Application-Level Scheduling Using AOP

Kenichi Kourai[⋆], Hideaki Hibino[⋆⋆], and 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 is a tool suite for developing application-level scheduling using aspects. QoSWeaver weaves scheduling code written in an aspect into web application code. The scheduling code gets an application thread to voluntarily yield its execution to implement a custom scheduling policy. The idea of scheduling at the application level is not new, but aspect-oriented programming (AOP) makes it more realistic by separation of scheduling code. For fine-grained scheduling, QoSWeaver provides a *profile-based pointcut generator*, which automatically generates appropriate pointcuts. 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*, which is a web application system. 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. The cost of the performance tuning was a reasonably small. Furthermore, our approach achieved better performance than other techniques such as admission control and priority scheduling provided by the JVM or Linux. We could implement various policies such as deadlock-aware or adaptive scheduling.

Key words: scheduler, aspect, QoS, performance tuning, case study

1 Introduction

Achieving sufficient execution performance is one of the 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

[⋆] Presently with Kyushu Institute of Technology
[⋆⋆] Presently with Hitachi Software Engineering Co.,Ltd.

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.

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. In the case of web applications, the second-best solution would be to improve the quality of service (QoS), but it is still a challenge. To exploit schedulers 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 policy 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 is a tool suite for developing application-level scheduling using aspects. 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 new 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 one of major obstacles 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.

Our contributions presented in this paper are the followings:

- We propose an AOP-based implementation of application-level scheduling, which is a new application from the AOP perspective.
- We report our experience of applying the application-level scheduling to a fairly realistic web application.
- We show that our approach worked well at least in our case study.

Note that this paper does not propose a new scheduling or resource-allocation algorithm. It proposes using AOP for implementing an application-specific scheduler. Since AOP makes the implementation easier, the use of such a custom scheduler is made practical. Although this paper deals with QoS, we implemented proportional-share scheduling; it is not real-time scheduling. It is out of the scope of this paper whether or not the proposed approach can be used to implement a real-time scheduler.

This paper is an extended version of our previous paper [1]. A difference between the two papers is that this paper presents two more scheduling policies implemented by our QoSWeaver: deadlock-aware and adaptive scheduling. Another difference is that this paper also shows that our application-level scheduler achieved better performance than the priority schedulers included in the standard Java virtual machine (JVM) and the Linux kernel. This paper also presents that the use of our pointcut generator makes a non-negligible impact on the overall performance. Selecting appropriate pointcuts is significant from the performance viewpoint.

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 and describes related work. 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 discusses the applicability of QoSWeaver. Section 7 concludes this paper.

2 Motivation

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, the software sometimes has to be largely modified to exploit that scheduling mechanism. 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 move a part of application code into kernel modules. Even if developers just want to use priority scheduling, which is provided by most operating systems, and raise the priorities of some threads, they may change their applications so that the threads will run with the root privilege. This change may cause security problems and thus the developers must check the entire code of their applications. In addition, developers cannot exploit the scheduling mechanism of the operating system if an application (Java) thread is not always bound to a particular operating-system thread. This depends on the implementation of the application threads. If the mapping between application threads and operating-system threads may change, for example, it is difficult to raise the priority of a particular application thread by changing the priorities of operating-system threads.

Furthermore, scheduling policies 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 [2]. Developers must spend a long time for testing an entire software system again. 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 JVM supports priority scheduling of Java threads, but it does not guarantee its effectiveness. Priorities passed from applications to the JVM are only used as hints. Whether or not the priorities really affect the scheduling depends on the implementation of the JVM and the underlying operating system. If the scheduling policy provided by the standard JVM is not suitable, developers can use another implementation of the JVM, for example, a real-time JVM implementing Real-Time Specification for Java [3]. However, changing the JVM at the final stage is not acceptable as well as changing the operating system. Although some web application servers provide built-in mechanisms for controlling QoS, those mechanisms are often insufficient. For example, if the maximum number of threads is limited to avoid excessive concurrency, high-priority threads cannot start execution when too many low-priority threads are already running.

2.1 Aspect-Oriented QoS Control

Re-QoS [4] 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, the authors showed that Re-QoS could control the deadline miss ratio by admission control of requests. Although Re-QoS uses aspects to add a new QoS management policy like our 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, our QoSWeaver provides the pointcut generator, which automatically generates pointcuts so that scheduling code will be executed periodically.

QuO [5] builds a QoS management system 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 it compiles an aspect into a delegate, which is a proxy for calls to remote objects (remote method invocation in Java). Therefore, QoS control is enforced only when an application calls remote objects. This is not sufficient for applications that do not frequently call remote objects.

Bossa [6] enables a scheduling expert to implement a new scheduling policy for operating system kernels. It provides a domain-specific language (DSL) to describe a scheduling policy using high-level abstractions. This DSL simplifies scheduler programming and allows the formal 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 AOP [7]. Using DSL and AOP, Bossa allows web application developers to change the kernel scheduler without changing the source code of the operating system. However, if the developers change the kernel scheduler, they need to spend a long time for examining the scheduling behavior of the entire software system.

2.2 Previous Approaches to Application-Level Scheduling

MS-Manners [8] achieves process scheduling called progress-based regulation at the application level. It stops low-importance processes when the progress rate is lower than expected and it gives remaining 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 method is similar to our approach. However, MS-Manners requires the developers to manually modify the source code of their applications so that Testpoint will be called. They may also have to modify the source code of the libraries used by their applications. Otherwise, the developers have to write scheduling code together while they are writing the application logic. Either way, their productivity will be decreased. Maintaining the source code becomes difficult because the scheduling code is directly embedded into the source code. To solve these problems, QoSWeaver separates scheduling code into an aspect and it automatically inserts scheduling code at appropriate places in the source code when it performs weaving.

For UNIX processes, several application-level scheduling mechanisms have been proposed. User-level scheduling [9] and ALPS [10] control UNIX processes from a scheduler process by using signals such as SIGSTOP and SIGCONT. User-level sandboxing [11] 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 with different user accounts. User-level sandboxing enforces a policy by a controller attached to a process. These mechanisms cannot distinguish or control threads. In addition, preemptive scheduling is difficult to implement on the standard Java 2 Platform. Its API specification does not recommend to use APIs for suspending and resuming threads by another thread. Also, Java provides the API for changing the priorities of Java threads but its effectiveness is not guaranteed.

Gatekeeper [12] can transparently apply admission control and request scheduling to servers by using a proxy server. The admission control limits the number of concurrently processed requests to avoid excessive concurrency. The request scheduling changes the order of handling requests to improve the response time. The proxy analyzes the kinds of requests and schedules the requests appropriately. Installing the proxy does not need modifying operating systems, middleware, or applications. However, if there are heavy-weight applications, which take long time for processing each request, the threads for those applications are not controllable once they start the execution. They cannot be suspended to decrease concurrency. QoSWeaver enables finer-grained control by weaving scheduling code, for example, at a method-call granularity.

Admission control based on the SEDA architecture [13] enables fine-grained scheduling by dividing an application into several stages [14]. 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 a sufficient number of small stages, fine-grained scheduling is achievable. The advantage of this architecture is that there are no threads suspended by a scheduler unlike our approach. Until a request is admitted, no thread is allocated to it. However, the developers must re-implement their applications using multiple stages to fix performance problems if they have already implemented the applications.

3 Aspect-Oriented Application-Level Scheduling

To solve the problem described in the previous section, this paper presents *QoS-Weaver*, which is a tool suite for developing application-level scheduling by aspects. 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

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. The scheduler's method suspends the caller thread 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 suspends a caller thread only if the *yield flag* of the thread is set. Thus, we can control the scheduling by setting and clearing this flag. Suppose that a scheduling policy is that all other threads are suspended while a particular thread A is running. This policy is implemented as illustrated in Fig. 1. If the thread A first calls the scheduler's method, the method does not suspend the thread 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 clears the yield flag of the thread B.

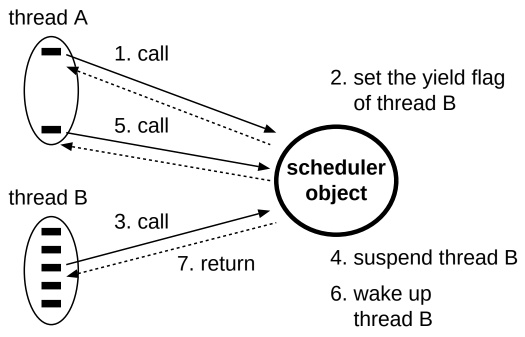

Fig. 1. A scheduling method based on thread yielding.

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. 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 scheduling policy of the target applications. Since application-level scheduling affects 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 application-level scheduling is not applied. Of course, application-level scheduling cannot achieve all kinds of scheduling independently of the underlying schedulers. For example, time-sharing scheduling cannot be developed on top of a batch scheduler. Another advantage is to enable developers to develop application-specific schedulers. Such schedulers can use application semantics given by application threads. For example, if application threads inform a scheduler of their roles, the scheduler can allocate CPU time to these threads according to their role. If a low-level scheduler is used, application threads must translate such high-level semantics to low-level properties such as thread priorities.

3.2 AOP Support for Application-Level Scheduling

The idea of scheduling at the application level is not new but it has not been practical because the developers have to insert scheduling code into their application programs by hand. For example, our web applications presented in Sect. 4 consist of about ten thousand lines of our own code and more than one hundred thousand lines of library code. 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 may 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. AOP makes the idea of the application-level scheduling practical. 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 allows developers to take a trial-and-error approach to develop an appropriate scheduling policy. Developers can easily try various scheduling policies to find the most appropriate one because an aspect weaver automatically inserts and removes scheduling code. They never accidentally corrupt their programs when they change a scheduling policy.

