Aspect-oriented Application-level Scheduling for J2EE Servers

Kenichi Kourai Hideaki Hibino * Shigeru Chiba

Tokyo Institute of Technology

{kourai,hibino,chiba}@csg.is.titech.ac.jp

Abstract

Achieving sufficient execution performance is a challenging goal of software development. Unfortunately, violating performance requirements is often revealed at a late stage of the development. Fixing a performance problem at such a late stage is difficult in terms of cost and time. To solve this problem, this paper presents *QoSWeaver*, which provides aspect-oriented application-level scheduling. QoSWeaver weaves scheduling code written in an aspect into application code. The scheduling code gets an application thread to voluntarily yield its execution to implement a scheduling policy. The idea of scheduling at the application level is not new, but aspect-oriented programming makes it more realistic by separation of scheduling code. QoSWeaver also provides a *profile-based pointcut generator*, which automatically generates pointcuts for fine-grained scheduling. To investigate the ability of QoSWeaver for implementing practical scheduling policies, we used QoSWeaver for tuning the performance of a river monitoring system named *Kasendas*. For reliable examination, Kasendas was originally developed by an outside corporation and then it was tuned by the authors with QoSWeaver. The authors could successfully improve the performance of Kasendas under heavy workload and the work of the performance tuning was not large.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement

General Terms Performance, Experimentation

Keywords QoS, performance tuning, pointcut generator, case study

1. Introduction

Achieving sufficient execution performance is one of primary goals of software development. However, it is always a challenging goal. For example, a web application may not satisfy its performance requirement but this fact is often uncovered when a stress test is performed at the final stage of software development, or in a worse case, after the application starts servicing to the users. Of course, the performance characteristics of the software should be carefully considered at the stage of architecture design but estimating the actual performance is difficult at that stage.

AOSD 07 March 12–16, 2007, Vancouver, Canada

Fixing a performance problem at such a late stage is difficult in terms of cost and time. Some readers might think that the problem can be fixed by upgrading hardware, but this approach is the last resort because it needs extra cost. The second-best solution would be to improve the quality of service (QoS) for web applications, but it is still a challenge. To exploit scheduling mechanisms provided by operating systems or middleware for controlling QoS, developers must modify web applications, sometimes largely. Such modification may be difficult to finish within a limited time. If the scheduling mechanism provided by operating systems or middleware is not suitable for the web applications, developers can use a different operating system or middleware. However, changing such underlying software requires them to test their software again because they must guarantee that the software system correctly works under the new circumstances. Executing all the test again would take long time.

To solve this problem, this paper presents *QoSWeaver*, which provides aspect-oriented application-level scheduling. QoSWeaver enables changing a scheduling policy for web applications on demand. QoSWeaver weaves scheduling code written in an aspect into application code. The scheduling code gets an application thread to voluntarily yield its execution to implement a scheduling policy. The idea of scheduling at the application level is not new, but aspect-oriented programming (AOP) makes it more realistic by separating scheduling code from applications. AOP prevents application logic from being corrupted when scheduling code is added or changed. This has been a major obstacle to adopt application-level scheduling. In addition, QoSWeaver provides a *profile-based pointcut generator*, which helps developers write aspects for fine-grained scheduling. The pointcut generator automatically generates pointcuts so that the scheduling code is executed at as regular intervals as possible, according to profile information of the execution of web applications.

To examine that QoSWeaver enables implementing a practical non-toy scheduling policy, we used a river monitoring system named *Kasendas*. Kasendas is a web application that periodically collects the water levels of major Japanese rivers and reports the collected data to the public through the web. We then executed the performance tuning of Kasendas so that Kasendas can periodically collect water levels at correct intervals even if a large number of clients simultaneously send requests to visualize the data of water levels. From the viewpoint of thread scheduling, we tried to give sufficient CPU time to the thread for periodically collecting water levels than the other threads for processing requests from clients. For reliable examination, we ordered the initial development of Kasendas to an outside corporation and we only executed performance tuning. We used QoSWeaver and we could successfully implement a scheduling policy that gives sufficient CPU time to the thread for collecting water levels. The work of the performance tuning was not large, compared with the modification of the software design of Kasendas.

* Currently, Hitachi Software Engineering Co.,Ltd.

The rest of this paper is organized as follows. Section 2 explains why fixing a performance problem at a late stage of software development is difficult. Section 3 presents QoSWeaver, which enables application-level scheduling by using AOP. Section 4 illustrates a river monitoring system named Kasendas, which is our case study, and shows an applied scheduling policy. Section 5 reports the results of our experiments to examine the usefulness of QoSWeaver. Section 6 describes related work and Section 7 concludes this paper.

2. Motivating Background

Fixing a performance problem at a late stage of software development is difficult because developers must consider several practical constraints – cost and time. First, the easiest way to fix the problem would be to upgrade hardware so that the software will run fast enough to satisfy the performance requirement. However, upgrading hardware is usually the last resort, which requires clients to pay extra costs. This solution is not realistic at the last stage of the development when the hardware for product run is already installed.

The second-best solution would be to improve QoS by adjusting a scheduling policy. A web application normally processes various kinds of tasks requested from web browsers (*i.e.* users) in parallel. Some kinds of tasks have higher importance while others have lower importance. The QoS of such a web application is often kept acceptable if higher-importance tasks obtain more computing resources such as CPU time than lower-importance tasks. However, this solution is still a challenge. Since modern operating systems provide a scheduling mechanism for controlling QoS, some readers might think that what developers should do is only to slightly modify their web applications to exploit that scheduling mechanism. Unfortunately, the reality is not such a simple thing.

First of all, developers cannot exploit that scheduling mechanism for web applications if the application thread and the kernel thread are not one-to-one mapping. If the mapping between them is changed after an application thread sets its priority to the kernel thread corresponding at that time, the priority of the application thread becomes ineffective. The mapping depends on the thread library used in operating systems. In addition, the software sometimes has to be largely modified to exploit the scheduling mechanism and such modification is not easy to finish within a limited time before the expected shipping date. For example, to use real-time scheduling provided by operating systems, developers may have to rewrite a part of application code to be kernel modules.

Furthermore, scheduling mechanisms provided by operating systems may not be suitable for web applications. For example, the priority scheduling provided by some general-purpose operating systems may not allocate sufficient CPU time to an application thread executing a periodic task with a high priority. If there are too many low-priority threads, a high-priority thread tends to miss its deadline. This problem will be avoided if developers use a different operating system, in particular, a real-time operating system, but changing an operating system at the final stage of software development is not acceptable. According to our previous work, the performance behavior of web applications largely changes if the underlying operating system is changed, even from a general-purpose one to another [11]. Developers must spend long time for testing a whole software system again if they change the operating system. They must guarantee that the software system correctly works under the new circumstances.

Exploiting the QoS mechanism provided by middleware has a similar problem. The standard Java virtual machine (JVM) supports priority scheduling of Java threads, but it does not guarantee the effectiveness. Priorities passed from applications to the JVM are only hints and the effectiveness strongly depends on the implementation of the JVM and the underlying operating system. If the scheduling mechanism provided by the JVM is not effective or not suitable, developers can use another implementation of the JVM, for example, a real-time JVM implementing Real-Time Specification for Java [5]. However, changing the JVM at the final stage is not acceptable as well as changing the operating system. Some web application servers provide mechanisms for controlling QoS, but those mechanisms are often insufficient. For example, when the maximum number of threads is limited to reduce the system load, high-priority threads may not be able to start execution if many low-priority threads are already running.

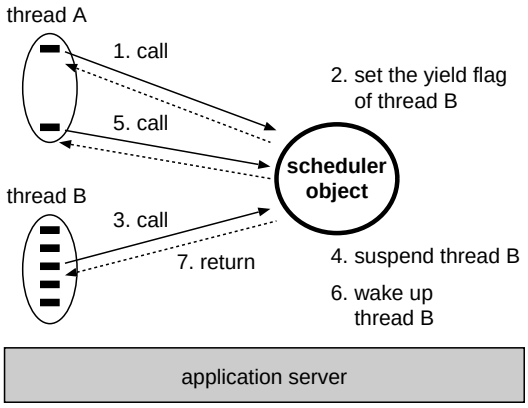

Figure 1. Scheduling mechanism based on thread yielding.

