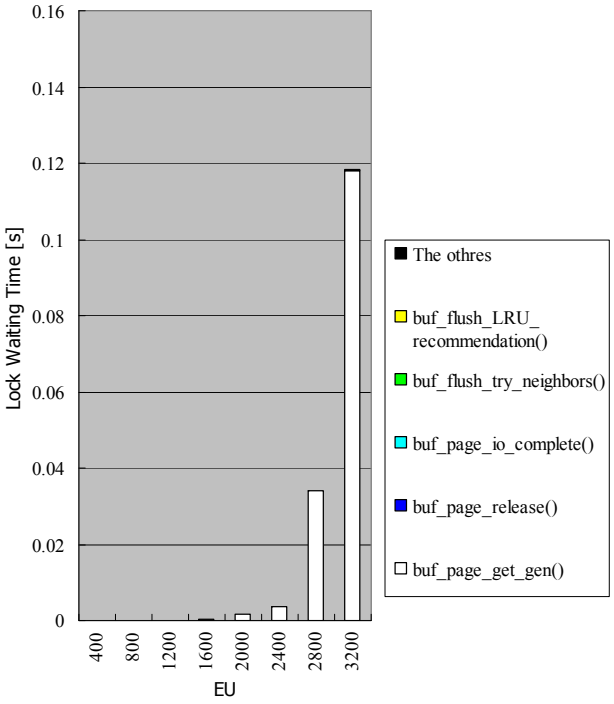


Figure 10: Variation of waiting times detail for a 16-CPU case of the baseline system in a critical-section to critical-section basis. This graph shows that the critical section in the buf_pool_get_gen() function was the bottleneck.

It is noteworthy of special mention that the waiting time on a particular critical section stood out, which means that only one critical section had a major effect on the system performance. Performance tuning should be focused on the bottleneck critical-section, and the others had to be paid little attention at the moment.

4. BOTTLENECK ELIMINATION

The measurement results presented in the previous section have been applied to the performance tuning, which is a repetition of bottleneck identification and solution. This section describes the performance tuning conducted to the baseline system in chronological order.

The approaches for the performance tuning were: 1) the bottleneck lock will be replaced with a set of fine-grained locks and 2) the operations executed in the bottleneck critical-section will be implemented by a particular atomic instruction of the CPUs.

4.1 Fine-Grained Lock

4.1.1 Usage of the bottleneck-lock

The usage of the bottleneck-lock (`buf_pool_mutex`) was revealed through the inspection of the MySQL source code; the

bottleneck-lock (`buf_pool_mutex`) was used to protect the structure of data named `buf_pool_struct` (Figure 11). The main task of the bottleneck critical-section, which was in the `buf_page_get_gen()` function, was also inspected; the specified buffer-block will be searched in and acquired from the hash-table named `page_hash` which is in the `buf_pool` structure.

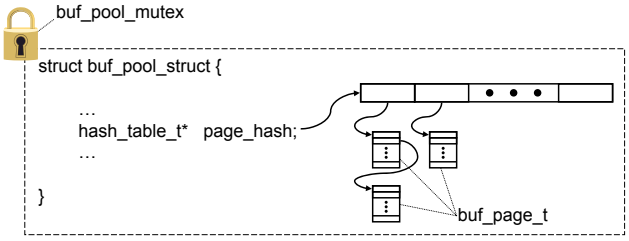


Figure 11: Coarse-grained lock used to protect the structure variable of buf_pool as a whole.

4.1.2 *Introducing a fine-grained lock*

Based on the investigation of the bottleneck-lock and bottleneck critical-section described in the previous section, the course of the performance tuning was to be determined; replacing the bottleneck-lock with fine-grained locks.

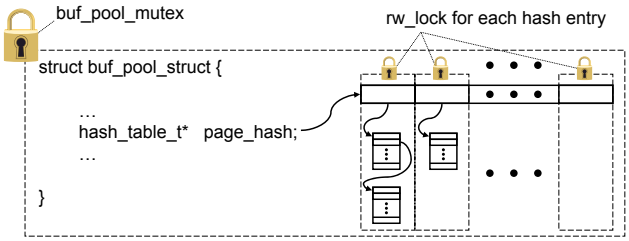


Figure 12: The modified lock structure; a set of fine-grained locks each of which protect the corresponding hash-entry in the page_hash structure.

