Metrics-Based Analysis of Thread Behavior Using an Aspect-Oriented Programming Approach

Oral Alan
Information Technologies Institute
Research Center for Advanced Technologies
On Informatics and Information Security
Kocaeli, Turkey
oral.alan@bte.tubitak.gov.tr

Nadia Erdoğan
Computer Engineering Department
Faculty of Electrical and Electronic Engineering
İstanbul Technical University
İstanbul, Turkey
nerdogan@itu.edu.tr

Abstract—**Understanding thread behavior makes multi-threaded programming easier. Thread behavior can be observed by calculating metrics that are driven from thread state changes and time durations spent in those states. This paper presents the design and implementation of a new thread profiler that builds on aspect-oriented programming (AOP) approach and its use on multi-threaded applications. The new profiler differs from existing profilers in that it adopts a metrics based approach, calculating certain metrics for threads profiled. Also presented is a method to retrieve exact time information at the machine level from the JVM context. The paper concludes with a description of results obtained for thread profiling and metric calculations and a comparison of the overhead the profiler introduces with that of other profilers.**

Keywords: threads, thread profiling, aspect-oriented programming, metrics-based analysis, dynamic metrics, multi-threaded programs

I. INTRODUCTION

Performance concerns in a multi-threaded application naturally lead to issues related to thread behavior. To diagnose the cause of poor performance, one needs to observe thread behavior and gather information as the program executes in order to determine modifications in thread characteristics or resources that would result in better performance.

One of the most important problems in developing multi-threaded programs is that the behavior of threads on a real system cannot be predicted until they are run, and detailed information about the runtime behavior of threads is usually not available. If exact time and event information can be collected, metrics-based approaches can present useful information about the complex system being analyzed, and useful metrics that represent the behavior of threads can be calculated.

This paper presents an approach based on exact time and thread state information to analyze the behavior of multi-threaded applications. For this purpose, an aspect-based thread profiler has been developed, namely Java Aspect-Oriented Thread Profiler (JAOTP), which uses aspect-oriented programming to profile threads. To deal with the problem of collecting exact time information, a technique that uses the Java Native Interface (JNI) is applied. Through calls to native methods, information on timing and instantaneous cycle count of a logical processor is retrieved.

II. PERFORMANCE ISSUES OF MULTI-THREADED SYSTEMS

With recent advances in multi-processor based computers, threads and systems that use threads effectively have become more important. A thread is a small and simple unit of resource usage that corresponds to a logical flow in program execution. A thread of execution is the smallest unit of processing that can be scheduled by an operating system. In most cases, a thread is contained inside a process.

Multi-threading allows multiple threads to exist within the context of a single process. These threads share process resources but are able to execute independently, each with a certain responsibility. Thus, the multi-threaded programming model provides developers with a useful abstraction of concurrent execution: with responsibilities shared among threads, the process can handle more than one job at the same time. In addition, multi-threading allows a single process to enable parallel execution on a computer system that has multiple CPUs or CPUs with multiple cores. This is because the threads of the program naturally lend themselves to truly concurrent execution on one or more processors.

With the recent advent of symmetric multiprocessor (SMP) systems, the effective use of threads on those systems has become an important issue. Programmers needs to be careful to avoid race conditions and other non-intuitive thread behaviors. Otherwise, the performance of a multi-threaded program where threads are not synchronized correctly to coordinate their execution can easily fall below the performance of a sequential program. Therefore, programmers need tools to observe both the behaviors of threads and the overall performance of the system during program development.

Multi-threaded programs have complex behaviors that cannot be predicted before a system starts running. In the design phase of a system, architects and developers try to answer questions about how the multi-threaded system will behave. The runtime behavior of threads depends on several concepts such as thread priority, logical processor choice, the state of the thread and system delay. To understand and observe the behavior of threads, relevant information must be collected on the running system.

A. *Metrics-Based Methods for Program Analysis*

Metrics-based approaches to understanding the behaviors of programs are generally divided into two classes: static and dynamic analyses. Multi-threaded programs can also be analyzed using these techniques. In the static approach, there is no need to run the program as analysis is based on examination of the model. Behavior is predicted using static metrics such as complexity and lines of code. Dynamic analysis, also called profiling, is better suited to understanding thread behavior. The advantage of profiling is that it is directly related to runtime information. The nature of threads is itself complex, and this complexity increases with the underlying systems.