Furthermore, QoSWeaver supports creating a scheduling mechanism for application-level scheduling. Scheduling code written as an aspect consists of a scheduling mechanism and the implementation of a scheduling policy. A scheduling mechanism for application-level scheduling is a set of method calls on a scheduler object from various places in application programs as in Fig. 1. This corresponds to timer interrupts for kernel-level scheduling. A scheduler for kernel-level scheduling is called periodically from hardware. A scheduler for application-level scheduling is called from the join points selected by pointcuts. AOP is used to implement this mechanism.

A *pointcut generator* provided by QoSWeaver automatically generates a set of pointcuts to create such an application-specific scheduling mechanism. 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 for stable control of application threads. In particular, fine-grained scheduling needs such a tool support because an application needs to frequently call a scheduler to yield its execution. It is difficult to manually define pointcuts for fine-grained 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 join points selected by pointcuts 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.

Figure 2 illustrates the architecture of QoSWeaver. QoSWeaver consists of two tools: an AOP system and a pointcut generator. QoSWeaver receives a web application and weaves a profiling aspect, which is provided by QoSWeaver, into it using the aspect weaver. Then QoSWeaver deploys the extended web application for profiling on a web application server. If developers run it, the profiling aspect records its execution profile. From the execution profile that the aspect recorded, the pointcut generator generates appropriate pointcuts for efficient application-level scheduling. Then, developers write a scheduler aspect by using the pointcuts. A scheduling policy is implemented as advice, which is executed at scheduling points specified by the pointcuts. In Fig. 1, it is implemented by the scheduler object. Finally, QoSWeaver weaves this aspect into the original web application and deploys the resulting web application on the server.

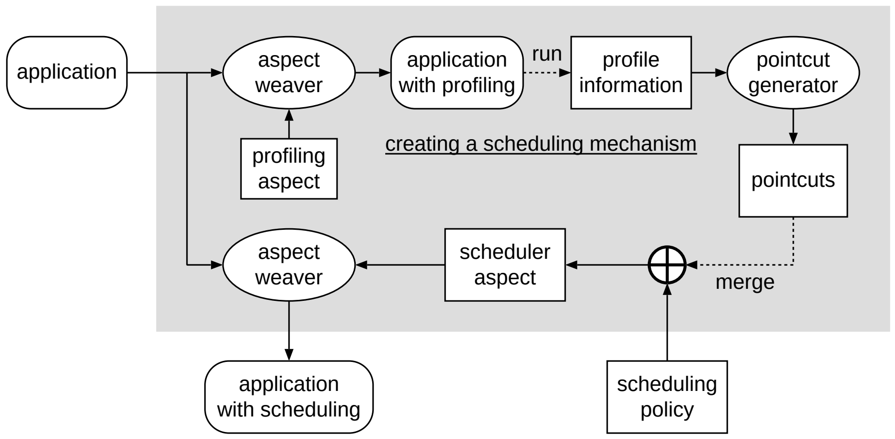

Fig. 2. The architecture of QoSWeaver.

Note that QoSWeaver does not directly support the development of a scheduling policy itself. It only generates appropriate pointcuts for creating a custom scheduling mechanism. Details of a scheduling policy have to be chosen by developers. For example, how priorities are assigned to tasks depends on the user requirements for the web applications. The maximum number of threads concurrently running for each task depends on scheduling algorithms. These parameters should be selected by experiments. QoSWeaver helps developers to select those parameters by using a trial-and-error approach.

3.3 Profile-Based Pointcut Generator

The pointcut generator generates appropriate pointcuts on the basis of the profile information of the execution of a target application. The profile data collected by QoSWeaver is when a thread reaches each join point. Our current implementation of the pointcut generator deals with only method calls as join points. QoSWeaver first weaves a target application with a profiling 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 developers run the target application into which the profiling aspect has been woven. Since their application is a web application, they also run a client to send requests to the application. The client sequentially sends requests to minimize the disturbance by the outside because we want to know when each single thread calls which method.

The execution profile should be the representative of the behavior of a target web application. If the behavior of an application is largely different from the execution profile, the application could not call the scheduler at regular intervals. However, it is difficult to guarantee that the obtained execution profile is a representative because this fact strongly depends on the characteristics of applications. A guideline for obtaining a representative execution profile is to send the most common request to the target web application. The developers should know what the most common request is for their applications. If the behavior of the application largely changes by request patterns, they can obtain multiple execution profiles for all the patterns and merge generated pointcuts. Currently, QoSWeaver does not provide any support for obtaining a representative execution profile.

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

- a *target interval* between adjacent join points selected by pointcuts, and
- the acceptable *maximum occurrences* of join points selected by a single pointcut.

The target interval specifies the time from when an application thread calls a scheduler until when the thread calls it again, on the profiled execution. The criterion for the pointcut generator is that the average time elapsed between adjacent join points selected by pointcuts is close to the target 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.

These two parameters should be determined so that QoSWeaver will generate a scheduling mechanism with acceptable accuracy and overhead for an intended scheduling policy. The accuracy and overhead of application-level scheduling also depends on the characteristics of the web application, the patterns of requests to web applications, and the underlying systems. Therefore, developers should give different sets of parameters to the pointcut generator and obtain multiple sets of pointcuts. Then they should choose the best one by changing the sets of pointcuts and running the web application for evaluation. Our pointcut generator makes this kind of parameter tuning easy.

Note that the target interval is the interval to be achieved in the same execution contexts as when we obtained the execution profile. During the profiled execution, we ran only one application thread on a server. If production run is also single-threaded, an observed interval, at which an application thread calls a scheduler, would be almost the same as the target interval on average. If it is multi-threaded, however, an observed interval for each application thread would be longer than the target. In application-level scheduling, the observed interval largely depends on the patterns of the requests to web applications.

Algorithm for Pointcut Generation. Figure 3 shows our algorithm of pointcut generation. The inputs to this algorithm are an execution profile and the two parameters described above. The execution profile consists of the data of the join points that an application thread reached during the profiled execution. In our implementation, it is a sequence of the invoked method calls, as in Fig. 4. According to the time stamps recorded at the join points, the algorithm divides the sequence into time slots by the target interval t. 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. Since a scheduler is called only at selected join points, this algorithm enables application threads to call a scheduler at as regular intervals as possible. The generated pointcuts are chosen among possible pairs of call and withincode pointcuts. The call pointcut specifies a method by its name and signature and matches points at which the method is called at runtime. The withincode pointcut limits the scope in which the call pointcut selects method calls to a specified method body. Thus each pair of pointcuts selects join points representing calls to the same method within the same method body. In this algorithm, no pointcut includes wildcards.

The algorithm first chooses pairs of pointcuts that select a join point occurring only once in the execution profile. For example, if a method A is called from a method B only once during the entire profiled execution, the caller and the callee are used to make a pair of withincode and call pointcuts. Let PC_1 be the set of chosen 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 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 pointcuts that select join points occurring only twice in the execution profile. Let PC_2 be the set of chosen pointcuts. The algorithm computes a subset of PC_2 that covers as many not-covered time slots as possible. If there are multiple pointcuts that cover the same time slot, the algorithm chooses the pointcut 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