Figure 12 illustrates the outline of the fine-grained lock; the set of the rw_locks for each hash-entry were added. In accordance with the change in the lock structure, the critical sections had to be modified as follows.

1. The acquisition and release of the exclusive-lock were added before and after the procedure that carries out the insert to and the delete from the corresponding hash entry of `page_hash` (Figure 13 (a)),

2. the acquisition and release of the shared-lock were added before and after the procedure that carries out the search in the corresponding hash entry of `page_hash` (Figure 13 (b)), and

3. acquisition and release of the bottleneck-lock at the bottleneck critical-section were omitted.

4.1.3 *Emergence of next bottlenecks*

The performance tuning with the fine-grained lock has succeeded to eliminate the `buf_pool_mutex` originated bottleneck and has resulted, however, in the emergence of new bottleneck-locks; they are `block->mutex` and `btr_search_latch` (Figure 14).

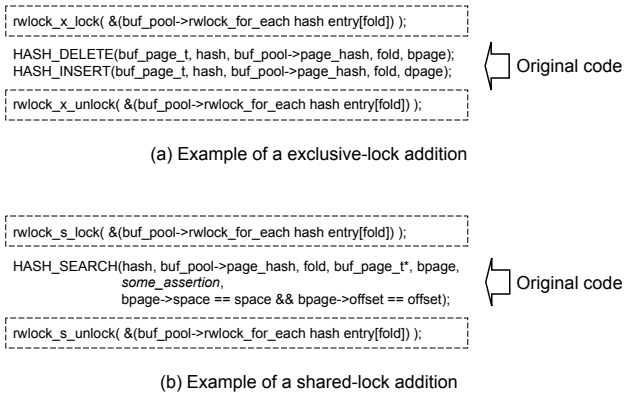


Figure 13: The mutual exclusion enforced with the exclusive or shared-lock was added to the access of the corresponding hash entry of page_hash.

The critical sections enforced by the `btr_search_latch` were possible to tune in the similar way to that for the `buf_pool_mutex`, because they were mainly used for ensuring the mutual exclusion of search/insert/delete of the hash table named `btr_search_sys->hash_index`, which exists to help B-Tree search operations in MySQL.

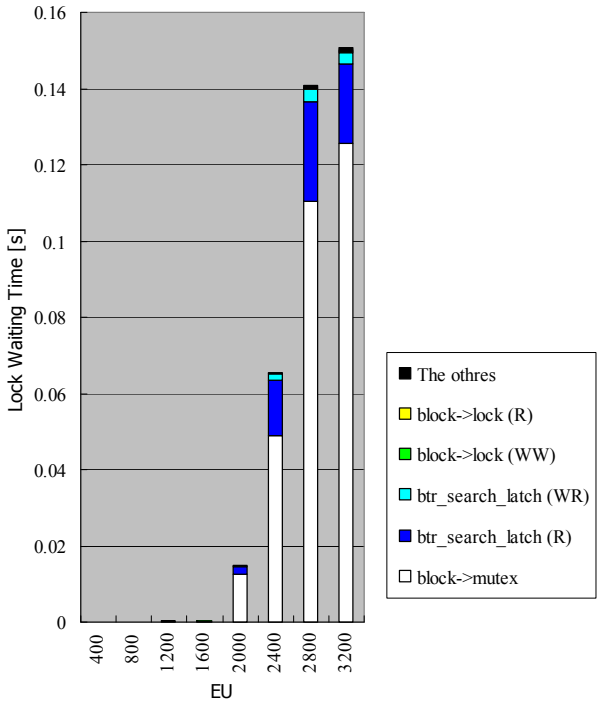


Figure 14: Variation of lock-waiting times detail for the Improved System (16-CPU case) in a lock to lock basis.

4.2 Atomic Instruction

4.2.1 *Usage of the bottleneck-lock*

The `block->lock` related bottleneck critical sections in the improved system were identified in a similar way to that described in Section 3.3.2; Figure 15 shows the detail of the lock-waiting

times in the CS to CS basis. The bottleneck critical-sections indicated by the graph were investigated by inspecting the MySQL source code and clarified that their task were increment or decrement of the `buf_fix_count` member variable in the `buf_page_struct`.