3. Aspect-oriented Application-level Scheduling

To solve the problem described in the previous section, this paper presents *QoSWeaver*, which provides aspect-oriented application-level scheduling. It enables developers to customize a policy of thread scheduling at the application level. In this section, we describe how AOP makes application-level scheduling feasible in practice.

3.1 Application-level Scheduling

The application-level scheduling is implemented by the cooperation among application threads, which voluntarily yield their execution in favor of other threads. Thus a thread must periodically invoke a method on a scheduler object provided by QoSWeaver. The scheduler's method causes the caller thread to yield its execution according to a specified scheduling policy. The suspended thread can be woken up and rescheduled when another thread calls the scheduler's method. The scheduler's method causes a caller thread to yield only if the yield flag of the thread is set. Thus, we can control the scheduling by setting and resetting this flag. Suppose that our scheduling policy is that all other threads are suspended while a particular thread A is running. This policy is implemented as illustrated in Figure 1. If the thread A first calls the scheduler's method, the method does not cause the thread to yield but sets the yield flag of another thread B. This will suspend the thread B when the thread calls the scheduler's method next time. The thread B will not be woken up again until the thread A finishes its execution and the scheduler resets the yield flag of the thread B.

Our application-level scheduling has several advantages, compared with scheduling at a lower level such as the operating system level or middleware level. One advantage is to enable developers to implement various scheduling policies without modifying the underlying systems. The application-level scheduling is independent of the underlying operating system and middleware and hence it does not need to change them. It changes only the schedul-

ing policy of the target applications. Since QoSWeaver schedules only the threads of the target applications, the rest of the threads in the software can obtain at least the same amount of CPU time as they can when QoSWeaver is not applied. Another advantage is to enable developers to develop application-specific schedulers. Such schedulers can use high-level information passed from application threads. For example, if application threads give their roles to a scheduler, the scheduler can give CPU time to these threads according to their role. If a low-level scheduler is used, application threads must translate such high-level information to low-level information such as thread priorities.

The further details of application-level scheduling are described in Section 3.3.

3.2 Separating Scheduling Code into an Aspect

The idea of scheduling at the application level is not new, but it has not been realistic because the developers have to insert scheduling code into their application programs by hand. It is difficult to insert scheduling code at right places, in particular, for average developers. Hence the code they inserted must be later checked by an experienced developer. This work is annoying and time consuming. Moreover, if a scheduling policy requires a thread to frequently yield its execution, developers must insert scheduling code at a large number of places. This causes the application logic to be tangled with scheduling code. Maintaining the tangled scheduling code is difficult. For example, if developers want to change a scheduling policy, they have to remove old scheduling code and insert new scheduling code. This modification is error-prone and hence developing an appropriate scheduling policy by a trial-and-error approach is difficult.

QoSWeaver lets developers to write scheduling code as an aspect and weaves it into application code. Using AOP makes the idea of the application-level scheduling realistic. Since scheduling code is separated from application logic code, it can be written by only a few experienced developers. Other average developers do not have to write scheduling code any more and can concentrate on writing application logic code without being aware of scheduling. Writing scheduling code as an aspect also makes it easy to develop an appropriate scheduling policy by a trial-and-error approach. Since an aspect weaver automatically inserts and removes scheduling code, developers never accidentally corrupt their programs when they change scheduling code.

3.2.1 Profile-based Pointcut Generator

QoSWeaver provides a *pointcut generator*, which automatically generates a set of pointcuts for inserting scheduling code. This tool helps developers define a right set of pointcuts for getting an application to call a scheduler at as regular intervals as possible. Calling a scheduler at regular intervals is desirable to control application threads stably. In particular, fine-grained scheduling needs support by such a tool because an application needs to frequently call a scheduler to yield its execution. It is difficult to manually define pointcuts for such scheduling because the pointcuts must select a large number of join points and a thread must reach those join points in regular intervals. Furthermore, the number of the selected join points should be minimum; otherwise, a scheduler will be called redundantly. Calling a scheduler twice within a single interval is useless. The second call is just a performance penalty.

The pointcut generator generates appropriate pointcuts on a basis of the profile information of the execution of a target application. To do this, the pointcut generator first obtains profile data, which is about when a thread reaches each join point. In our current implementation, the pointcut generator deals with only method calls as join points. To do this profiling, developers must first weave a target application with an aspect that records a caller's method name, a callee's method signature (the method name, the parameter types, and the return type), and the time stamp for each method call. Then they run the target application. Since their application is a web application, they also run a client to send requests to the application. The client sequentially sends requests because we want to know when each single thread calls a method.

To generate appropriate pointcuts from that profile information, the pointcut generator takes two parameters from the developers:

- an ideal interval between adjacent join points selected by pointcuts, and
- the allowed maximum occurrence of join points selected by a single pointcut.

One criterion for the pointcut generator is that the average interval between adjacent join points selected by pointcuts is close to the ideal interval t given from the developers. The pointcut generator generates pointcuts that satisfy this criterion as much as possible. The maximum occurrence m is used to avoid that too many join points are selected.

Developers give different sets of parameters to the pointcut generator and obtain multiple sets of pointcuts, each of which corresponds to each set of given parameters. Then the developers manually choose the best set of pointcuts among the multiple sets they obtained.

$t := ideal\ interval$
$m := maximum\ occurrence$
$exec\_time := total\ execution\ time$

$PC_{all} := a\ set\ of\ possible\ pairs\ of\ pointcuts$
$PC_{gen} := \{\}$
$SLOT := \{0, ..., \lceil exec\_time/t \rceil\}$

for each $i = 1..m$
  $PC_i = \{pc \in PC_{all} \mid |select(pc)| = i\}$
  for each $j \in SLOT$
    $PC_{ij} = \{pc \in PC_i \mid j \in cover(pc)\}$
    $PC_{best} = \{pc \in PC_{ij} \mid |cover(pc) \cap SLOT|\ is\ biggest\}$
    $best\_pc = eval(PC_{best})$
    $PC_{del} = \{pc \in PC_{gen} \mid cover(pc) \subset cover(best\_pc)\}$
    $PC_{gen} = PC_{gen} - PC_{del} + best\_pc$
    $SLOT = SLOT - cover(best\_pc)$
  endfor
endfor

Figure 2. The algorithm of pointcut generation. Function $select$ receives a pair of call and withincode pointcuts. It returns a set of join points selected by the pair. Function $cover$ receives a pair of pointcuts and returns a set of time slots covered by the pair. Function $eval$ receives a set of pairs of pointcuts and returns one of them. $|S|$ is the size of a set S.

3.2.2 Algorithm of Pointcut Generation

Figure 2 shows our algorithm of pointcut generation. The aim of this algorithm is that the generated pointcuts select only one (or as a small number as possible) join point for each time slot. The length of a time slot is the given interval t. The generated pointcuts are chosen among possible pairs of call and withincode pointcuts. Each pointcut does not include wildcards. Thus each pair selects join points representing calls to the same method within the same method body.

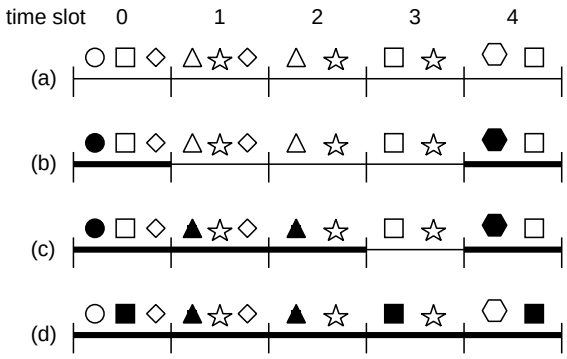

Figure 3. An Example of pointcut generation. Each icon denotes a join point during the profiled execution. Icons filled by black are selected join points.