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

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

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
```

Fig. 3. The algorithm for 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.

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.

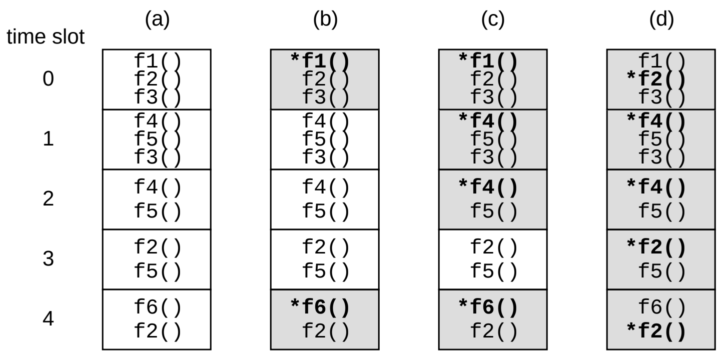

Fig. 4. An example of pointcut generation. A sequence of method calls invoked during profiled execution is divided by a target interval. Marked method calls are selected join points.

For example, suppose that the profiled execution is modeled as a sequence of invoked method calls, f1() to f6(), in Fig. 4 (a). For simplicity, we ignore caller's methods in this example. The profiled execution consists of five time slots. The algorithm first chooses a pointcut that selects f1() in time slot 0 and one that selects f6() in time slot 4. Now, these two pointcuts are in PC_{gen} and the time slot 0 and 4 are covered (Fig. 4 (b)). Then the algorithm chooses PC_2, which contains two pointcuts: a pointcut that selects two f3() and one that selects two f4(). Since the pointcut that selects f4() covers two new time slots (time slot 1 and 2), the algorithm adds the pointcut that selects f4() to PC_{gen} (Fig. 4 (c)). The pointcut that selects f3() covers only one new time slot (time slot 1). At the final iteration, a pointcut that selects three f2() is added to PC_{gen}. At the same time, the pointcuts that select f1() and f6() are removed from PC_{gen} (Fig. 4 (d)). The time slot 0 and 4 are covered by the pointcut that selects f2(). The algorithm results in the pointcuts that select f2() and f4(). In this example, all the time slots are covered by those pointcuts and each time slot includes only one selected method call.

3.4 Concerns for Scheduling Policies

Synchronization. Scheduling code woven into applications by QoSWeaver includes synchronization code for suspending and restarting a thread. We implemented the synchronization by the Object.wait and Object.notify methods in Java.

Adding such synchronization code may cause deadlocks if the original applications also include synchronization code. Suppose that thread A and B running in an application access the same synchronized object in a synchronized block as shown in Fig. 5. If join points in the block are selected by pointcuts, the thread B calls the scheduler object and can be suspended in the block to yield the allocated CPU time. While the thread B is suspended in the block, the thread A will be blocked if it attempts to enter the block because the thread B does not release the lock. If the thread A has a role to wake up the thread B within or after the block, a deadlock occurs. The thread A cannot wake up the thread B forever. However, typical web applications do not often include synchronization code. In particular, the Enterprise JavaBeans (EJB) specification [15] prohibits using thread synchronization. In fact, the web applications that QoSWeaver was applied to in Sect. 4 did not use thread synchronization although they were not EJB applications.

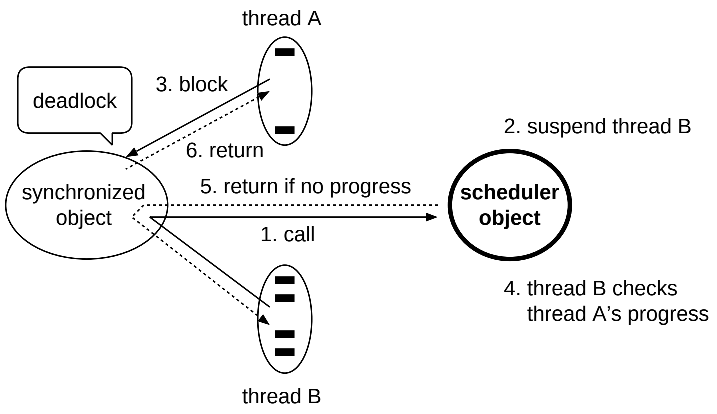

Fig. 5. A scheduling method for breaking deadlocks.

If web applications include synchronization code, developers can write scheduling policies that wake up suspended threads periodically and run application code a little. This is implemented by using the Object.wait method with timeout in scheduler code. As shown in Fig. 5, when the specified timeout is expired after thread B was suspended, thread B executes the scheduler code, which checks the progress of the other running threads, in this case, thread A. If some threads do not make progress for a while, the scheduler decides to restart thread B temporarily because those threads may make no progress due to deadlocks. If all threads make progress when thread B calls the scheduler code at the next join point selected, thread B calls the Object.wait method to suspend again. It is guaranteed that the applications recover from a deadlock if all the suspended threads temporarily run, as far as the original applications do not include deadlocks in their logic. This approach also prevents livelocks, that is, a restarted thread never suspends instantly again for waiting the same lock. When a suspended thread is restarted by timeout, it necessarily proceeds to the next join point selected by pointcuts.

To record the progress of threads for the above method, AOP is also useful. For example, QoSWeaver can weave an aspect for incrementing a progress counter at various join points in applications. Such join points should be selected by pointcuts so that a thread will reach them at as regular intervals as possible and the accurate progress can be monitored. Our pointcut generator can generate appropriate pointcuts for that purpose.

If applications are written without using thread synchronization, the developers can write scheduling policies concisely because they do not need to consider deadlocks. Even when thread synchronization is used, deadlocks are avoidable if the usage is localized. The pointcut generator can generate pointcuts that do not select any join points in synchronized blocks.

Some readers may think that using an independent scheduler thread is straightforward to solve deadlocks because the scheduler 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. If an application thread is suspended within a synchronized method in a system class, the scheduler thread may block at that method and then a deadlock may occur.

I/O. When a thread blocks for I/O, the schedulers of the underlying operating system and middleware automatically allocate CPU time to another thread. An application-level scheduler does not need to reschedule threads whenever a thread issues blocking I/O. This makes it easy to implement scheduling policies. 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 modify scheduling policies to make a thread call a scheduler and temporarily run another thread just before it issues blocking I/O. After the thread completes the I/O, it can immediately call the scheduler to suspend the temporarily running thread.

4 Case Study

To examine that QoSWeaver enables 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 6 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 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.

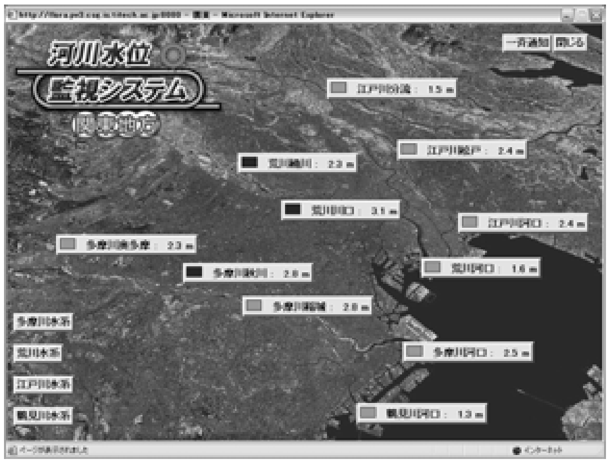

Fig. 6. The up-to-date water levels in Tokyo shown by Kasendas.

To make the results of our experiment reliable, Kasendas was initially developed by an outside corporation with CMMI level 3 [16]. 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: JBoss application server [17], the Tomcat web container [18], the Struts framework [19], and the Seasar2 container [20]. 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 7 shows the architectural overview of Kasendas. Kasendas has three web applications for *periodic data collection*, *chart generation*, and *water-level*

Table 1. The code size of 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

update. One web application 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, the application sends the same number of requests as rivers to the data generator. The collected data are stored in the PostgreSQL database [21] and the latest data is also kept in memory. The other two applications generate web pages for the users. 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 [22]. This is a heavy-weight application because it accesses the database and produces a PNG image of the chart like Fig. 8. The other generates a web page showing up-to-date water levels. It reads data on memory and generates an image like Fig. 6.

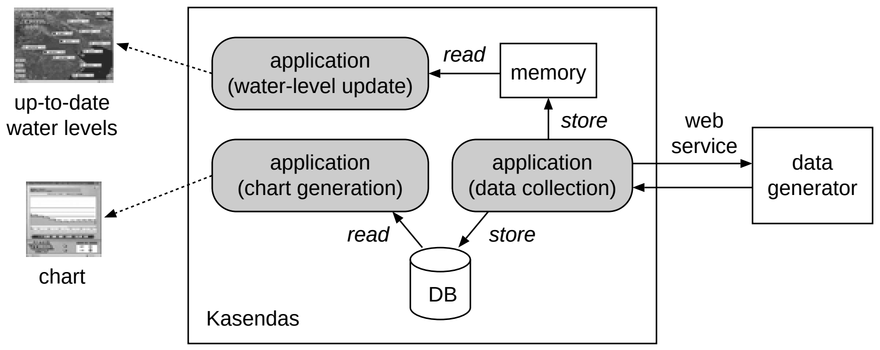

Fig. 7. The architecture of Kasendas.

Kasendas executes these three applications as follows. A timer in Kasendas triggers the execution of the application for periodic data collection. A single thread allocated by the timer executes the application at regular intervals. On the other hand, the other two applications for the chart generation and the water-level update 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.

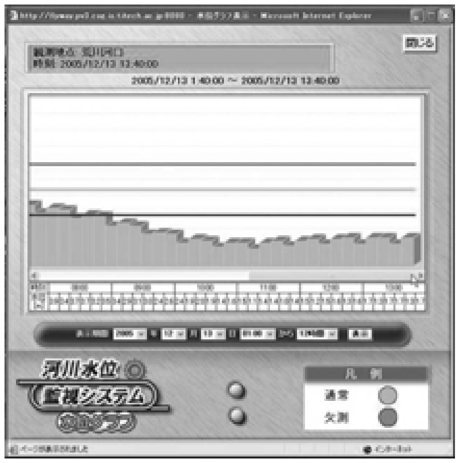
Fig. 8. The generated chart of recent changes of water levels in a river.

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. Since generating that page is a heavy-weight task, it consumes a large amount of CPU time and thus it disturbs another application for periodic data collection. 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 up-to-date water levels in the next time period.

4.2 The Applied Scheduling Policy

To fix this performance problem, we used QoSWeaver. There were two performance requirements:

- preventing a deadline miss in the periodic data collection even under heavy workload, and
- preventing performance degradation of the chart generation.

Note that true real-time scheduling was not required for this case. These requirements are somewhat vague and they are about user experiences. Our goal is to satisfy these requirements *as much as possible* with a minimum software development cost when we already have a program that mostly works well on a normal software stack.

Therefore, the scheduling policy applied to Kasendas was proportional-share scheduling for two groups of threads for the chart generation and a thread for periodic data collection. The former threads have low importance and the latter has high importance in this policy. While the high-importance thread does not run, the scheduling policy runs all the low-importance threads to generate a chart. When the high-importance thread starts running for periodic data collection, the scheduling policy 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 for the water-level update. 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 periodic data collection. At this time, the scheduler limits the number of running low-importance threads by setting the yield flags. 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. 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 suspended low-importance threads.

To write an aspect that implements this scheduling policy for QoSWeaver, we used our own AOP system named GluonJ [23]. 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 Fig. 9 and Fig. 10.

Figure 9 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 and highImportance pointcuts consist of the execution pointcut, which specifies a method by its name and signature and matches points at which the method is executed at runtime. The lowImportance pointcut selects the execution of the method starting the chart generation. The highImportance pointcut selects the execution of the method starting periodic data collection. 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 [24], respectively. The call pointcut specifies a method by its name and signature and matches points at which the method is called at runtime, and the withincode pointcut limits a caller's method within which

[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>
```

Fig. 9. The pointcut declaration in our aspect.

the call pointcut matches method calls. The controlPoint pointcut selects various method calls during the chart generation as far as these methods are called from the specified caller methods. A scheduling mechanism specific to Kasendas is constructed from these pointcuts. The pointcuts specify scheduling points in the source code of Kasendas and its application threads call a scheduler implemented as advice when they reach the scheduling points at runtime.