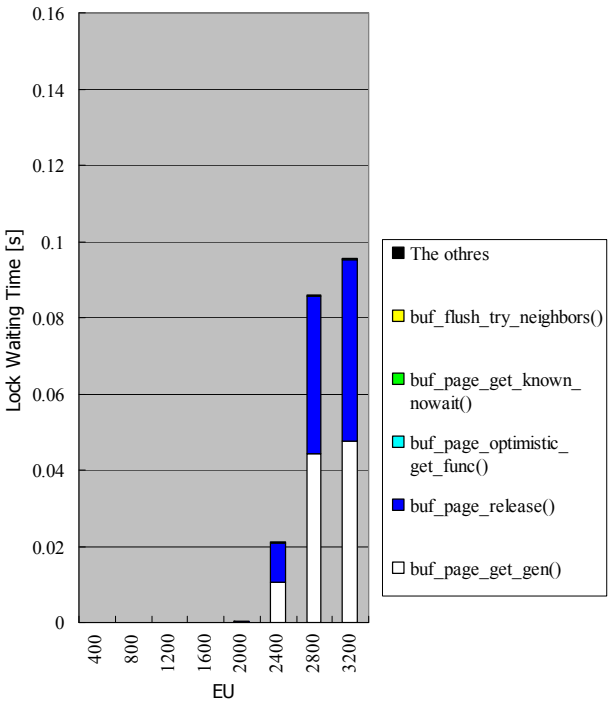


Figure 15: Variation of waiting times detail for a 16-CPU case of the improved system in a critical-section to critical-section basis. This graph shows that the critical sections in the buf_page_get_gen() and the buf_page_release() functions were the comparable bottlenecks.

4.2.2 *Using atomic instructions of the CPU*

Since the task of the bottleneck critical-sections are just the increment or decrement of a particular counter, there was a possibility to avoid using critical section; the increment/decrement operations were to be carried out by a particular atomic instruction of the CPU. The reality is, however, somewhat complicated; the value of the `buf_fix_count` member variable is often changed with other member variables such as `io_fix`, `flush_type`, `state`, and `access_time`, and thus, the values of these member variables should be changed atomically.

In order to change those member variables atomically and simultaneously, several points of the program had to be modified. The first step was that those member variables were packed into a 64-bit data, which can be handled by an atomic instruction of the CPU such as CMPXCHG [17]. Then, the source code were modified to use a particular CPU's atomic instruction to access the related member variables; an example of the modifications are shown in Figure 16.

4.3 Evaluation of Performance Improvement

4.3.1 *Lock-waiting time*

Thanks to the performance tuning described in the previous subsections, bottleneck locks has been successfully eliminated. Figure

```
block_mutex = &((buf_block_t*) bpage)->mutex;
mutex_enter(block_mutex);
   bpage->buf_fix_count++;
   must_read = buf_page_get_io_fix(bpage) == BUF_IO_READ;
   access_time = buf_page_is_accessed(bpage);
mutex_enter(block_mutex);
```

(a) Example of an original code (rewrote for conciseness)

```
old_value = bpage.sbits;
do {
   new_value = old_value;
   new_value.sbits.buf_fix_cont++
   must_read = new_value.sbits.io_fix == BUF_IO_READ;
   access_time = new_value.sbits.access_time;
while ( CompareAndSwap( &bpage.sbits, &old_value, new_value) != SUCCESS );
```

(b) The modified code by using CompareAndSwap operation

Figure 16: An example of the program modifications to use the CAS (compare-and-swap) instruction (CMPXCHG in x86 CPUs) for the change of the related member variables.

17 exhibits the lock-waiting time of the tuned system in the lock to lock basis. By comparing the graph with that of Figures 9 and/or 14, it is obvious that lock-waiting times has become negligible. These results clearly demonstrate that the tunings have contributed to the elimination of the bottleneck lock.

4.3.2 *Throughput and CPU utilization*

Performance characteristics of the tuned system are illustrated in Figure 18, which shows the relationship between throughputs and CPU utilizations of the tuned system with varying the load intensity, which is the same manner as that in Figure 4. The series of bottleneck eliminations has made an improvement in maximum throughput as follows. Maximum throughput of original and baseline system with 16 CPUs measured by the DBT-1 benchmark program were about 580 BT/s and 800 BT/s respectively, and that of the tuned system was about 930 BT/s; the bottleneck elimination produced the performance improvement of 1.6 times from the original system, which is remarkable especially because the change was made only in the software and not in the hardware.