The algorithm first chooses pairs of pointcuts that selected only one join point during the profiled execution. Let PC_1 be the set of chosen pairs of pointcuts. Then, the algorithm computes a subset of PC_1 that covers as many time slots as possible. Here, covering a time slot means that a join point selected by a pointcut occurs in that time slot. If there are multiple pairs of pointcuts that cover the same time slot, the algorithm chooses one of them. Let PC_{gen} be the computed subset of PC_1.

Then, for the time slots that have not been covered, which are denoted by $SLOT$, the algorithm chooses pairs of pointcuts that selected two join points during the profiled execution. Let PC_2 be the set of chosen pairs of pointcuts. The algorithm computes a subset of PC_2 that covers as many not-covered time slots as possible. If there are multiple pairs that cover the same time slot, the algorithm chooses the pair that covers the most time slots. The elements of the computed subset are added to PC_{gen}. If PC_{gen} contains an element that covers the same time slots as an element newly added to PC_{gen}, then the former element is removed from PC_{gen}. If the former element covers a time slot that the latter element does not cover, it is not removed. The algorithm iterates this choosing process from PC_1 to PC_m, where m is a parameter given by the developers. After the iteration, the pointcut generator generates PC_{gen} as its result.

For example, suppose that the profiled execution is modeled as a sequence of join points in Figure 3 (a) The profiled execution consists of five time slots. Each icon like a circle and a star is a join point. The algorithm first chooses pairs of pointcuts that selected the circle in time slot 0 and the hexagon in time slot 4. Now, these two join points are in PC_{gen} and the time slot 0 and 4 are covered (Figure 3 (b)). Then the algorithm chooses PC_2, which contains two pairs: one that selected two triangles and the other that selected two diamonds. Since the pair selecting the two triangles covers two new time slots (the time slot 0 of the diamond has been already covered), the algorithm adds that pair to PC_{gen} (Figure 3 (c)). At the final iteration, the pair selecting the rectangles is added to PC_{gen}. At the same time, the pairs of pointcuts that selected the circle and the hexagon are removed from PC_{gen} (Figure 3 (d)). The time slot 0 and 4 are covered by the pair selecting the rectangles. The algorithm results in pairs of pointcuts that selected the triangles and rectangles. In this example, all the time slots are covered by those pointcuts and each time slot includes only one join point.

3.3 Details of Application-level Scheduling

Scheduling code woven into applications by QoSWeaver includes synchronization code to yield threads' execution and wake up the threads suspended by thread yielding. In our implementation, the synchronization is achieved by the wait and notify methods in the java.lang.Object class. Adding such synchronization code can cause deadlocks if the original applications also include synchronization code. Suppose that applications use synchronized blocks. A deadlock occurs when a thread in a synchronized block yields its execution and another thread to wake up that thread attempts to enter the block. However, web applications do not often include synchronization code. In particular, the Enterprise JavaBeans (EJB) specification prohibits using thread synchronization. In our case study in Section 4, web applications that QoSWeaver was applied to did not use thread synchronization although they were not EJB applications.

For web applications including synchronization code to prevent deadlocks, QoSWeaver enables each thread that has yielded its execution to execute scheduling code periodically. This is achieved by using the Object.wait method with the timeout argument. If deadlocks occur, the scheduling code can continue that suspended thread and break deadlocks. If not, the scheduling code yields the thread's execution again. Some readers may think that using an independent scheduler thread is straightforward to prevent deadlocks because the thread can always run. However, it is difficult in Java that such a scheduler thread preemptively suspends other threads. Although Java provides the Thread.suspend and Thread.resume methods for thread preemption, these methods are not recommended to use because they are inherently deadlock-prone. Unlike wait method, the suspend method cannot be timed out.

To make it easy to implement scheduling policies, application-level schedulers can rely on the schedulers of the underlying operating system and middleware. For example, when a thread blocks for I/O, the underlying schedulers automatically allocate CPU to another thread if multiple threads are running on the application-level scheduler. The developers do not need to write scheduling code to reschedule threads whenever a thread issues blocking I/O. This mechanism assumes that the underlying schedulers schedule threads in a fair manner. This assumption is satisfied in most operating systems and middleware. If developers want to control threads strictly, they can write scheduling code such that a thread calls a scheduler to temporarily run another thread just before it issues blocking I/O and it suspends the temporarily running thread just after it completes I/O.

4. Case Study

To examine that QoSWeaver enable implementing practical scheduling policies, we executed performance tuning of a web application system with QoSWeaver. The web application that we used is a river monitoring system named *Kasendas*. This section describes the overview of the web application and what scheduling policy we developed for the web application.

4.1 Kasendas: A River Monitoring System

Kasendas is a river monitoring system that collects and reports the water levels of major rivers in Japan to the public through the web. Figure 4 shows a screenshot of its client's view. The web applications that supply such information related to natural disaster should be carefully implemented to be able to work under a large number of simultaneous accesses, known as *flash crowds*. Usually people will not access such a web application but, once a large typhoon approaches, they will rush to the web application for making sure that their local rivers are not flooded. We executed performance tuning so that the software will work under such heavy workload. We chose this application because this work was done for demonstrating our AOP technology within the framework of a research project funded by Japan Science and Technology

Figure 4. The current water levels in Tokyo shown by Kasendas.

server	file type	number	lines
Kasendas	.java files	82	9238
	JSP files	12	1736
	dicon files	15	558
Data generator	.java files	8	646

Table 1. The code size of Kasendas.

Agency, which is studying dependable IT infrastructure for secure life. Since Kasendas is for technology demonstration, the water levels shown by Kasendas were pseudo data produced by the data generator, which emulates sensor nodes that measure the water levels of rivers and provides the data to Kasendas.

To make the results of our experiment reliable, Kasendas was initially developed by an outside corporation with CMMI level 3 [21]. We only received its source files and executed performance tuning. Although we told them the aim of Kasendas, they developed it independently of QoSWeaver. The requirement from us was to build Kasendas with typical open source middleware: such as JBoss application server [12], the Tomcat web container [2], the Struts framework [3], and the Seasar2 container [19]. Table 1 shows the code size of Kasendas. JSP files specify the design of web pages and the dicon files specify the configuration of components. This table does not include third-party libraries and frameworks. The development cost of Kasendas was 8.8 man-month, including tests and the design of web pages.

Figure 5 shows the architectural overview of Kasendas. Kasendas collects the water levels of rivers through web services provided by the data generator periodically, for example, every 15 seconds. To collect the water levels of all rivers, Kasendas sends the same number of requests as rivers to the data generator. The collected data are stored in the PostgreSQL database [17] and the latest data is also kept on memory. Kasendas runs two other applications. One generates a web page showing recent changes of water levels, for example, for the last 12 hours. It reads data from the database and generates a chart of water levels by using the JFreeChart library [16]. This is a heavy-weight application because it accesses the database and produces a PNG image of the chart like Figure 6. The other generates a web page showing the current water levels. It reads data on memory and generates an image like Figure 4.

Kasendas executes these three applications as follows. A timer in Kasendas triggers the execution of the application collecting water levels. A single thread allocated by the timer executes the application at regular intervals. On the other hand, the other two applications are executed when Kasendas receives requests from the clients. Since these applications must be able to process a number of simultaneous requests in parallel, they are multi-threaded. When Kasendas receives a request, it obtains a thread from the thread pool provided in the web application server and the thread executes the requested application.

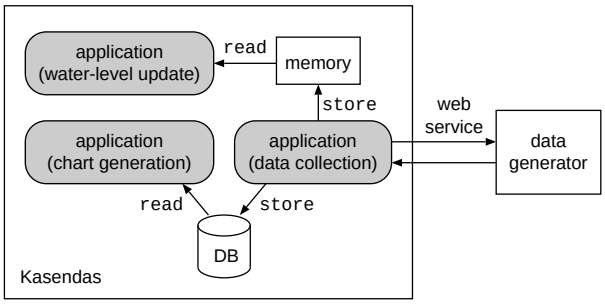

Figure 5. The architecture of Kasendas.