Figure 10 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. The around advice is executed in place of its join points selected by a pointcut while the before advice is executed before its join points. In the around tag, proceed() calls the original method selected by a pointcut. 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 Fig. 11. Advice and its support classes are the implementation of a scheduling policy, which is named a scheduler.

```
<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>
```

Fig. 10. The advice declaration in our aspect.

The first around advice is executed when a low-importance thread generates a chart. It calls the scheduler to register the thread to be controlled before the selected method execution and to unregister it after that. The second around advice is executed when a high-importance thread performs the periodic data collection. It calls the scheduler to control the number of running low-importance threads before and after the selected method execution. This policy temporarily reduces the number to one and restores it to 40, which was equal to the maximum number of concurrent requests to a web page showing a chart in our experiments. While the periodic data collection was not performed, we did not need to restrict the number of running threads. The last before advice is executed before a low-importance thread performs the selected method calls during the chart generation. It calls the scheduler to yield the execution of the thread itself that executed the advice.

```
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());
   }
}
```

Fig. 11. A support class for our aspect.

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:

- 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 suspends the current thread by invoking the Object.wait method.
- The remove method removes the current thread from the run queue. If the number of threads in the run queue is under the maximum, the method

resets the yield flags of some threads in the wait queue and wakes up the threads by invoking the Object.notify method.
- The schedule method moves some 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 some threads in the wait queue to the run queue, resets their yield flags, and wakes up those threads.
- The yield method suspends the current thread, if its yield flag is set, by invoking the Object.wait method.

We did not specify a timeout for the Object.wait method because the original 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 was 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 *except* 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 to other ones 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 was not equivalent to our work. It did not include the analysis of performance bottlenecks. The work was 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 prevented 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.

4.4 Additional Scheduling Policies

Deadlock-Aware Scheduling Policy. To examine that QoSWeaver can break deadlocks introduced by woven scheduling code, we modified Kasendas so that it would cause a deadlock when it ran with our scheduling policy described in Sect. 4.2. We added a Logging class including two synchronized methods to Kasendas. These two methods, writeCollection and writeGeneration, are called by a high-importance thread for the periodic data collection and low-importance threads for the chart generation, respectively. When a low-importance thread is suspended within the writeGeneration method and waits for being woken up by the high-importance thread, a deadlock occurs if the high-importance thread calls the writeCollection method. Since the high-importance thread blocks at the writeCollection method, it cannot wake up the suspended low-importance thread.

An aspect as shown in Fig. 12 records the progress of the high-importance thread. The advice calls the PSScheduler.setProgress method 100 times during the execution of the high-importance thread. The pointcut was generated by our pointcut generator on the basis of the execution profile for the periodic data collection. The setProgress method increments the value of a progress variable. In addition, we modified the controlPoint pointcut in our original aspect so that a join point within the writeGeneration method was selected to yield thread's execution.

```
<pointcut-decl>
   <name> progress </name>
   <pointcut>
      (withincode(int CollectorImpl.getWaterLevel(int)) ANDAND
       call(int Integer.parseInt(java.lang.String)))
   </pointcut>
</pointcut-decl>

<behavior>
   <pointcut> progress() </pointcut>
   <before> PSScheduler.setProgress(); </before>
</behavior>
```

Fig. 12. The aspect added for measuring a progress.

To enable breaking deadlocks, we also modified the yield method in the PSScheduler class. Within the yield method, a thread calls the Object.wait method with the timeout of 200 ms if its yield flag is set. When the execution of the Object.wait method finishes, the thread checks its yield flag again. If the flag is reset, the thread finishes the execution of the yield method and executes application code because this means that the thread is woken up by the Object.notify method. Otherwise, the thread compares the current value of the progress variable with the saved previous one to check the progress of the high-importance thread. If the value is changed, the high-importance thread is running and no deadlock occurs. Therefore, the thread calls the Object.wait method again. Otherwise, it temporarily executes application code to break potential deadlocks. During this temporary execution, the yield flag is set.

After that, the low-importance thread checks the progress of the high-importance thread whenever the yield method is called. If the yield flag is set and if the high-importance thread makes no progress, the thread skips the rest of the yield method and continues the temporal execution. If the high-importance thread makes progress, the low-importance thread finishes the temporal execution and executes the yield method normally.

Adaptive Scheduling Policy. So far, we considered only the low-importance threads for the chart generation and the high-importance thread for the periodic data collection. Kasendas has another web application for the water-level update. This application is of intermediate importance and should be run by a middle-priority thread. It is more important than that for the chart generation because, in the disaster case, the up-to-date water levels are more critical information than their changes in the past. It is not as important as the application for the periodic data collection. When we consider such middle-importance threads as well, they conflict with both low- and high-importance threads. The throughput of the middle-importance task may decrease due to the low-importance task when many low-importance threads are running because the chart generation performed by the low-importance threads is a heavy-weight task. While the periodic data collection is performed, the time needed for periodic data collection may be increased by the middle-importance task. Although the web application for the water-level update is light-weight, too many requests to the web application affect the other applications.

To minimize such conflicts, we extended our scheduling policy described in Sect. 4.2. The extended policy guarantees the target throughput for the middle-importance task on average when a high-importance thread does not run. It adaptively adjusts the number of low-importance threads so that the middle-importance task achieves the target throughput. While the high-importance thread is running, the policy limits the maximum throughput of the middle-importance task. It directly adjusts the execution of the middle-importance threads because the number of the low-importance threads is limited to one during data collection by our original policy and it is difficult to be adjusted furthermore. Figure 13 is an aspect added for the middle-importance task. The midImportance pointcut selects the execution of the method in the AbstractMapAction class, which is invoked from the ActionServlet class used by the Struts framework [19]. The around advice is executed when a middle-importance thread performs the water-level update.

```
<pointcut-decl>
   <name> midImportance </name>
   <pointcut>
      execution(ActionForward AbstractMapAction.execute(..))
   </pointcut>
</pointcut-decl>

<behavior>
   <pointcut> midImportance() </pointcut>
   <around>
      PSScheduler.startMidImportance();
      $_ = proceed($$);
      PSScheduler.endMidImportance();
   </around>
</behavior>
```

Fig. 13. The aspect added for the middle-importance task.

First, we consider only middle- and low-importance tasks when the periodic data collection is not performed. To calculate the throughput of the middle-importance task, our scheduler counts the number of pages generated during a certain period. The number is incremented by the PSScheduler.endMidImportance method, which is called after the execution of the water-level update. To adjust the maximum number of low-importance threads, we added the adjust method to the ThreadController class, which is described in Sect. 4.2. The method is called in the startMidImportance, endMidImportance, and yield methods of the PSScheduler class. Thus the adjust method is frequently called if the middle- or low-importance threads are running.

The adjust method adjusts the maximum number of low-importance threads if the specified time elapses since the last adjustment. It does nothing until the specified time elapses. In the current implementation, the time is one second. The `adjust` method first calculates the latest throughput of the middle-importance task from the number of generated pages and the time elapsed since the last adjustment. Then it calculates the ratio of the observed throughput to the specified target throughput. If the ratio is less than one, the method decreases the maximum number of low-importance threads by the ratio to decrease the load by the low-importance threads. If the ratio is more than one, the method increases the number by the ratio to suppress the execution of the middle-importance threads. For example, suppose that the maximum number of low-importance threads is 5. When the observed throughput is 180 pages/sec and the target is 150 pages/sec, the ratio is 1.2 and then the maximum number is increased to 6.

Next, we consider three kinds of tasks when the periodic data collection is performed. To limit the number of requests processed in parallel, our scheduler maintains the number of running middle-importance threads. The startMidImportance method increments the number and the endMidImportance method decrements it. While the high-importance thread is running, the startMidImportance method checks the number of running middle-importance threads. If the number is more than the specified value, the current thread is suspended. When a middle-importance thread finishes the water-level update, the endMidImportance method wakes up one of the suspended threads. The woken-up thread waits for the specified time because of adjusting the rate of request processing. The maximum throughput is determined by the maximum number of middle-importance threads and the waiting time. When the high-importance thread finishes the periodic data collection, all the suspended threads are woken up. At the same time, the maximum number of low-importance threads is restored to the value just before starting the periodic data collection. This behavior is changed from the original policy, which restores the number to the fixed value, 40. This change enables quickly stabilizing the maximum number of low-importance threads after the periodic data collection.

5 Experiments

Our application-level scheduler successfully improved the execution performance of Kasendas. Interestingly, we could not achieve this improvement by using existing schedulers of the underlying software layers such as the JVM and the Linux operating system. This section illustrates this fact through the results of our experiments.

5.1 Five Versions of Kasendas

We ran not only our Kasendas tuned with QoSWeaver but also the original Kasendas without any tuning and other versions of Kasendas tuned with admission control, Java priority scheduling, and Linux priority scheduling. 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 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 restricts the maximum number of running low-importance threads for generating a chart. It limits the maximum number to one while a high-importance thread is 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.

The other two versions of Kasendas were tuned by scheduling mechanisms in the JVM and the operating system. To support our claim in Sect. 2, it is important to show that only using such existing scheduling mechanisms is insufficient. Java provides the Thread.setPriority method for priority scheduling. Using this method, Kasendas sets the priority of the high-importance thread to high (`MAX_PRIORITY`) while it sets the priority of the low-importance threads to low (`MIN_PRIORITY`). After the low-importance threads finish to process requests, Kasendas resets their priorities to normal (`NORM_PRIORITY`) because those threads were reused via a thread pool.