From the scalability viewpoint, however, those improvements are by no means satisfactory as the throughput achieved with the 16-CPU system was less than that that of the 12-CPU system. In addition, there is a little CPU jump phenomena in the graph in the 16-CPU results; CPU utilization increased from about 1100% to 1600%, which was not deserved the increase in the throughput from 870 to 930 [BT/s]. This shows that there was a next scalability-bottleneck.

Since the lock-waiting times in the tuned system (Figure 17) were negligible magnitude, the scalability bottleneck of the tuned system was not supposed to be lock-related factors. Next subsection tries to identify the scalability bottleneck of the tuned system.

4.3.3 *Investigation on the next bottleneck*

By going back to the basics of typical bottlenecks in SMP (Symmetric Multiprocessing) systems as illustrated in Figure 19, a reasonable inference on the scalability bottleneck of the tuned system was made; access conflicts on the shared bus were the cause. In order to verify this inference, activities at the hardware layer were measured with the CPU-built-in performance counters.

Similar way to that used to obtain the result shown in Figure 7 was applied in the measurement of the tuned system and produced the results shown in Figure 20. They clearly indicated that access conflicts at the shared-bus occurred in the 16-CPU and 12-CPU

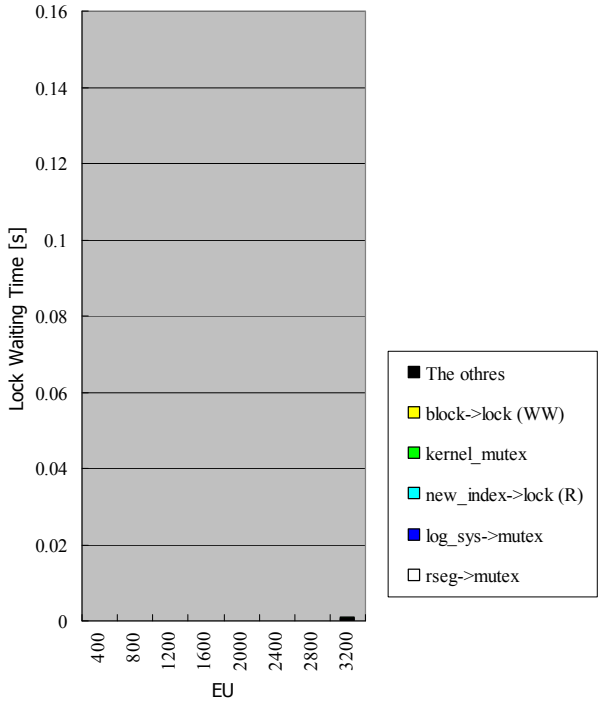


Figure 17: Variation of lock-waiting times detail for a 16-CPU case of the tuned system in a lock to lock basis.

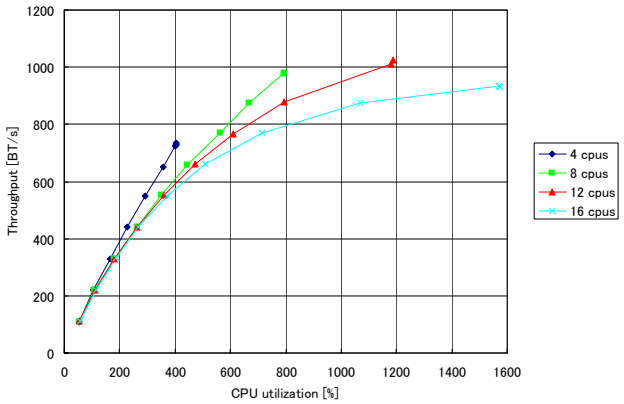


Figure 18: Throughput vs. CPU utilization of the tuned system.

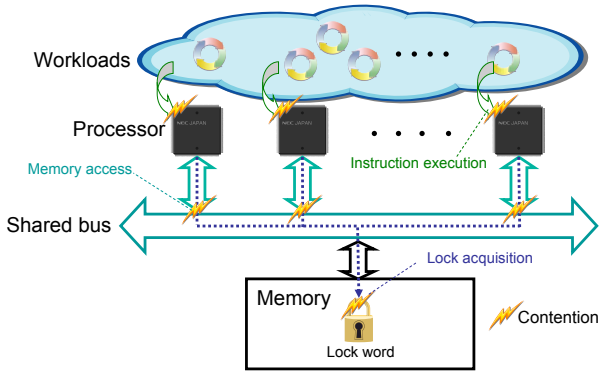


Figure 19: Possible bottlenecks in a SMP system.

systems because the graphs (b) and (c) show that each performance index increased with the increase in the load intensity. Figure 7, on the other hand, shows that those indices were virtually constant even if the load intensities were changed.