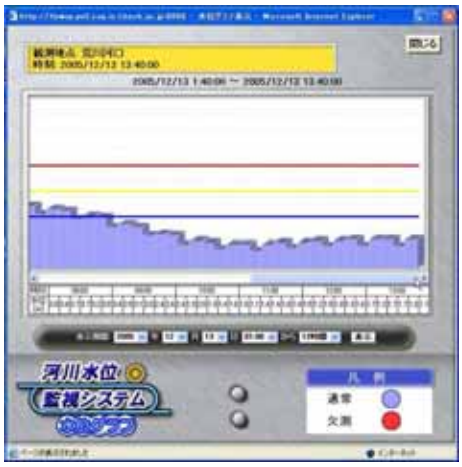

Figure 6. The generated chart of recent changes of water levels in a river.

The initial version of Kasendas that we obtained from the outside corporation was unstable under heavy workload. It frequently failed to collect water levels on time from the data generator. According to our investigation, it became unstable when a number of clients simultaneously access the web page showing a chart of recent changes of water levels. Since generating that page is a heavy-weight task, it consumes a large amount of CPU time and thus it disturbs another application that is periodically collecting water levels from the data generator. This collector application will miss its deadline and lose a part of the water levels at that time. Furthermore, this application continues to collect the rest of the water levels in the next time period because it is not aware of the deadline miss. Thus it fails again to collect the current water levels in the next time period.

4.2 The Applied Scheduling Policy

To fix this performance problem, we used QoSWeaver. The scheduling policy applied to Kasendas was proportional-share scheduling for two groups of low-importance threads of generating a chart and a high-importance thread of periodic data collection. While

the high-importance thread does not run, the scheduling policy runs all the low-importance threads. When the high-importance thread starts to run, the scheduler quickly limits the number of low-importance threads to keep the ratio of the number of threads for each group. The scheduling policy did not consider threads that execute the application providing the current water levels because we could ignore them. They did not have high importance and did not make the system load high because they were light-weight.

The scheduling policy makes a low-importance thread call a scheduler periodically. If the yield flag of the thread is set by the scheduler, the thread is suspended. On the other hand, the scheduling policy makes a high-importance thread call the scheduler when the thread starts the periodic collection of water levels. At this time, the scheduler limits the number of running low-importance threads. We experimentally configured the number to one, which made the behavior of the high-importance thread the stablest. This means that our scheduling policy gives a share of 50% to each group of threads. We did not limit the number to zero because we wanted low-importance threads to run while a high-importance thread was running. To suspend low-importance threads, the scheduler sets the yield flags of all but one low-importance thread. When low-importance threads call the scheduler after that, they are suspended if their yield flags are set. The scheduling policy makes a high-importance thread call the scheduler again when it finishes the execution. The scheduler removes the limitation on the number of running low-importance threads, resets the yield flags of the low-importance threads, and wakes up all the waiting low-importance threads.

To write an aspect that implements this scheduling policy for QoSWeaver, we used our own AOP system named GluonJ [7]. In GluonJ, an aspect is written in XML as glue code [1]. An aspect is woven into web applications when they are loaded into a web application server. The aspect we wrote is shown in Figure 7 and Figure 8.

Figure 7 shows the part of pointcut declaration in our aspect. For simplicity, we omit package names from class names. A pointcut is declared within the pointcut-decl tag. It is named with the name tag and specified with the pointcut tag. In our case, we declared three pointcuts: lowImportance, highImportance, and controlPoint. The lowImportance pointcut selects the execution of the method starting the chart generation. The highImportance pointcut selects the execution of the method starting the periodic collection of water levels. These two pointcuts were written by hand with the knowledge of the source code of Kasendas. On the other hand, the controlPoint pointcut was generated by our pointcut generator. The definition of controlPoint consists of 17 pairs of withincode and call pointcuts, which are concatenated with OROR. ANDAND and OROR correspond to && and || in AspectJ [13], respectively. The controlPoint pointcut selects various method calls during chart generation. We advised these join points so that the scheduler will be called at those join points.

Figure 8 shows the part of advice declaration in our aspect. Advice is declared within the behavior tag. It consists of the pointcut tag and the around or before tag. The pointcut tag specifies a named pointcut declared with the pointcut-decl tag. The around and before tags specify around advice and before advice, respectively. In the around tag, a special variable $$ represents arguments passed to a target method and $_ represents a return value in this context. The advice bodies invoke the methods declared in the PSScheduler class, which is a support class of our aspect, shown in Figure 9. The first around advice is executed when a chart is generated. It calls the scheduler to register the thread to be controlled.

[1] In the latest version of GluonJ, an aspect is written in Java. See `http://www.csg.is.titech.ac.jp/projects/gluonj/`.

```
<pointcut-decl>
  <name> lowImportance </name>
  <pointcut>
    execution(void
      PlaceChartCreatePseudActionImpl.execute(..))
  </pointcut>
</pointcut-decl>

<pointcut-decl>
  <name> highImportance </name>
  <pointcut>
    execution(void CollectorImpl.doObtain())
  </pointcut>
</pointcut-decl>

<pointcut-decl>
  <name> controlPoint </name>
  <pointcut>
    (withincode(Range
       CategoryPlot.getDataRange(ValueAxis))
     ANDAND
     call(Range Range.combine(Range,Range)))
    OROR
      :
  </pointcut>
</pointcut-decl>
```

Figure 7. The pointcut declaration in our aspect.

```
<behavior>
  <pointcut> lowImportance </pointcut>
  <around>
    PSScheduler.startLowImportance();
    $_ = proceed($$);
    PSScheduler.endLowImportance();
  </around>
</behavior>

<behavior>
  <pointcut> highImportance </pointcut>
  <around>
    PSScheduler.startHighImportance();
    $_ = proceed($$);
    PSScheduler.endHighImportance();
  </around>
</behavior>

<behavior>
  <pointcut> controlPoint </pointcut>
  <before> PSScheduler.yield(); </before>
</behavior>
```

Figure 8. The advice declaration in our aspect.

The second around advice is executed when a high-importance thread starts the periodic collection of water levels. It calls the scheduler to control the number of running low-importance threads. The last before advice is executed periodically during the chart generation. It calls the scheduler to yield the current thread.

The ThreadController class used in PSScheduler implements our scheduling algorithm. This class is reusable and the code size is 151 lines. The implementation is as follows:

```
public class PSScheduler {
  private static ThreadController controller =
    ThreadController.getInstance();

  public static void startLowImportance() {
    controller.add(Thread.currentThread());
  }
  public static void endLowImportance() {
    controller.remove(Thread.currentThread());
  }
  public static void startHighImportance() {
    controller.schedule(1);
  }
  public static void endHighImportance() {
    controller.schedule(40);
  }
  public static void yield() {
    controller.yield(Thread.currentThread());
  }
}
```

Figure 9. A support class for our aspect.

- The add method puts the current thread into a run queue of a scheduler if the number of threads in the run queue is under the configured maximum. Otherwise, the method puts the current thread into a wait queue and causes the current thread to yield its execution by invoking the Object.wait method.
- The remove method removes the current thread from the run queue. If the number of threads is under the maximum, the method resets the yield flags of threads in the wait queue and wakes up the threads by invoking the Object.notify method.
- The schedule method moves threads in the run queue to the wait queue if the number of threads in the run queue is above the new maximum specified by the argument. Then, the method sets yield flags of those threads. Otherwise, the method moves threads in a wait queue to the run queue, resets their yield flags, and wakes up those threads.
- The yield method causes the current thread to yield its execution, if its yield flag is set, by invoking the Object.wait method.