Kasendas tuned with Linux priority scheduling issues the setpriority system call using a native method in Java when threads start processing requests. Like the setPriority method in Java, Kasendas sets the priorities of the high- and low-importance threads to high (-20) and low (19), respectively. However, Kasendas has to be run with root privilege to raise the priority of the native thread. It has to change the priority of the high-importance thread from normal (0) to high. In addition, it has to change the priority of the low-importance thread from low to normal when it returns the low-importance thread to the thread pool.

We did not change the operating system and the JVM used in Kasendas to real-time ones because that is not realistic at software development in practice as described in Sect. 2. If we largely change such underlying software, we may rewrite our applications. In addition, all of the underlying software need to be changed to real-time systems, but there existed no real-time JBoss server, which Kasendas required.

For our experiments, the interval at which Kasendas collects water levels was 15 seconds. To generate workloads, we used Apache JMeter [25]. JMeter concurrently sent requests to the web page showing a chart of recent changes for the last 12 hours, except one experiment in Sect. 5.2. The number of concurrent requests was 40, except one experiment in Sect. 5.3. The number, 40, was the maximum number of concurrent requests through our experiments because the chart generation was a heavy-weight task and processing 40 requests in parallel caused overload in our server. Although more requests may be sent to the server in practice, we assume that more than 40 requests are discarded by admission control. We did not send requests to the web page showing the up-to-date water levels, except one experiment in Sect. 5.5.

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.

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 was 15 seconds in our configuration. Our aim is to prevent such deadline misses for the periodic data collection.

Figure 14 shows the changes of the time needed for collecting water levels every 15 seconds and Fig. 15 shows the average collection time. When we used the original Kasendas, we could measure the collection time only six times during 180 seconds. This is because each data collection took long time. The average collection time was 24.8 seconds and every collection time was more than 15 seconds, which was a deadline, except at 30 seconds. 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. In addition, the collection time was the stablest among the five versions of Kasendas. The variance of our scheduling was the smallest. This is very important for applications with deadlines because it becomes easier to guarantee that applications do not miss their deadlines. For the admission control, the average collection time was 10.9 seconds, but the collection time sometimes exceeded 15 seconds, for example, at 30 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.

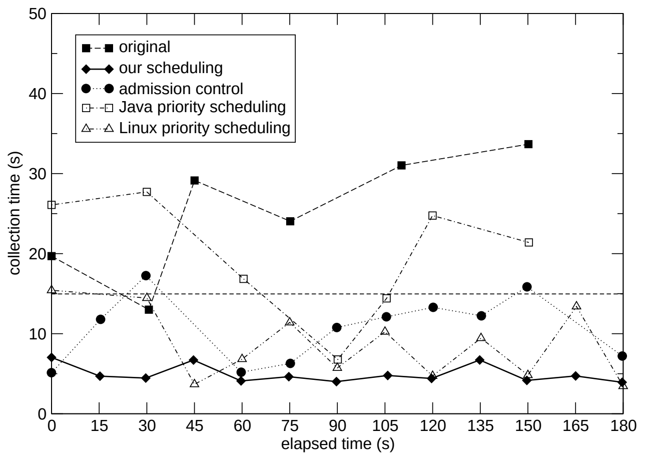

Fig. 14. The changes of the time needed for a high-importance thread to collect water levels.

Kasendas tuned by Java priority scheduling achieved a little shorter collection time on average than the original one, but the average was still longer than 15 seconds. The average collection time of Kasendas tuned by Linux priority scheduling was 9.0 seconds. However, Linux priority scheduling was not as good as our scheduling. The collection time was not stable and longer than 15 seconds at the first data collection. The cause of this instability is that the execution of the chart generation has several phases. Since JMeter simultaneously sent 40 requests at time 0, many threads tended to execute the same phase synchronously. Therefore, the characteristics in each phase affected the performance of the periodic data collection.

The deadline miss ratio was 0.89 for the original Kasendas, but it was reduced to zero by our scheduling. The other approaches could not achieve this: 0.31 for the admission control, 0.89 for Java priority scheduling, and 0.18 for Linux priority scheduling.

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

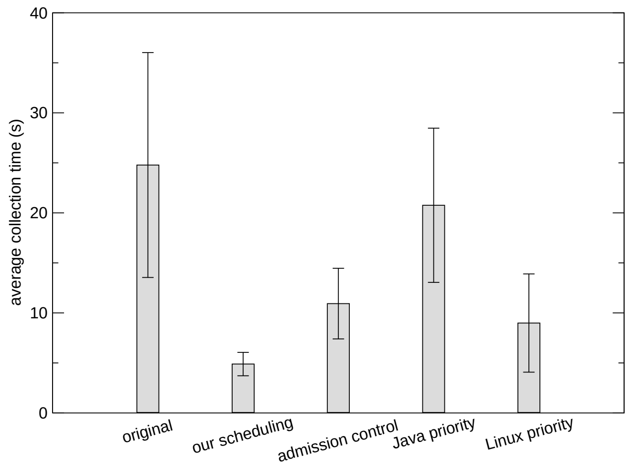

Fig. 15. The average time needed for a high-importance thread to collect water levels.

how quickly low-importance threads are suspended. The quickness of the thread suspension can affect the collection time of water levels.

Figure 16 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 2.2 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 0 second to time 30 seconds. Even when all but one low-importance thread were suspended, the average suspension time was 10.2 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.

Impact of Changing Workloads. In the above experiments, JMeter sent requests to a web page showing a chart of recent changes for the last 12 hours. Generating the 12-hours chart was the same workload as what we used to obtain execution profile for our pointcut generator. We changed workloads so that JMeter sent requests to web pages showing charts for the last 24, 12, 6, and 3 hours. As the period of a generated chart becomes smaller, the application generating a chart obtains the 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 different workloads.

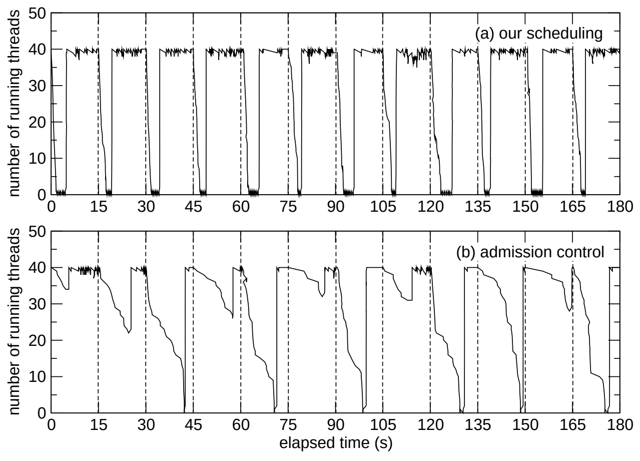

Fig. 16. The changes of the number of running low-importance threads.

Figure 17 shows the average time needed for a high-importance thread to collect water levels when we changed the workloads. Our scheduling, the admission control, and Linux priority scheduling kept the average collection time to almost the same under any workloads. On the other hand, when we used the original Kasendas, the average collection time increased from 23.5 to 49.0 seconds at maximum and more data were lost. Kasendas tuned with Java priority scheduling was worse than the original one for the 24-hours and 6-hours charts. Like Fig. 15, the variance of our scheduling was the smallest. That of the original Kasendas becomes larger for the shorter period. On the contrary, that of our scheduling becomes smaller as the period of the generated chart is decreasing. This indicates that threads for generating a chart of a shorter period are easier to control under application-level scheduling because the chart generation becomes relatively lighter-weight task. Table 2 shows the deadline miss ratios. In our scheduling, the deadline miss ratio was zero for every case. From these results, it is shown that our scheduling can control Kasendas stably even when the workload is a little different from that used in the profiled execution.

Table 2. The deadline miss ratios.

period	24-hours	12-hours	6-hours	3-hours
original	0.92	0.89	0.97	0.95
our scheduling	0.00	0.00	0.00	0.00
admission control	0.39	0.31	0.28	0.11
Java priority	0.95	0.89	1.00	0.98
Linux priority	0.15	0.18	0.18	0.00

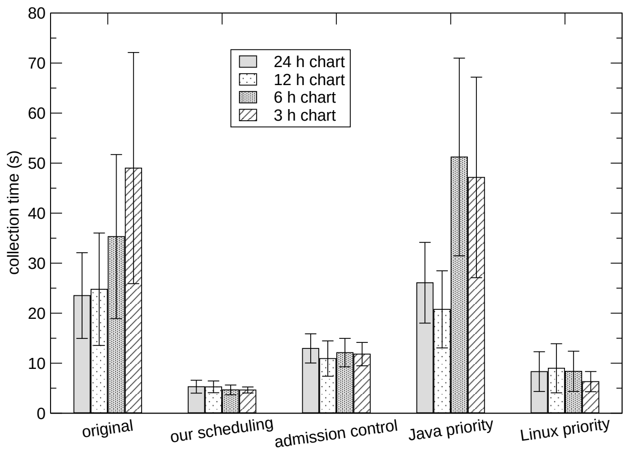

Fig. 17. The time needed for collecting water levels when the workload is changed.

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. Since our scheduling policy temporarily suspends low-importance threads to give sufficient CPU time to a high-importance thread, the throughput of the chart generation would be degraded. Figure 18 (a) shows the throughput of the chart generation. Compared with the original Kasendas, the performance degradation under our scheduling was 15.7 % and larger than that under the other approaches. This is because our scheduling gave more sufficient CPU time to the high-importance thread than the other approaches. 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. Figure 18 (b) shows the average response time. Compared with the original Kasendas, the average response time under our scheduling increased by 18 %. The 95 % confidence intervals are (17.2, 18.2) and (19.7, 21.2) for the original Kasendas and our scheduling, respectively. Since these two do not overlap, the increase is statistically significant. In addition, the variance of our scheduling was larger than that in the original Kasendas because the low-importance threads were given a low priority and all but one thread were suspended during periodic data collection.