It is very possible that the increase in the degree of shared-bus conflict was due to the increase in the number of bus transactions per instruction. Although the difference between graphs (a) in Figures 7 and 20 seems small, the difference can be very influential in the shard-bus activity if the shared-bus utilization were close to its saturation point. After all, feasible assumption here is that the conflicts at the shared-bus leaded to the slower instruction execution of the CPUs, which adversely affect the system performance. The assumption was possible to be verified from the benchmark execution on a system that uses the CPUs with larger internal cache, because larger CPU-internal cache promises the effect of the reduction in the bus-transaction frequency.

The measurement results [4] are shown in Figure 21, which illustrates the performance characteristics of the system using processors with larger internal cache; many the system had CPUs, more the throughput were achieved. This means that maximum throughput (saturation point) can be increased by adding CPUs to the system, which means that structural scalability is not impaired, as long as the number of CPUs is less than 24.

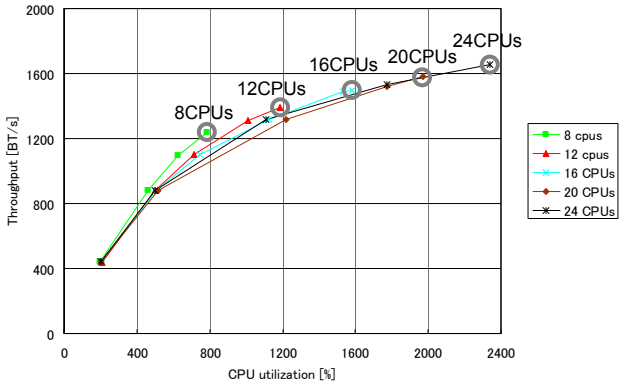


Figure 21: Throughput vs. CPU utilization of a system using processors with larger internal cache.

5. DISCUSSION

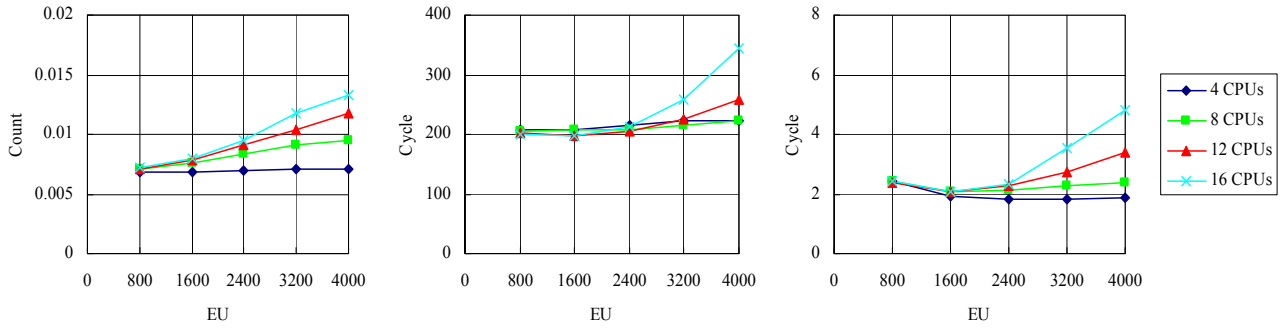
5.1 Nature of Bottlenecks

5.1.1 Universal principles

Although bottlenecks in many-core systems seems much more complicated than those in traditional systems, fundamental natures of both bottlenecks, however, are the same; only a few primal bottlenecks are visible to us, meaning that a few factors affect the system in its performance. When the primal bottlenecks are successfully eliminated, the next bottlenecks will emerge (become primal) and come to affect the system performance. This requires us to tune the system performance step by step.

As to the performance tuning presented in Section 4, the biggest bottleneck of the first (baseline) system was the `buf_pool_mutex`

[4]The measured system had four Intel X7460 CPUs, each of which had six processor-cores and 16 Mega-Byte of internal cache memory, were driven by a 2.66 GHz clock and had a FSB driven by a 1066MHz clock. The main memory capacity was 48 Giga-Byte.