We did not specify a timeout for the Object.wait method because Kasendas did not include synchronization code among threads and it was guaranteed that suspended threads were always woken up by other threads.

4.3 Our Experiences

Throughout the development of our scheduling policy, we found that QoSWeaver made the development easy. First, our scheduling policy was not affected by the modifications of the source code of Kasendas, thanks to aspects and our pointcut generator. During one month before the final release of Kasendas, we had to develop the scheduling policy for the intermediate version of Kasendas in parallel while the development team of Kasendas is still testing and fixing bugs. This is because we had to demonstrate Kasendas at a symposium held by our grant sponsor soon after the expected final release date. Since our scheduling policy was implemented by an aspect and its support classes, we could apply our scheduling policy to the final version of Kasendas without manual modifications of the policy. Although pointcut declaration strongly depends on the code of Kasendas, the pointcut generator automatically generated a new appropriate controlPoint pointcut for the final version.

Second, an aspect allowed us to change a scheduling policy without affecting the source code of Kasendas. We developed the best scheduling policy in the following steps. At the beginning, we tried to cause low-importance threads to yield their execution by getting them to sleep during a certain period by the Thread.sleep method. We implemented the scheduling policy that got threads to sleep at join points out of the JFreeChart library. This policy could not control a system load well because the execution of low-importance threads took long time in that library. Next, we changed this policy so as to get threads to sleep at join points within the library. This policy almost worked well, but it sometimes failed to collect water levels at correct intervals when many threads were woken up accidentally at the same time. Finally, we changed this policy so as to use the Object.wait method for thread yielding. This policy could always control thread execution properly. While we modified our aspect and its support classes through these steps, we could not need to modify the source code of Kasendas. In addition, it was easy to change our scheduling policy so as to perform admission control for our experiments described in the next section.

Third, our pointcut generator enabled us to select the controlPoint pointcut for periodic thread yielding without examining the source code of Kasendas in detail. For periodic thread yielding, we had to choose pointcuts that selected join points that a thread reached at reasonable intervals. However, there were too many candidates for pointcuts in Kasendas even if we limited pointcuts to the pair of the withincode and call pointcuts without wildcards. For a web application generating a chart, there were 803 candidates of pointcuts even in the execution profile we obtained. If we did not have the execution profile, there were much more candidates in the source code of Kasendas. It was impossible to select appropriate pointcuts among these enormous candidates by hand. Using our pointcut generator, we only needed to run a target application for obtaining execution profile, which was used to run the pointcut generator with several sets of parameters, so that the best set of generated pointcuts would be experimentally selected.

The development of our scheduling policy was less than one man-month. Our student, which is one of the authors of this paper, found the condition where Kasendas became unstable and developed the best scheduling policy by trial and error. He found the condition in one week, developed a scheduling policy in less than two weeks, and tested and modified it in one week. For comparison, the developers of Kasendas proposed 0.9 man-month for modifying Kasendas for potential performance improvement. Note that the proposed work is not equivalent to our work. It does not include the analysis of performance bottlenecks. The work is only modifying Kasendas to use multiple threads for collecting water levels in parallel from the data generator. Furthermore, the developers could not guarantee that their modification prevents data loss under heavy workload because they did not know the real performance bottlenecks. Since their modification would make the software more complicated, estimating the performance was difficult.

5. Experiments

To evaluate QoSWeaver from the performance viewpoint, we ran our tuned Kasendas and measured its execution performance.

5.1 Three Versions of Kasendas

We ran not only our Kasendas tuned with QoSWeaver but also the original Kasendas without any tuning and another version of Kasendas tuned with admission control. Admission control is a simple scheduling technique for limiting the number of threads concurrently running. Because of its simplicity, it is often used for controlling the concurrency of web application servers. A web application server adopting admission control checks the number of running threads when it receives a new request from a client. If the number of running threads exceeds the limit, the server does not start processing the new request. A main difference between

admission control and our scheduling by QoSWeaver is that admission control can suspend processing a request only when the server starts processing it. Once it starts processing, a thread processing the request is not suspended until it finishes processing. It is never suspended halfway.

The admission control for Kasendas restricted the maximum number of running low-importance threads for generating a chart. It limited the maximum number to one while a high-importance thread was collecting water levels. This policy is the same as the policy of our scheduling except that it is enforced only when a thread starts. Thus, the comparison between admission control and QoSWeaver will reveal a performance benefit of enforcing the policy by suspending a thread halfway through the execution.

For our experiments, we configured the interval at which Kasendas collects water levels to 15 seconds. To generate workloads, we used Apache JMeter [1]. JMeter concurrently sent requests to the web page showing a chart of recent changes of water levels for the last 12 hours, except the experiment in Section 5.2.3. The number of concurrent requests was 40, except the experiment in Section 5.4.2. We did not send requests to the web page showing the current water levels.

To run Kasendas and the data generator, we used two Sun Fire V60x with dual Intel Xeon 3.06 GHz processors, 2 GB of memory, a gigabit Ethernet NIC. These machines ran Linux 2.6.8 as the operating systems, Sun JVM 1.4.2_06, and JBoss 4.0.2 as the J2EE servers. To run JMeter, we used Sun Fire B100x with a single AMD AthlonXP-M 1.53 GHz processor, 1 GB of memory, and a gigabit Ethernet NIC. This machine ran the Solaris 9 operating system and Sun JVM 1.4.2_05. These machines were connected with a gigabit Ethernet switch.

5.2 Effectiveness of Our Scheduling

We examined whether our scheduling could give sufficient CPU time to the thread executing the application periodically collecting water levels.

5.2.1 Time for Collecting Water Levels

We measured the elapsed time from when a high-importance thread starts collecting water levels until it completes the collection. Since this data collection is performed periodically, data loss occurs if the collection does not finish within its interval, which is 15 seconds in our configuration. Our aim is to prevent such deadline misses for the periodic data collection.

Figure 10 shows the time needed for a high-importance thread to collect water levels every 15 seconds. When we used the original Kasendas, we could measure the collection time only four times during 180 seconds. This is because each data collection took long time. The average collection time was 29.5 seconds and every collection time was more than 15 seconds, which is a deadline. On the other hand, our scheduling reduced the average collection time to 5.3 seconds. The collection time was always within 15 seconds and no data was lost. For the admission control, the average collection time was 10.1 seconds, but the collection time sometimes exceeded 15 seconds, for example, at 45 seconds after the start. This means that the admission control could not always prevent data loss. Fine-grained scheduling by QoSWeaver could prevent data loss by giving sufficient CPU time to the thread for collecting water levels.

5.2.2 Number of Running Low-importance Threads

To examine the scheduling behaviors in detail, we measured changes of the number of running low-importance threads for generating a chart. In our configuration, both our scheduling and the admission control give CPU time to the high-importance thread for the periodic data collection by suspending all but one low-importance thread after the data collection is started. The aim of this experiment is to examine how quickly low-importance threads are suspended. The quickness of the thread suspension can affect the collection time of water levels.

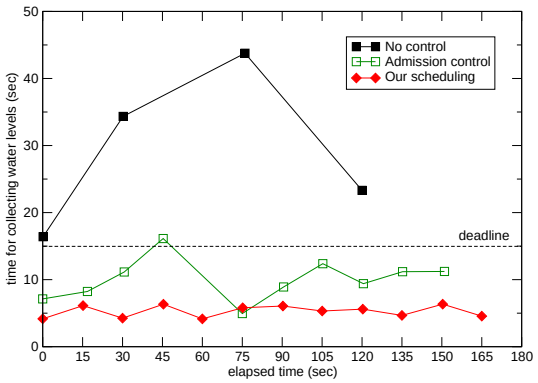

Figure 10. The time needed for a high-importance thread to collect water levels.