Scheduling Overhead. To examine the scheduling overhead, we measured the throughput of the chart generation without performing the periodic collection of water levels. In our scheduling policy, low-importance threads execute the chart generation and periodically call a scheduler's method. Thus the chart generation can cause scheduling overhead. We stopped the periodic data collection to mea-

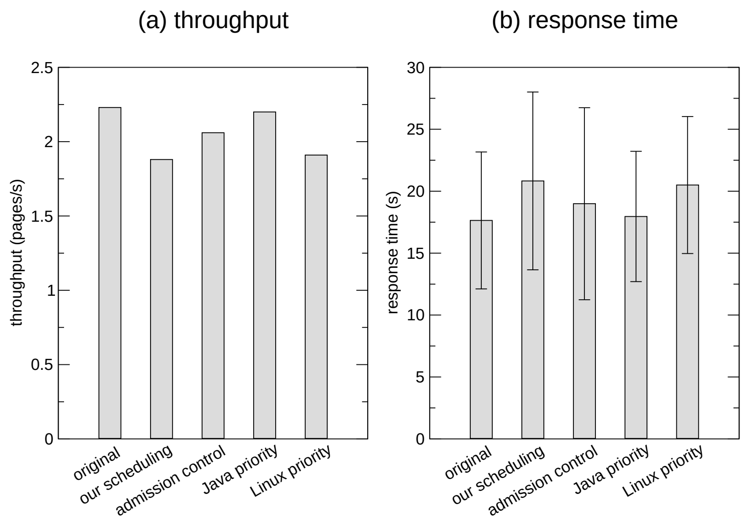

Fig. 18. The throughput of the chart generation and the response time of a web page showing a chart.

sure pure overhead because the periodic data collection makes low-importance threads be suspended.

Compared with the original Kasendas, the throughput was not degraded in our scheduling and no scheduling overhead was measured. This is because calling a scheduler's method was very light-weight and a low-importance thread called the method at only 17 join points during handling one request in our experiment. For the admission control, there was also no overhead. On the other hand, the throughputs were degraded by 3 % and 11 % with the priority scheduling by the JVM and by Linux, respectively. This is because priority scheduling gave a low priority to low-importance threads even while the periodic data collection was not performed.

5.3 Usefulness of the Pointcut Generator

We examined the impact of parameters given to our pointcut generator. The pointcut generator takes two parameters: a target interval between adjacent join points selected and the maximum occurrences of join points selected by a single pointcut. In Sect. 5.2, we used the controlPoint pointcut generated with the target interval of 10 ms and the maximum occurrence of 1. For the experiments in this section, we changed the target interval to either 10 or 100 ms and the maximum occurrence to 1, 50, 100, or 200.

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 3 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 target 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.

Table 3. The numbers of generated pairs of pointcuts and join points selected by them for different sets of parameters.

target interval	maximum occurrence	generated pointcuts	selected join points
10 ms	1	15	15
	50	16	32
	100	17	231
	200	15	309
100 ms	1	8	8
	50	9	13
	100	8	83
	200	8	83
random		15	2034
all		803	248661

For comparison, we chose 15 pointcuts from 803 candidates at random without the pointcut generator. Such random pointcuts become the baseline for examining the goodness of the pointcuts generated by the pointcut generator. Choosing pointcuts at random only reduces the number of pointcuts whereas the pointcut generator minimizes the number of join points selected in a certain period as well. When we used random pointcuts, the number of selected join points was much larger than when we used the pointcut generator. This is because some pointcuts selected too many join points in Kasendas. We should also compare the pointcuts generated by the pointcut generator with ideal ones, but obtaining ideal ones is very difficult for non-toy applications.

The time needed for generating these pointcuts was 20 seconds even when we specified 10 ms for the target interval and 200 for the maximum occurrence. The time depends mainly on the number of join points included in execution profile. To examine the scalability of pointcut generation, we also measured the time needed to generate pointcuts for the web application generating a chart for the last 24 hours. The number of join points was 959148 in its execution profile and the time needed for pointcut generation was 103 seconds. Compared with the chart generation for the last 12 hours, the number of join points becomes 3.9 times larger while the time becomes 5.2 times longer. The increment of the time is not proportional to that of the number of join points, but the time is not too long.

Influences to Scheduling Intervals. We examined how the parameters given to the pointcut generator affected the interval at which the scheduler was called at runtime. Since the scheduler is called at join points selected by generated pointcuts, the interval is the time between adjacent join points selected. For comparison, we also examined the interval between all adjacent join points and that between adjacent join points selected by random pointcuts. First, we measured these intervals in single-thread execution, which was performed to obtain execution profile for pointcut generation. Only one low-importance thread ran at the same time. Figure 19 shows the averages of the observed intervals for different sets of parameters given to the pointcut generator. When we did not use the pointcut generator, the observed intervals were too small for application-level scheduling. The observed interval was 0.01 ms when all join points were selected and that was 0.9 ms when join points were selected by random pointcuts. When we gave appropriate parameters, the pointcut generator could generate pointcuts so that the observed interval approached the target. For example, if the target interval was 10 ms and the maximum occurrence was 100, the average interval was the nearest to the target one. The observed interval tends to be smaller as the maximum occurrence became larger.

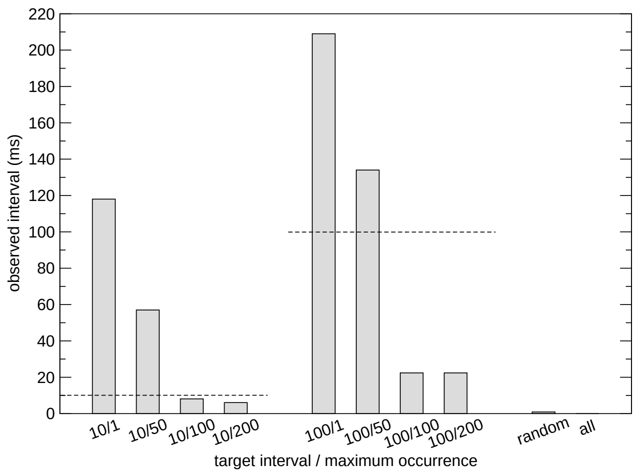

Fig. 19. The intervals at which the scheduler is called in single-thread execution.

The variance of observed intervals 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 20 plots the time when a program flow reached join points selected by pointcuts. 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 the image for the 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 that included too many join points: time 1.0 to 1.6 seconds, for example. In this part, JFreeChart repeated the same processing to generate a chart too many times. Even the most infrequent method call was done every 1.8 ms. This is too frequent, compared with the time quantum of 5 ms assigned to processes with the lowest priority in Linux. 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.

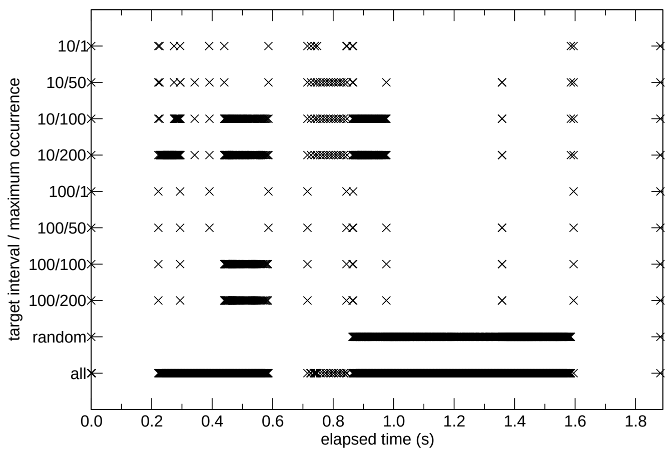

Fig. 20. The time when a program flow reaches join points selected by generated pointcuts.

Next, we measured the intervals in multi-thread execution. JMeter sent 10, 20, and 40 requests to a web page showing a chart in parallel. The aim of this experiment is to examine how a server load affects the observed intervals. Figure 21 shows the average of the observed intervals. At worst, each low-importance thread could call the scheduler every 1.9 seconds on average. For the parameters used in our experiments in Sect. 5.2, each low-importance thread called the scheduler every 1.1 seconds on average. This enabled stable control as shown in Sect. 5.2. This figure also shows that the observed interval was proportional to the number of concurrent requests. These results show that we could predict the observed interval in multi-thread execution from that in single-thread execution.

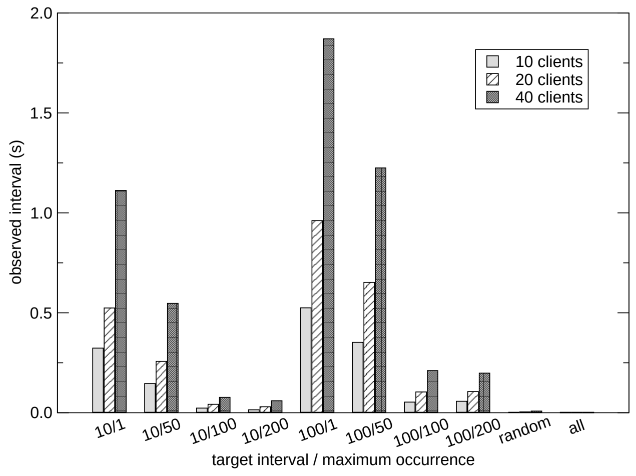

Fig. 21. The intervals at which the scheduler is called in multi-thread execution.