(a) Number of Bus Transactions per Instruction

(b) Clock Cycles per Bus Transaction

(c) Clock Cycles per Instruction

Figure 20: Performance indices at the hardware layer (tuned system).

related critical section, and then the `block->lock` and `btr_search_latch` related critical section became the bottleneck when the first bottleneck had been eliminated. This makes us re-realize a fundamental fact that performance tuning is a repetition of the basic operations of bottleneck identification and elimination. An example of this point was the Section 4 in which a bottleneck at the hardware layer emerged after the three lock-related bottlenecks had been eliminated.

5.1.2 Scalability Bottlenecks

Many-core systems have more potential bottlenecks than traditional systems, as Figure 20 shows an example of this tendency; the more cores, the larger performance indices, which reflect the degree of interference for CPU's instruction execution by other CPUs. Since many-core systems are becoming common for IT systems, this tendency suggests that we should progress the methodology for performance evaluation and tuning to address scalability issues appeared in many-core systems. Establishing a well-formed method to identify the bottleneck is the first step in the progress toward promptly solving scalability related performance problems.

As to the lock-related bottlenecks, measurement of the lock-waiting times was useful to identify the bottleneck lock and the critical section(s) enforced by the lock. On the other hand, measurements of hardware related activities were necessary to identify the shared-bus contention derived bottleneck. The series of the performance tuning is an example case of using right tool for the right purpose and is expected as the first step to establish the well-formed method for solving scalability bottlenecks.

5.2 Performance Tuning

5.2.1 Dimension of the bottleneck

Measurement of the lock-waiting times presented in Section 4 showed that 1) a particular lock formed the bottleneck at a time and 2) a few (one or two) critical sections enforced by the bottleneck lock were the bottleneck. This implied the possibility that the modification of a small portion of the program can be effective.

In fact, the InnoDB storage engine had over 40 locks, each of which had ten to twenty critical sections, out of which several critical sections had to be inspected for the performance tuning presented in Section 4. Especially, it was somewhat surprising that just increment/decrement of a counter for buffer management come to form the primary bottleneck. This could be due to using a traditional buffer replacement algorithm that does not aims higher structural scalability but effective utilization of small memory. Since recent server machines have large-capacity memory, buffer replacement algorithm should be built with focusing on the structural scalability.

Of course, the similar portion in other DBMSs does not always form the bottleneck and thus tuning of them does not always produce performance improvement. In particular, commercial DBMSs are believed to be tested thoroughly, which already eliminated that kind of bottlenecks.

The contribution of this paper is not pointing out a scalability bottleneck specific to a particular DBMS but proposing a method for identifying and tuning of scalability bottlenecks based on an event-tracing technique. The author believes that proposed method is applicable to any multi-threaded server systems because it does not rely on prior knowledge on target software and hardware systems.

5.2.2 Another challenge

An important issue in using critical section is ensuring there is no race conditions among the shared variables accessed in the critical sections, which was also important in the program modifications described in Section 4. Since the variable accessed in the bottleneck critical section is also accessed in other related critical sections, the performance tuning was accomplished by modifying several critical sections, which made the correctness issue a little bit complicated.

Fortunately, the source code of the InnoDB storage engine included some assertion statements. When some sort of error occurred in the benchmark executions, the structure of mutual executions was reviewed and reformed to fix the error. Examples of countermeasures include enlarging the critical section of a newly introduced fine-grained lock and/or adding a new lock to ensure the mutual exclusion access to another resource, other than that protected by the newly introduced lock. That is to say, they were used to confirm the correctness of the atomic operations in modifying the program.

Although no assertion error has been occurred on the tuned system during the benchmark executions so far, it does not mean that there are no race conditions in the modified source code. This means that there should be another crucial challenge other than that of identifying the bottleneck; the verification of the correctness condition relating to critical sections. After all, the author thinks that it will be a promising approach that combining a tool for identifying the bottleneck and a tool for verifying the correctness of modified critical section such as FUSION [23] and/or CoBe [22].

6. CONCLUSIONS

Owing to the emergence of the many-core era in recent years, structural scalability is coming to be the most important property in performance issues of IT systems. Since many-core systems can behave in unfamiliar ways for us and can pose us a performance problem that we have never come across, we need a well-formed methodology to address the performance issues.