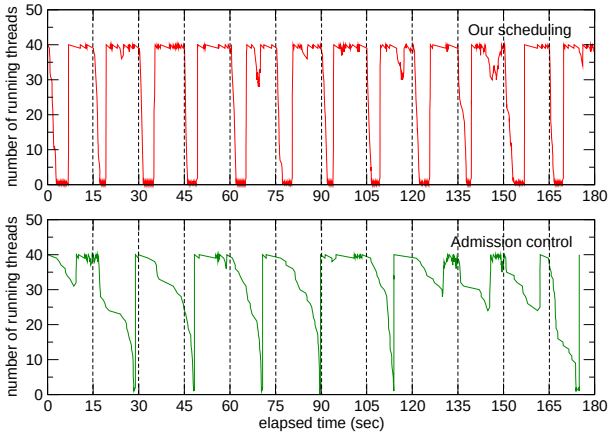

Figure 11. Changes of the number of running low-importance threads.

Figure 11 shows the changes of the number of running low-importance threads. Our scheduling always suspended all but one low-importance thread just after the data collection was started every 15 seconds. The average suspension time was 1.9 seconds. The suspension time means the time from when a high-importance thread calls a scheduler until all but one low-importance thread are suspended. For the admission control, on the other hand, the number of low-importance threads was not reduced to one in several intervals, for example, from time 120 seconds to time 165 seconds. Even when all but one low-importance thread were suspended, the average suspension time was 12.0 seconds. This suspension time is long, compared with the interval of 15 seconds. Since the high-importance thread runs together with low-importance threads, the data collection performed by the high-importance thread tends to be delayed.

5.2.3 Impact of Changing Workloads

We performed the same measurement as Section 5.2.1 when JMeter sent requests to a web page showing a chart of recent changes of water levels for the last 6 hours instead of for the last 12 hours. Under this workload, the application generating a chart obtains the

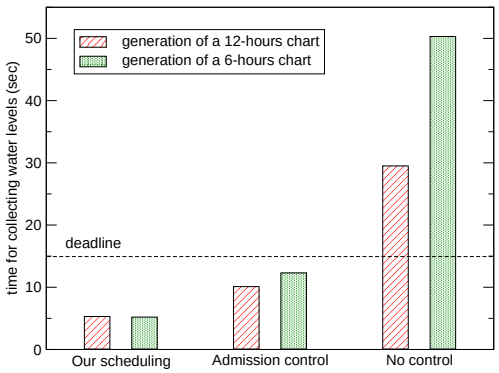

Figure 12. The time needed to collect water levels when the workload is changed.

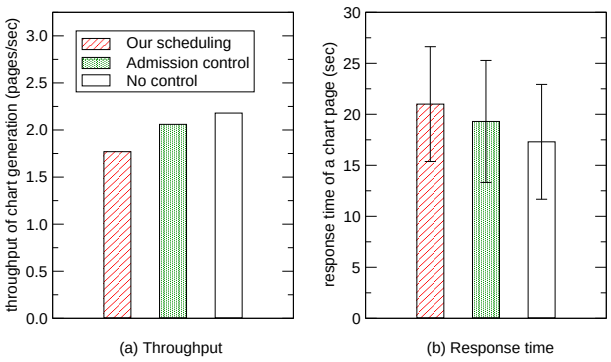

Figure 13. The throughput of the chart generation and the response time of a web page showing a chart.

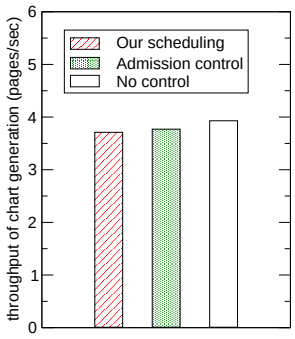

Figure 14. The throughput of the chart generation without performing the periodic data collection.

smaller number of water levels from the database and produces a chart in shorter time. Nevertheless, the system load becomes higher because Kasendas must process more requests per second. The aim of this experiment is to examine how well our scheduling can give sufficient CPU time to the thread for the periodic data collection under heavier workload.

Figure 12 shows the average time needed for a high-importance thread to collect water levels when we changed the workload from the generation of a 12-hours chart to that of a 6-hours chart. Our scheduling kept the average collection time to almost the same under both workloads. On the other hand, when we used the original Kasendas, the average collection time increased from 29.5 to 50.3 seconds and more data were lost. Under the admission control, the average collection time increased from 10.1 to 12.3. The impact of changing workloads was not large for the admission control, but data loss occurred more frequently. From these results, only our scheduling can control Kasendas stably even when the workload is changed.

5.2.4 Influences to Low-importance Threads

To examine how our scheduling affects the performance of low-importance threads, we first measured the throughput of the chart generation, which is executed by low-importance threads. Our scheduling policy temporarily suspends low-importance threads to give sufficient CPU time to a high-importance thread. Therefore, the throughput of the chart generation would be degraded. Figure 13 (a) shows the throughput of the chart generation. Compared with the original Kasendas, the performance degradation under our scheduling was 19% and that under the admission control was 6.5%. This is because our scheduling gave more sufficient CPU time to the high-importance thread than the admission control. In the case of Kasendas, this level of performance degradation was acceptable because our first priority was to prevent data loss for providing reliable information.

Next, we measured the response time of a web page showing a chart for recent changes of water levels. Figure 13 (b) shows the average response time and their standard deviations. Compared with the original Kasendas, the average response time under our scheduling increased by 3.7 seconds and that under the admission control increased by 2.0 seconds. Since the average response time is 17.3 seconds even when we used the original Kasendas, the increase of the response time is not large. Fortunately, their standard deviations were almost the same. This means that the execution of low-importance threads under our scheduling was as stable as those under the other two control methods.

5.3 Scheduling Overhead

In our scheduling policy, QoSWeaver weaves scheduling code into the application generating a chart in a fine-grained manner. At run time, low-importance threads executing the application periodically calls a scheduler's method to yield their execution if necessary. In our experiments, a low-importance thread called the scheduler's method at 19 join points in total during handling one request. If a thread calls the scheduler frequently, the performance penalties can become large.

To examine the scheduling overhead, we measured the throughput of the chart generation, which is executed by low-importance threads, without performing the periodic collection of water levels. Since the periodic data collection causes low-importance threads to be suspended under our scheduling, we stopped the periodic data collection to measure pure overhead of calling the scheduler's method. Figure 14 shows the throughput of the chart generation without the periodic data collection. Compared with the original Kasendas, the scheduling overhead was 6.6% in our scheduling and 4.1% in the admission control. From these results, the overhead due to our fine-grained scheduling was comparable with coarse-grained scheduling in the admission control.

5.4 Usefulness of the Pointcut Generator

We examined the impact of parameters given to our pointcut generator. The pointcut generator takes two parameters: an ideal interval between adjacent join points selected and the maximum occurrence of join points selected by a single pointcut. In the previous sections, we used the `controlPoint` pointcut generated with the ideal interval of 10 ms and the maximum occurrence of 1. For the experiments

Ideal interval	Maximum occurrence	Generated pointcuts	Selected join points
100 ms	1	8	8
	50	9	13
	100	8	83
	200	8	83
10 ms	1	17	17
	50	16	32
	100	17	231
	200	15	309
No pointcut generator		–	248661

Table 2. The numbers of generated pairs of pointcuts and join points selected by them for different sets of parameters.

in this section, we changed the ideal interval to either 10 or 100 ms and the maximum occurrence to 1, 50, 100, or 200.

5.4.1 Generated Pointcuts

Compared with when we used a pointcut that selects all method calls without the pointcut generator, the pointcut generator dramatically reduced the number of selected join points. Table 2 shows the number of generated pairs of call and withincode pointcuts and join points selected by them. The number of join points selected without the pointcut generator was 248661, but the number was reduced to several hundreds at most by using the pointcut generator. As the specified ideal interval got longer or the maximum occurrence got smaller, the number of selected join points was reduced more largely. In addition, the pointcut generator generated the reasonable number of pairs of pointcuts. The number of possible pairs of pointcuts in the execution of the application generating a chart was 803 whereas the pointcut generator selected only 17 pairs of pointcuts from them at maximum. The time needed to generate these pointcuts was 20 seconds at maximum and it was not too long.