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. In this experiment, JMeter sent 40 requests in parallel. The result is shown in Fig. 22. The time needed for thread suspension was different for each set of parameters. The time needed for the data collection reflected these differences, but it was not different as largely as the suspension time. The collection time was between 4.5 and 5.3 seconds and sufficient for avoiding deadline misses. On the other hand, when we used a pointcut that selected all method calls, the suspension time decreased too much and achieved short collection time. By contrast, when we used random pointcuts, the suspension time increased by a factor of two and the collection time became long. Also, its variance of the suspension time became very large because the scheduler was called at random intervals. The deadline miss ratio was zero for every case.

Next, we measured the throughput and the response time of the chart generation. The performance was almost never affected by the parameters. However, when too many join points were selected due to using no pointcut generator, the performance was degraded largely. The throughput was degraded by 57 % and the response time was 2.2 times longer. This means that decreasing the number of selected join points is important in terms of performance.

Finally, we examined scheduling overheads by measuring the throughput of the chart generation without the periodic data collection. When we did not use the pointcut generator, the scheduling overhead was 63 %. The pointcut generator reduced the overhead to less than 5 % for every set of parameters. For our experiments in Sect. 5.2, we experimentally selected the target interval of 10 ms and the maximum occurrence of 1 so that the scheduling overhead was minimized.

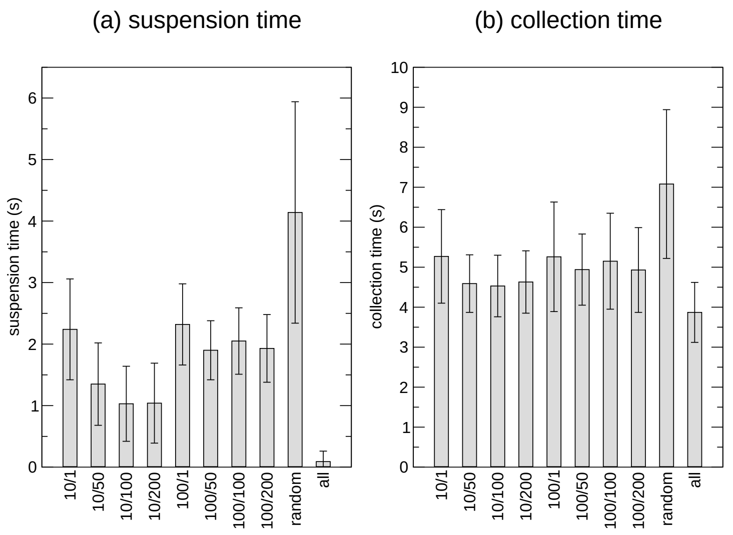

Fig. 22. The time for the thread suspension and the data collection for different sets of parameters.

5.4 Effectiveness of Deadlock-Aware Scheduling

In this subsection, we used the Kasendas to which the Logging class described in Sect. 4.4 was added for introducing synchronization. When we wove the original aspect into this Kasendas and sent requests, a deadlock always occurred soon as we intended. A thread for collecting water levels blocked at the Logging.writeCollection method and it was not continued. On the other hand, when we wove the deadlock-aware version of aspect into the Kasendas, deadlocks did not occur.

Next, we measured the time needed for collecting water levels when we wove the deadlock-aware version of aspect. We also measured the times for the other four versions of Kasendas, which used their own scheduling policies. For them, we introduced synchronization by adding the Logging class. They did not cause deadlocks although we did not change their scheduling policies. The results were almost the same as Fig. 15 and the average collection time by our scheduling was the shortest (6.1 seconds). However, this is 0.8 second longer than that by the results in Fig. 15. The increment was caused by the overhead due to checking progress and the delay for detecting deadlocks.

Figure 23 shows the changes of the number of running low-importance threads. The result was different a little from that in Fig. 16 (a). The scheduling policy was to reduce the number of running low-importance threads to one, but this goal was often not achieved due to synchronization among low-importance threads. If low-importance threads are blocked at the Logging.writeGeneration method after another low-importance thread is suspended within the same method, the blocked threads cannot yield their execution.

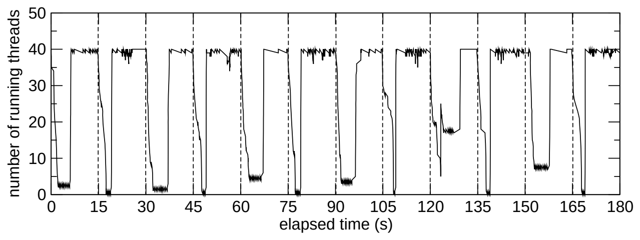

Fig. 23. The changes of the number of running low-importance threads for Kasendas with synchronization.

The performance of a chart generation under the new scheduling was similar to that in Fig. 18, but the performance degradation was larger. Compared with the original Kasendas, the throughput under the new scheduling was degraded by 20 % and the response time was 25 % longer. This is caused by the increment of the collection time. The collection time increased by 0.8 second while the response time was increased by 1.0 second.

5.5 Effectiveness of Adaptive Scheduling

First, we examined the effectiveness of our adaptive scheduling policy described in Sect. 4.4 when the periodic data collection was not performed. JMeter sent requests to both the web page showing a chart of recent changes for the last 12 hours and the web page showing the up-to-date water levels. The number of concurrent requests was 40 for the former page and 100 for the latter page.

Figure 24 shows the changes of the throughput of the middle-importance task for the water-level update when we did not apply any scheduling policy. The dotted line is the observed throughput and the solid line is the average per five seconds. Even the average throughput was very unstable because the execution of the middle-importance threads was largely affected by that of the low-importance threads. The cause of the periodic changes is that the execution of the chart generation had several phases as we explained in the experiment for Linux priority scheduling in Sect. 5.2.

Figure 25 shows the changes of the throughput when we applied our adaptive scheduling policy. We set the target throughput of the middle-importance task to 150 pages/sec. The average throughput per five seconds was stabilized and achieved 152 pages/sec, which was very near to the target throughput. This figure also shows the changes of the maximum number of low-importance threads. Our scheduler frequently adjusted the maximum number of low-importance threads between one and six. It was almost exactly called every second according to our policy. In addition, it decreased the number just after the server started processing requests to the web page for the water-level update and increased the number just after JMeter stopped sending requests.

The response time of the water-level update was also improved by a factor of two. The response time under no scheduling policy was 1.2 seconds while that

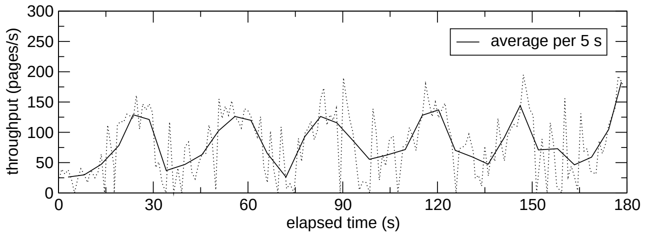

Fig. 24. The changes of the throughput of the middle-importance task under no scheduling policy.

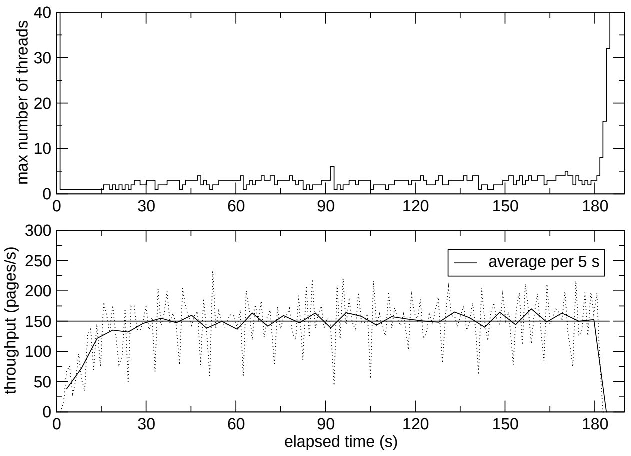

Fig. 25. The changes of the maximum number of low-importance threads and the throughput of the middle-importance task under our adaptive scheduling policy.

under our scheduling policy was 0.62 second. The variance was also smaller under our scheduling policy. On the other hand, the throughput of the chart generation was degraded by adjusting the maximum number of low-importance threads. The throughput under no scheduling policy was 1.84 pages/sec, but that under our scheduling policy was 1.27 pages/sec. This means that the low-importance threads were given a lower priority than the middle-importance threads.

Figure 26 shows the maximum number of low-importance threads and the observed throughputs for various target throughputs. We changed the target throughput from 25 to 250 pages/sec. The observed throughput was near to the target when the target was between 125 and 200 pages/sec. For these target throughputs, the maximum number of low-importance threads was less than ten and the variance was small. When the target throughput was more than 225 pages/sec, the observed throughput was lower than the target due to the upper limits of the system. The maximum number of low-importance threads was almost one because it was the best strategy to minimize the impact by the low-importance threads in our policy. When the target throughput was less than 100 pages/sec, the observed throughput was higher than the target one. The middle-importance threads could run too much even if the large number of low-importance threads simultaneously runs.

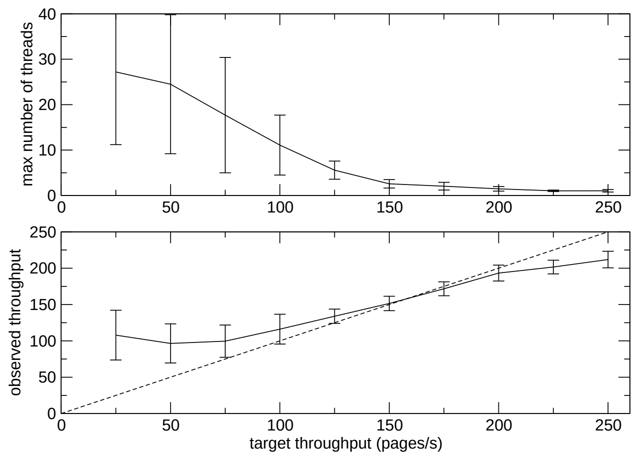

Fig. 26. The averages of the maximum numbers of low-importance threads and the observed throughputs of the middle-importance task for various target throughputs.