In order to address the scalability issues, this paper proposes a method for identifying the scalability bottlenecks based on an event-tracing technique. One of the main features of the method is that the lock-waiting times of the DBMS threads are measured and used to identify the bottleneck lock and critical section. The method has been applied in the evaluation of MySQL with the InnoDB storage engine, one of the most widely used open source DBMSs, and has succeeded to identify its lock-related scalability bottlenecks. This case should imply the availability of the method.

This paper also demonstrates the possibility of eliminating the bottleneck by breaking down the granularity of the bottleneck-lock and by adopting an atomic instruction of the CPU. By using the fine-grained locks in hash-table manipulation and the atomic instruction in buffer-management functions, maximum throughput of the 16-CPU system has increased by 1.6 times, which shows the possibility of the method, as long as lock-related bottlenecks are of concern.

It is noteworthy that these bottleneck identifications and eliminations have been accomplished by the author who was unfamiliar with the source code of MySQL and InnoDB. This means that the proposed methods can be applied to other systems with relative ease, and thus, they should be a meaningful first step for establishing a well-formed procedure to address scalability issues.

7. ACKNOWLEDGMENTS

The author would like to thank Professor Akira Fukuda of Kyushu University for his support, and the anonymous reviewers for their useful comments to improve this paper.

8. REFERENCES

[1] The community enterprise operating system. http://centos.org/.

[2] Database test suite. http://osdldbt.sourceforge.net/.

[3] The linux kernel archives. http://www.kernel.org/.

[4] Mysql :: The world's most popular open source database. http://www.mysql.com/.

[5] Oprofile. http://oprofile.sourceforge.net/.

[6] Oss iPedia (Japanese web-site). http://ossipedia.ipa.go.jp/capacity/EV0612260303.

[7] Oss iPedia (Japanese web-site). http://ossipedia.ipa.go.jp/capacity/CS0612210243.

[8] Sun announces mysql 5.4: Up to 9016-way x86 servers and 64-way cmt servers. http://www.mysql.com/news-and-events/generate-article.php?id=1602.

[9] Tpc-w. http://www.tpc.org/tpcw/.

[10] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[11] M. Blasgen, J. Gray, M. Mitoma, and T. Price. The convoy phenomenon. *ACM SIGOPS Operating Systems Review*, 13(2):20–25, 1979.

[12] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, and A. Vainshtein. Optimal strategies for spinning and blocking. *Journal of Parallel and Distributed Computing*, 21(2):246–254, 1994.

[13] A. Bondi. Characteristics of Scalability and their Impact on Performance. In *Proceedings of the 2nd International Workshop on Software and Performance*, pages 195–203. ACM, 2000.

[14] T. Horikawa. TinyTOPAZ: A Hybrid Event Tracer for UNIX Systems. In *SPECTS '99: The Proceedings of the 1999 Symposium on Performance Evaluation of Computer and Telecommunications Systems*, pages 203–210, 1999.

[15] T. Horikawa. A Framework for Performance Evaluation Based on Event Tracing. *IPSJ Journal*, 42(1):68–78, 2001.

[16] T. Horikawa and A. Fukuda. A Method for Analysis and Solution of Scalability Bottleneck in DBMS. In *SoICT 2010: The Proceedings of the 2010 Symposium on Information and Communication Technology*, pages 139–146. ACM Chapter Vietnam, 2010.

[17] Intel Coporation. *Intel(R) 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M*.

[18] M. Ji, E. W. Felten, and K. Li. Performance measurements for multithreaded programs. In *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS joint International Conference on Measurement and Modeling of Computer Systems*, pages 161–170, New York, NY, USA, 1998. ACM.

[19] R. Johnson, M. Athanassoulis, R. Stoica, and A. Ailamaki. A New Look at the Roles of Spinning and Blocking. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 21–26. ACM, 2009.

[20] R. Johnson, I. Pandis, and A. Ailamaki. Critical Sections: Re-Emerging Scalability Concerns for Database Storage Engines. In *Proceedings of the 4th International Workshop on Data Management on New Hardware*, pages 35–40. ACM, 2008.

[21] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 24–35. ACM, 2009.

[22] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium*, pages 13–22. ACM, 2009.

[23] C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 328–342, 2010.