5.4.2 Impact of Changing Parameters

First, we examined the intervals between adjacent join points selected by generated pointcuts during the profiled execution. For comparison, we also examined the intervals between all adjacent method calls during the profiled execution. Figure 15 shows the average profiled intervals for different sets of parameters given to the pointcut generator. When we did not use the pointcut generator, the interval was 1.67 ms and it is often too small. When we selected appropriate parameters, the pointcut generator could generate pointcuts so that the average interval approached the ideal one specified. For example, if the ideal interval was 10 ms and the maximum occurrence was 100, the average interval was the nearest to the ideal one. The profiled interval tends to be small as the maximum occurrence became large.

Unfortunately, the standard deviation of each profiled interval was very large. The reason is that the pointcut generator cannot always generate pointcuts so that join points selected by them occur at regular intervals. It depends on the characteristics of applications. Figure 16 plots the time when a program flow reached join points selected by generated pointcuts during the profiled execution. This figure shows that there were no join points in parts of a program flow: time 0.0 to 0.2 second, time 0.6 to 0.7 second, and time 1.6 to 1.9 seconds. In the first part, the application waited for finishing database accesses. In the second part, the application created a large buffered image for a chart. In the third part, the application sent a chart to the client through the network. The pointcut generator could not generate any pointcuts that selected join points during these periods. By contrast, there was a part

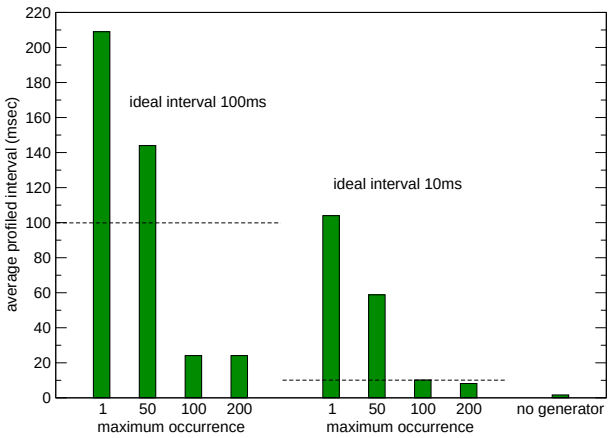

Figure 15. The average profiled intervals between adjacent join points selected by generated pointcuts.

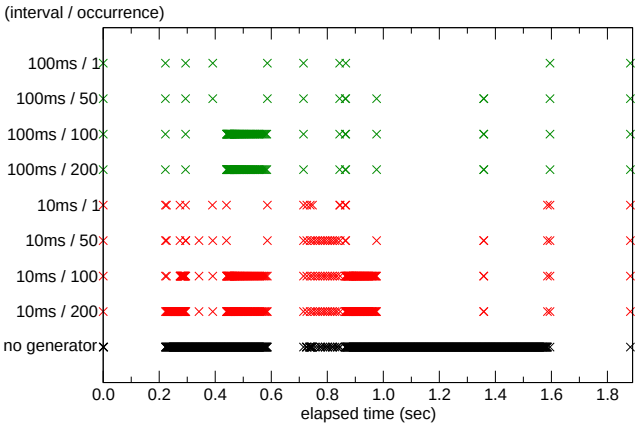

Figure 16. The time when a program flow reaches join points selected by generated pointcuts.

that included too many join points: time 1.0 to 1.6 seconds. In this part, JFreeChart repeated the same processing to generate a chart too many times. The pointcut generator could not generate any pointcuts so that the occurrence of join points was within the specified maximum value. Nevertheless, our scheduling worked well because it did not need to control threads too strictly.

To examine real intervals between adjacent join points selected by generated pointcuts, we measured the real execution time at each join point selected for several sets of parameters. For this experiment, we changed the number of concurrent requests sent by JMeter to 20 or 40 to examine how a server load affects the real intervals. Figure 17 shows that each low-importance thread could call a scheduler's method every 1.1 seconds on average at worst. Under the parameters used in our experiments described in the previous sections, each low-importance thread called the method every 0.5 second on average. This enabled stable control as shown in the previous sections. This figure also shows that the real interval is proportional to the profiled one and the number of concurrent requests. These results show that we can predict the real interval from the profiled one, according to workloads.

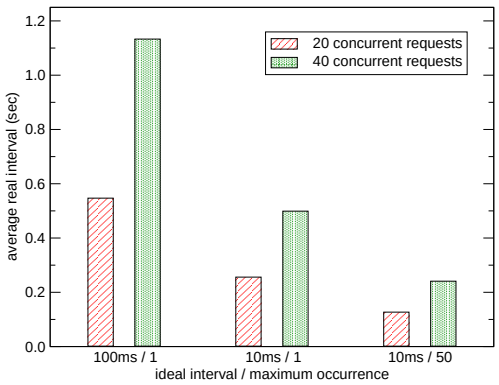

Figure 17. The average real intervals between adjacent join points selected by generated pairs of pointcuts.

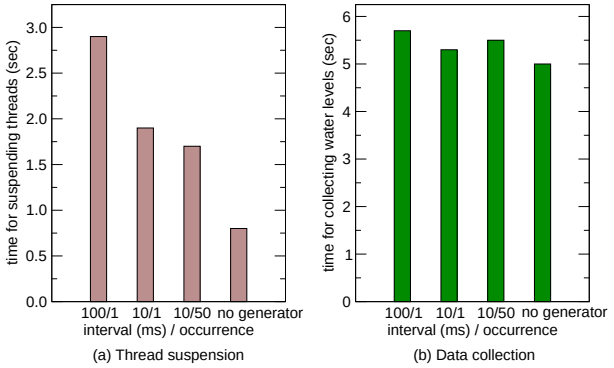

Figure 18. The time for the thread suspension and the data collection for several sets of parameters.

5.4.3 Influences to Execution Performance

To examine how these parameters affect execution performance, we first measured the time needed for suspending low-importance threads and the time needed for periodically collecting water levels. We made JMeter send 40 requests in parallel. The result is shown in Figure 18. The time needed for thread suspension was largely different for each set of parameters. However, the time needed for the data collection was not affected largely by the suspension time. The collection time is between 5 and 6 seconds and sufficient for avoiding deadline misses.

Next, we measured the throughput of the chart generation without the periodic data collection to examine scheduling overhead. Figure 19 shows the result. When we did not use the pointcut generator, the throughput was degraded by 8.4% at maximum. This performance degradation is not large but it may be important for commercial web applications. This fact means that the pointcut generator is useful to reduce scheduling overhead. For our experiments in the previous sections, we selected the ideal interval of 10 ms and the maximum occurrence of 1 so that the scheduling overhead was minimized.

5.5 Summary

The results of our experiments showed that QoSWeaver could implement a practical scheduling policy. Our scheduling policy successfully suspended low-importance threads just after Kasendas started the periodic collection of water levels. Hence, the Kasendas tuned by QoSWeaver could perform the periodic collection stably.

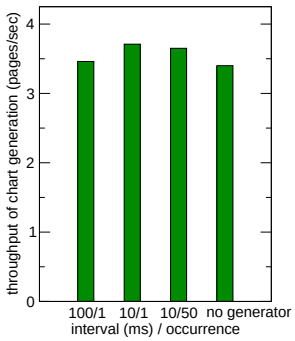

Figure 19. The throughput of the chart generation without the periodic data collection for several sets of parameters.

In addition, its scheduling overhead was acceptable for us. To reduce the scheduling overhead, the pointcut generator was useful. It generated only appropriate pointcuts from enormous candidates of pointcuts.

6. Related Work

Re-QoS [20] uses a QoS aspect package to adapt applications to the real-time systems. A QoS aspect package is a set of aspects and components that provide different QoS policies. In their case study, they showed that Re-QoS could control the deadline miss ratio by admission control of requests. Although Re-QoS uses aspects to add new QoS management like QoSWeaver, it is difficult to use Re-QoS for fine-grained scheduling because the developers have to find appropriate pointcuts by hand. On the other hand, QoSWeaver provides the pointcut generator that automatically generates pointcuts so that scheduling code is executed periodically.