B. *Problems For Multi-Threaded Systems*

Thread profiling introduces challenges due to certain properties of the underlying systems and threads. Realizations of measured events occur at high frequency, and matching between the layers and the complex relationships between threads are the most important challenges to profiling multi-threaded systems accurately [1].

Due to the difficulties of profiling multi-thread applications, the results can change for each follow-up. Delays can be introduced according to the thread switching policy of the operating system.

C. *Metrics for Thread Behavior*

Dynamic metrics are based on time and event information. Accordingly, there must be an event that represents thread behavior. This paper uses the state changes of the threads as the basic profiling event. Time information is calculated for each state, and the metrics [2] can be constructed using this base information.

III. Profiling Methods

Operations to profile systems can be carried out before a program is running on a processor [1]. The aim of these operations is to add profiling information at that stage.

A. *Compiler-Based Profiling Methods*

The steps where profiling operations can be added are shown in Figure 1 [1]. These methods cover source-to-source transformation (shown as number 1 in Figure 1), static binary code conversion (shown as number 2), and dynamic binary code conversion (shown as number 3).

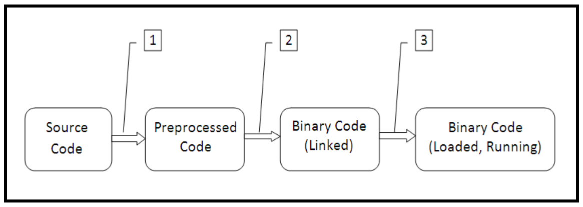


Figure 1. Steps Where Profiling Operations Can Be Added

Source–to-source transformation consists of transforming the source code to another source for adding the profiling logic based on the preliminary information that identifies points where code can be added. Proteus [3] is an example of source-to-source transformation. Static binary code conversion uses source code compilation with pre-prepared libraries that have profiling operations. Rational Purify and Quantify [4] is an example.

Dynamic binary code conversion is the most difficult and low-level way to add profiling information. It uses Just-in-Time (JIT) compilation. The advantage of this method is that profiling operations can be added to a program without compiling and are utilized only at the required time. Profiling can be disabled at desired times, thus reducing the profiling workload induced.

Paradyn [5] is a system that measures the performance of parallel programs and finds bottlenecks that reduce the performance of the program. For the node that is profiled, if there is a performance bottleneck that causes a performance decrease, profiling is repeated on the given resources and problems under that node. Paradyn uses hypotheses generated in this way in a hierarchy to determine performance problems in parallel systems.

B. *Operating System Profiling*

Applications use services offered by the underlying operating system. These services provide access to shared resources. Resources profiling may be possible using those transition points that are captured with appropriate structures, such as probes.

C. *Virtual Machine Profiling*

Virtual machines make it easy to profile threads. The virtual machine is located between a user program and the operating system. Byte code transformation is based on the addition of profiling information to the byte code, similar to adding profiling information using the transformation of binary code.

IV. Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) has emerged as a programming approach that adds features missing from object-oriented programming (OOP). The main difference is that AOP separates cross-cutting concerns from the business logic using a new unit of modularization called an *aspect*. Cross-cutting concerns are system-wide concerns that are used in most modules, and which have relations with other concerns. It becomes difficult to design these relations with OOP. By separating these concerns into separate aspects, the system becomes easy to manage and the business logic is simplified by eliminating the code that is related with other concerns. In the AOP approach, aspects are the central units of modularization and crosscut the system according to rules that define when to activate or deactivate the aspect during program execution.

The actual executing code that does the main job of the cross-cutting concern in an aspect is called an *advice*. The executing code of the aspect is applied according to rules based on the definitions of certain points in the program code called *joinpoints*. Joinpoints are places of interest where aspects should begin operations in the program execution. A method call, a method execution, or an object initialization can be joinpoints. The rules that define the execution of an advice on a joinpoint are called *pointcuts*. Advices are executed according to pointcuts on the joinpoints.