Next, we examined the effectiveness of our adaptive scheduling policy when the periodic data collection was performed. We set the target throughput to 150 pages/sec on average when the high-importance thread for the data collection does not run. While the high-importance thread is running, we set its maximum throughput to 150 pages/sec. To limit the maximum throughput, the maximum number of middle-importance threads was limited to 15 and each middle-importance thread waited for 100 ms before starting its execution. This means that the water-level update is performed 10 times at maximum every second for each middle-importance thread.

Figure 27 shows the changes of the maximum number of low-importance threads and the throughput of the middle-importance task. The data collection was performed during the periods marked as “C”. When the high-importance thread did not run, the average throughput was 143 pages/sec, which is near to the target throughput. While the high-importance thread was running, the average throughput was 122 pages/sec, which is less than the specified maximum throughput. In detail, the throughput was more than 150 pages/sec at the beginning of each data collection. Since the number of running middle-importance threads was decreasing to the specified one in several seconds, many middle-importance threads were running just after the data collection started. The average collection time was 6.8 seconds, which is 1.5 seconds longer than the result in Fig. 15 due to running middle-importance threads during the periodic data collection.

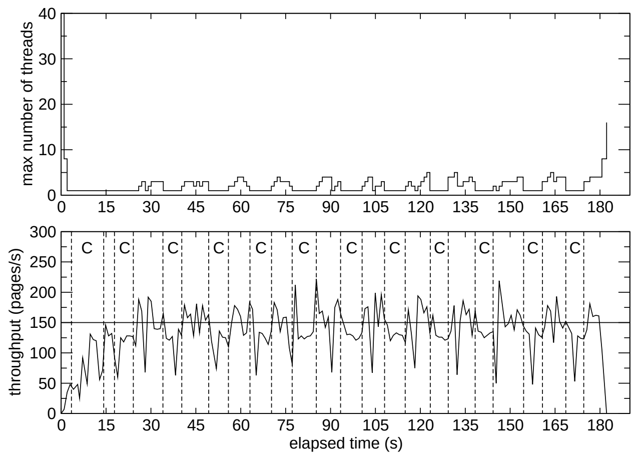

Fig. 27. The changes of the maximum number of low-importance threads and the throughput of the middle-importance task with the periodic data collection.

6 Applicability of QoSWeaver

QoSWeaver is not appropriate if the application needs accurate scheduling. The application-level scheduler by QoSWeaver slowly responds to workload changes. It may take several seconds because application threads are under the control of the underlying operating-system scheduler and the threads only voluntarily yields the allocated CPU time. For example, the result of our experiment in Fig. 16 shows that the number of the running low-importance threads was decreased from 40 to 25 in one second although it must be decreased to one according to the scheduling policy. The scheduling accuracy could be improved if the application program calls a scheduler more frequently. However, the scheduling overheads would be bigger. In the worst case of our experiment, in which a scheduler was called at every method call, the throughput was less than the half of the original. Thus, QoSWeaver could not be used for implementing real-time scheduling. Likewise, within a short period such as one second, it cannot allocate the exact CPU time computed from the priority of the thread. It can only allocate so that the average of the allocated CPU time during several seconds reflects the priority.

QoSWeaver does not work well if the application is I/O intensive and the threads frequently suspend for long time for waiting until an I/O request is completed. Since the thread must be running to call a scheduler, every I/O request should be short and infrequent. Otherwise, the scheduler would not run frequently enough to implement a specified scheduling policy. Furthermore, QoSWeaver does not work well if a small code block is repeatedly executed for long time. If a pointcut does not select a join point in that code block, a scheduler will not be called for long time. On the other hand, if it selects, a scheduler will be called too frequently; the scheduling overheads will be non-negligible.

QoSWeaver assumes that the behavior of the application does not largely change for every execution. It must be similar to the behavior of the profiled execution. If the behavior of the application is categorized into several patterns, we can obtain execution profiles for each pattern, generate pointcuts for each, and merge them all. However, the accuracy of the scheduling will be degraded more as the variation of the application behavior is larger. This fact has been discussed in Sect. 3.3.

7 Concluding Remarks

In this paper, we presented QoSWeaver, which is a tool suite for developing application-level scheduling using aspects. The idea of scheduling at the application level is not new; it is a useful technique for adjusting execution performance with a minimum development cost. We showed that AOP makes this technique more realistic by separating scheduling code from applications. Furthermore, the pointcut generator provided by QoSWeaver generates appropriate pointcuts and helps developers create an application-specific scheduling mechanism, which calls a scheduler from applications periodically.

As a case study, we used a river monitoring system named Kasendas, which is a web application system initially developed by the outside corporation. Using QoSWeaver, we could successfully implement three practical scheduling policies for Kasendas. According to our experiences in the development of Kasendas, QoSWeaver made it easy to develop the scheduling policies in (1) that the scheduling policies could be developed independently of Kasendas and (2) that appropriate pointcuts were automatically generated without examining a large amount of source code of Kasendas. Through this case study, we also experimentally showed the effectiveness of our scheduling policies and the usefulness of the pointcut generator under several workloads.

One of our future work is to apply QoSWeaver to other types of applications. As discussed in Sect. 6, the application classes to which QoSWeaver is applicable are limited from the nature of application-level scheduling and profile-based pointcut generation. To analyze the applicability quantitatively, we need to examine whether QoSWeaver is applicable or not to other real applications. Another direction is to develop other scheduling policies using QoSWeaver. In this paper, we have developed three scheduling policies: proportional-share, deadlock-aware, and adaptive scheduling. To develop scheduling policies that dynamically change the frequency of calling a scheduler, depending on workload changes, it would be necessary for QoSWeaver to support dynamic weaving, for example. We would like to examine that QoSWeaver is useful in practice to achieve other classes of scheduling.

References

1. Kourai, K., Hibino, H., Chiba, S.: Aspect-oriented application-level scheduling for J2EE servers. In: Proceedings of the 6th ACM International Conference on Aspect-Oriented Software Development. (2007) 1–13
2. Hibino, H., Kourai, K., Chiba, S.: Difference of degradation schemes among operating systems. In: Proceedings of DSN2005 Workshop on Dependable Software – Tools and Methods. (2005) 172–179
3. Bollella, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D., Turnbull, M.: The real-time specification for Java. Addison-Wesley (2000)
4. Tesanovic, A., Amirijoo, M., Björk, M., Hansson, J.: Empowering configurable QoS management in real-time systems. In: Proceedings of the 4th International Conference on Aspect-oriented Software Development. (2005) 39–50
5. Duzan, G., Loyall, J., Schantz, R., Shapiro, R., Zinky, J.: Building adaptive distributed applications with middleware and aspects. In: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development. (2004) 66–73
6. Barreto, L., Muller, G.: Bossa: A language-based approach to the design of real-time schedulers. In: Proceedings of the 10th International Conference on Real-Time Systems. (2002) 19–31
7. Åberg, R., Lawall, J., Südholt, M., Muller, G., Meur, A.L.: 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. (2003) 196–204
8. Douceur, J., Bolosky, W.: Progress-based regulation of low-importance processes. In: Proceedings of the 17th ACM Symposium on Operating Systems Principles. (1999) 247–260
9. Newhouse, T., Pasquale, J.: A user-level framework for scheduling within service execution environments. In: Proceedings of the IEEE International Conference on Services Computing. (2004) 311–318
10. Newhouse, T., Pasquale, J.: ALPS: An application-level proportional-share scheduler. In: Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing. (2006) 279–290

11. Chang, F., Itzkovitz, A., Karamcheti, V.: User-level resource-constrained sandboxing. In: Proceedings of the 4th USENIX Windows System Symposium. (2000) 25–36
12. Elnikety, S., Nahum, E., Tracey, J., Zwaenepoel, W.: 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. (2004) 276–286
13. Welsh, M., Culler, D., Brewer, E.: SEDA: An architecture for well-conditioned, scalable Internet services. In: Proceedings of the 18th ACM Symposium on Operating Systems Principles. (2001) 230–243
14. Welsh, M., Culler, D.: Adaptive overload control for busy Internet servers. In: Proceedings of the 4th USENIX Conference on Internet Technologies and Systems. (2003)
15. Sun Microsystems: JSR 220: Enterprise JavaBeans, version 3.0 (2006)
16. The Carnegie Mellon Software Engineering Institute: Capability maturity model integration. `http://www.sei.cmu.edu/cmmi/`
17. JBoss Group: JBoss application server. `http://www.jboss.com/`
18. Apache Jakarta Project: Apache Tomcat. `http://tomcat.apache.org/`
19. Apache Struts Project: Apache Struts. `http://struts.apache.org/`
20. Seasar Foundation Project: Seasar. `http://www.seasar.org/`
21. PostgreSQL Global Development Group: PostgreSQL. `http://www.postgresql.org/`
22. Object Refinery Ltd: JFreeChart. `http://www.jfree.org/`
23. Chiba, S., Ishikawa, R.: Aspect-oriented programming beyond dependency injection. In: ECOOP 2005 – Object-Oriented Programming. LNCS 3586 (2005) 121–143
24. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An overview of AspectJ. In: Proceedings of the 15th European Conference on Object-Oriented Programming. (2001) 327–353
25. Apache Jakarta Project: Apache JMeter. `http://jakarta.apache.org/jmeter/`