QuO [9] builds QoS management as an aspect and weaves it into the boundary between the application and the middleware. Its aspect language allows the developers to describe adaptive QoS, which changes the behavior of applications according to available system resources. QuO supports distributed object middleware like CORBA and compiles an aspect into a delegate, which is a proxy for calls to remote objects (remote method invocation in Java). Therefore, QoS control can be performed only when an application calls remote objects. This may not be sufficient for applications that do not call remote objects frequently.

Bossa [4] enables a scheduling expert to implement a scheduling policy for operating system kernels independently. It provides a domain-specific language (DSL) to describe a scheduling policy using high-level abstractions. This DSL simplifies scheduler programming and allows the verification of safety properties. To make the kernel raise scheduling events to a scheduler compiled as a kernel module, Bossa inserts the code for raising events into the kernel using aspects [18]. Using DSL and AOP, Bossa allows the developers of web applications to change the kernel scheduler without changing the operating system itself. However, if the developers change the kernel scheduler, they must still spend long time for examining the scheduling behavior to the whole software system.

MS-Manners [8] achieves scheduling called progress-based regulation at the application level. It stops low-importance processes whose progress rate is bad to give sufficient CPU time to high-importance processes. Its platform-independent implementation is to insert calls to the Testpoint function everywhere in a program. This function examines the progress rate and blocks the process for a while if necessary. This mechanism is similar to our mechanism based on thread yielding. However, MS-Manners needs to insert the code for function calls by hand. This decreases efficiency

of development because developers have to consider scheduling at the beginning of the development. In addition, since the scheduling code is inserted into a program, it is difficult to maintain the program.

For UNIX processes, several application-level scheduling mechanisms have been proposed. User-level scheduling [14] and ALPS [15] control UNIX processes from a scheduler process by using signals such as SIGSTOP and SIGCONT. User-level sandboxing [6] restricts the CPU usage of processes by changing the priorities of threads. These mechanisms enable more accurate control than QoSWeaver because they can perform preemptive scheduling. However, it is difficult to apply them to threads instead of processes. User-level scheduling and ALPS distinguish applications by running them in different user accounts. User-level sandboxing enforces a policy by a controller attached to a process. These mechanisms cannot distinguish and control threads. In addition, these preemptive scheduling is not applicable to Java applications. Java does not recommend suspending and resuming threads from another thread, and it does not guarantee the effectiveness of changing the priorities of Java threads using the Thread.setPriority method.

Gatekeeper [10] can transparently apply admission control and request scheduling to servers by inserting a proxy server. Admission control limits the number of concurrent requests to prevent an overload condition while request scheduling changes the order of request handling to improve the response time. The proxy analyzes the contents of requests and applies a scheduling policy to the requests. Inserting the proxy does not need changing any operating systems, middleware, and applications. However, if there are heavy-weight applications whose execution time are long, threads for the applications are not controllable once they start the execution. QoSWeaver enables finer-grained control by weaving scheduling code, for example, at a method-call granularity.

Admission control based on the SEDA architecture [23] enables fine-grained scheduling by dividing an application into several stages [22]. In SEDA, the execution of the application is performed by sending a request to the next stage. Each stage has a queue to receive the request and allows admission control for each request. If an application is divided into enough small stages, fine-grained scheduling is achievable. The advantage of this architecture is that there are no threads stopped by a scheduler unlike our approach. Until a request is admitted, no thread is allocated to it. However, the developers must re-implement applications using stages to fix performance problems when they have already implemented the applications.

7. Concluding Remarks

In this paper, we presented QoSWeaver, which provides aspect-oriented application-level scheduling. The idea of scheduling at the application level is not new, but AOP makes it more realistic by separating scheduling code from applications. In addition, the pointcut generator provided by QoSWeaver helps developers write aspects for fine-grained scheduling. As a case study, we used a river monitoring system named Kasendas, which is a web application system initially developed by the outside corporation. We could successfully implement a practical scheduling policy using QoSWeaver to improve the performance of Kasendas under heavy workload. According to our experiences, QoSWeaver made the development of scheduling policies easy in (1) that the development of scheduling policies and that of Kasendas did not affect each other and (2) that appropriate pointcuts were automatically generated without examining a large amount of source code of Kasendas.

One of our future work is to develop other scheduling policies using QoSWeaver. In this paper, we have developed proportional-share scheduling as our scheduling policy and admission control for comparison. We would like to examine that QoSWeaver is useful in practice to achieve other classes of scheduling. Another direction is to apply QoSWeaver to other web applications. As shown in Section 5.4.2, if an application does not have appropriate join points, QoSWeaver may not control thread execution precisely. We need to examine whether sufficient join points exist in other real applications and whether QoSWeaver is applicable or not to them.

Acknowledgments

This work was supported in part by a Grant-in-Aid for Core Research for Evolutional Science and Technology, from the Japan Science and Technology Agency.

References

[1] Apache Jakarta Project. Apache JMeter. `http://jakarta.apache.org/jmeter/`.

[2] Apache Jakarta Project. Apache Tomcat. `http://tomcat.apache.org/`.

[3] Apache Struts Project. Apache Struts. `http://struts.apache.org/`.

[4] L. Barreto and G. Muller. Bossa: A language-based approach to the design of real-time schedulers. In *Proceedings of the 10th International Conference on Real-Time Systems*, pages 19–31, 2002.

[5] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. The real-time specification for Java. Addison-Wesley, 2000.

[6] F. Chang, A. Itzkovitz, and V. Karamcheti. User-level resource-constrained sandboxing. In *Proceedings of the 4th USENIX Windows System Symposium*, pages 25–36, 2000.

[7] S. Chiba and R. Ishikawa. Aspect-oriented programming beyond dependency injection. In *ECOOP 2005 – Object-Oriented Programming*, LNCS 3586, pages 121–143, 2005.

[8] J. Douceur and W. Bolosky. Progress-based regulation of low-importance processes. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 247–260, 1999.

[9] G. Duzan, J. Loyall, R. Schantz, R. Shapiro, and J. Zinky. Building adaptive distributed applications with middleware and aspects. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 66–73, 2004.

[10] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of the 13th International Conference on World Wide Web*, pages 276–286, 2004.

[11] H. Hibino, K. Kourai, and S. Chiba. Difference of degradation schemes among operating systems. In *Proceedings of DSN2005 Workshop on Dependable Software – Tools and Methods*, pages 172–179, 2005.

[12] JBoss Group. JBoss application server. `http://www.jboss.com/`.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, 2001.

[14] T. Newhouse and J. Pasquale. A user-level framework for scheduling within service execution environments. In *Proceedings of the IEEE International Conference on Services Computing*, pages 311–318, 2004.

[15] T. Newhouse and J. Pasquale. ALPS: An application-level proportional-share scheduler. In *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing*, pages 279–290, 2006.

[16] Object Refinery Ltd. JFreeChart. `http://www.jfree.org/`.

[17] PostgreSQL Global Development Group. PostgreSQL. `http://www.postgresql.org/`.

[18] R. Åberg, J. Lawall, M. Südholt, G. Muller, and A. L. Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 196–204, 2003.

[19] Seasar Foundation Project. Seasar. `http://www.seasar.org/`.

[20] A. Tesanovic, M. Amirijoo, M. Björk, and J. Hansson. Empowering configurable QoS management in real-time systems. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, pages 39–50, 2005.

[21] The Carnegie Mellon Software Engineering Institute. Capability maturity model integration. `http://www.sei.cmu.edu/cmmi/`.

[22] M. Welsh and D. Culler. Adaptive overload control for busy Internet servers. In *Proceedings of the 4th USENIX Conference on Internet Technologies and Systems*, 2003.

[23] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 230–243, 2001.