Profiling is a cross-cutting concern, unrelated to other modules and business logic, and can be defined as an aspect. The profiler's advice and joinpoints and the pointcuts can change according to the aim of the profiler. Using profiling as an aspect makes profiling modular and allows the advice to be changed without changing the actual code being profiled. Since it is separate from the other parts of the system, profiling can be applied to other modules.

A. *AspectJ*

Several implementations of AOP have been adapted into programming languages. The most widely used of these implementations are AspectJ [6], Spring AOP [7] and JBoss AOP [8]. AspectJ is an extension to the Java. Table I shows, in nanosec. the additional overhead of AOP implementations for a call before and after execution points [9].

TABLE I. AOP IMPLEMENTATION COMPARISON

AOP Comparison	**Aspect Werkz**	**AspectJ**	**JBoss AOP**	**Spring AOP**
Before, args() target()	10	10	220	355
After, args() target()	80	50	290	436

According to this table, AspectJ brings the least overhead to the execution points. Consequently, JAOTP is implemented using AspectJ because of its simplicity, its similarity to the Java language, and its minimal overhead on the source code.

V. JAVA THREADS

Threads are frequently used in Java programming, as in all other current programming languages. Java offers an application programming interface (API) for programmers to use in developing multi-threaded applications. To understand the behavior of Java threads, the states of the threads must be identified and time information must be collected between state transitions.

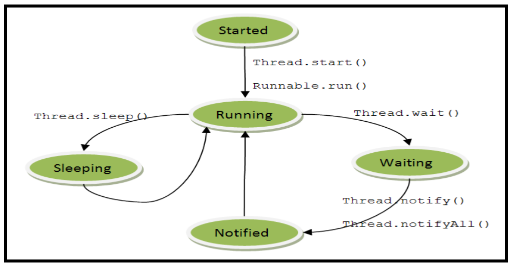


Figure 2. Thread States and Transitions

Figure 2 shows thread states and calls that result in state transitions in Java. In JAOTP, these calls are treated as joining points.

VI. JAOTP

The Java Aspect-Oriented Thread Profiler (JAOTP) is discussed in this section. We present in detail design decisions, the general architecture of the profiler and the viewer, implementation issues, and its application.

A. *JAOTP Architecture*

JAOTP consists of two main modules: the JAOTP Core and the JAOTP Viewer. The JAOTP Core is mainly responsible for handling thread state transitions using aspects. The JAOTP Viewer is a separate Eclipse-based plug-in for viewing the results of the JAOTP Core and calculating the metrics of the information that is received. The JAOTP Core sends the information that it collects through a socket connection to the JAOTP Viewer. The general architecture of JAOTP is shown in Figure 3.

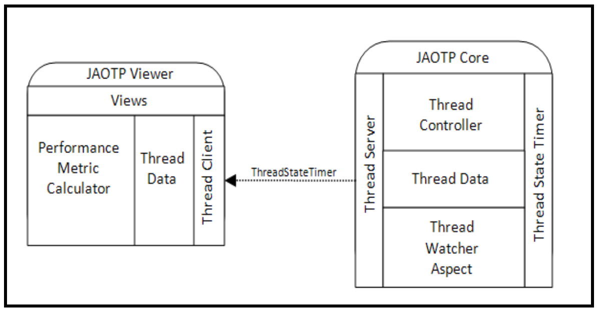


Figure 3. JAOTP General Architecture

A Thread Watcher Aspect is executed on the joinpoints, as shown in Figure 3. When the aspect is allowed to run according to pointcuts, the aspect calls the Thread Controller to run additional operations for thread profiling. The Thread Controller carries out operations in critical sections. Thus, after capturing a thread state, another thread is not allowed to operate until the thread registration is completed. The Thread Controller registers thread state change information to Thread Data, a map structure that holds thread state change lists, one for each thread. Thread Data maintains data on the thread number, the new thread state and instant time information. The registration operation also triggers an operation that calculates the time duration for the previous state change by subtracting the old time instant value from the new time instant value, and sets it into the previous time value. This operation involves Java Native Interface (JNI) calls to retrieve current time information from the microprocessor. It also uses a shared library that must be compiled under the platform on which the Java virtual machine is running.

The JAOTP Core sends thread state change information to the JAOTP Viewer over a socket communication. The Viewer updates the Thread Data lists with the information it receives. Next, the core sends ThreadStateTimer objects, which hold the information that is captured, and the Viewer adds them to the appropriate places in the lists. When the communication finishes, the Viewer can calculate the metrics using the information captured on the current state of the system. The Viewer also displays detailed information and calculated metrics about thread behaviors, allowing the developer to observe and understand the system.

Table II shows the pointcut names and their execution types as used in JAOTP according to Figure 3.

TABLE II. POINTCUTS AND TYPES USED IN JAOTP

Description	Pointcut Name	Pointcut Type
Thread Start Call	start()	call
Thread Start Call from Runnable Interface	Runnable+.run()	call
Thread Waiting Call	wait()	call
Thread Sleep Call	sleep()	call
Thread Notify Call	notify()	call
Notify Call For All Threads	notifyAll()	call

B. *Retrieving Processor Time Information From The Java Virtual Machine*

Threads can change states very quickly. JAOTP and metrics-based thread behavior analysis are dependent on correct timing information. The Java application development platform is a high-level development environment. In the experiments carried out for this paper, it was noted that thread transition tests on the basis of milliseconds actually have no use because threads can be switched in nanoseconds. Therefore, JAOTP is modified to support time measurement at the level of nanoseconds to get accurate results. To take measurements at nanosecond accuracy requires getting down to the lowest machine level in order to have instant access to the cycle number counter of the microprocessors. Java provides an interface called (JNI) to access such low-level features.

To obtain accurate time information between thread transitions, the JAOTP shared library needs to be compiled under the working environment that is used. Through this library, JAOTP invokes C programming language procedures that retrieve the logical processor number and the instant cycle counter by allowing the execution of assembly level machine codes that bring back this information. The instant cycle counter is handled by the read time-stamp counter (RDTSC) command [10] while the logical processor number is taken by the CPUID commands [19] supported by most general purpose microprocessors such as Intel Pentium and AMD Opteron.

To calculate exact time information, it is also necessary to know the processor's clock frequency. When JAOTP starts execution, it first gets the processor frequency through a procedure call by the shared library. With this information, the exact time elapsed between two thread states changes can be calculated according to equation (1), where the frequency used is in MHz (1 MHz = 1,000,000 Hz).

$$\# \text{nanosecond} = (\# \text{cycle} / \# \text{frequency}) * 1000 \qquad (1)$$

C. *Usage of JAOTP*

JAOTP is designed to be woven at compile time. The JAOTP Core and the AspectJ library must be added to a project that consists of one or multiple threads. The JAOTP Viewer can work with the Eclipse IDE if the developer adds the Viewer to the IDE. When the core is added to the project, the source code of the project is woven by the thread aspect; the joinpoints are found and the source code is woven into byte code by the AspectJ compiler. After the multi-threaded program starts, the Viewer developed with the Eclipse IDE can be used whenever the developer wants to see the results of the examination.

D. *JAOTP Metrics*

For each thread, JAOTP calculates three metrics that are based on collected time and event information. State change is the main event used in this work. Other metrics can be derived from time and state information. The metric base can be increased if another event type is collected. JAOTP uses the three metrics stated below to gather information about thread behavior:

- ***Response time:*** The total time elapsed between thread creation and termination.
- ***Utilization:*** The ratio of thread running time to the total response time.
- ***Critical time:*** the time in which a thread spends the longest period of time in its life cycle.
- ***Critical state:*** the state in which a thread spends the longest period of time in its life cycle.

VII. RESULTS

In this section, the usage of JAOTP on the producer-consumer problem and Apache Tomcat web server [12] with load testing is presented and the results are described.

A. *Producer-Consumer Problem*

The producer-consumer problem (also known as the bounded-buffer problem) is a classic example of synchronization and mutual exclusion in parallel programming. The problem describes two processes/threads, the producer and the consumer, which share a common, fixed-size buffer; a producer thread inserts data into the buffer, while a consumer thread deletes data from the buffer. The problem is twofold: to make sure that the producer will not try to add data into the buffer if it is full, and that the consumer will not try to remove data from an empty buffer. The problem can also be generalized to multiple producers and consumers.

B. *Apache Tomcat Web Server and Load Testing*

Apache Tomcat is an open source servlet container which provides a Java HTTP web server for Java web applications to run. It is a very popular application that is widely used in development and production environments.

Load testing is used to determine a system’s behavior by simulating the environment with the same number of users. In this work we used Apache JMeter [13], an open source Java application, to provide test functional behavior and used JAOTP on Apache Tomcat web server. We present the results with different number of users simulated with Apache JMeter. We need to use a web application that is close to real-world applications on Tomcat to get the results of JAOTP on production systems and compare with other profilers. LightPortal [14] is an open source Java based portal application that uses technologies such as Java Server Pages (JSP) [15], Spring framework [16], Hibernate [17]. The application is deployed to Apache Tomcat and the database is configured to run on MySQL [18], a widely used open source database.

C. *Test Applications*

JAOTP was used on the producer-consumer problem and on Tomcat web server to observe thread behavior. For the producer-consumer problem two sets of experiments were carried out: each with a varying number of producer and consumer threads and a buffer of fixed length in one set and a buffer of varying length in the second set. Tests were carried out on a system a 2.66 GHz Intel Core 2 Quad CPU and 4 GB memory, with Java 1.6.0_20 virtual machine running on the Ubuntu 9.10 operating system.

Experiment Set 1: This set of experiments involve a buffer of fixed length and varying number of threads, with both equal and different number of producer and consumer threads. Table III shows the computed metrics for two threads of each type. Table IV gives the results for the case where the number of threads is doubled. The results indicate a rise in thread response times which can be explained by the increase in the thread number and consequently increase in waiting time due to mutual exclusion. However, the critical time values increased in parallel with the response time. This shows that the balance between the producer and consumer threads has been preserved. Another observation is that the utilization values decrease as the number of threads increases, which can be explained by the longer periods of wait states.

TABLE III. JAOTP METRIC RESULTS WITH TWO PRODUCER THREADS, TWO CONSUMER THREADS AND A BUFFER SIZE OF FIVE

Thread	Response Time (ms)	Critical Time (ms)	Critical State	Utilization (%)
Producer-1	222.5	1.1	Waiting	0.159
Producer-2	220.6	6.9	Started	0.078
Consumer-1	221.9	1.5	Started	0.153
Consumer-2	151.4	0.5	Waiting	0.123

TABLE IV. JAOTP METRIC RESULTS WITH FOUR PRODUCER THREADS, FOUR CONSUMER THREADS AND A BUFFER SIZE OF FIVE

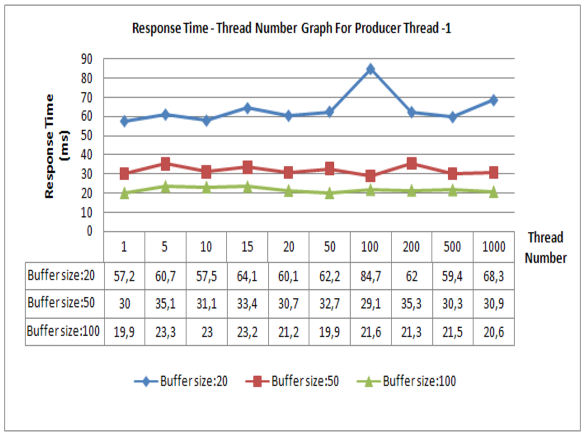
Thread	Response Time (ms)	Critical Time (ms)	Critical State	Utilization (%)
Producer-1	246.9	1.1	Waiting	0.093
Producer-2	244.3	4.1	Started	0.052
Producer-3	203.4	7.8	Started	0.052
Producer-4	203.3	7.8	Started	0.052
Consumer-1	245.9	1.5	Started	0.091
Consumer-2	203.7	0.6	Waiting	0.061
Consumer-3	203.3	7.8	Started	0.061
Consumer-4	203.2	0.5	Waiting	0.054

In Table V, we see the metrics for the case where the numbers of producer and consumer threads are different while the buffer size remains the same. As expected, the response times of the producer and consumer threads increase. This is because the balance between the producer and consumer threads has been destroyed, leading to longer wait periods. Utilization rates of consumer threads have also increased.

TABLE V. JAOTP METRIC RESULTS WITH TWO PRODUCER THREADS, FOUR CONSUMER THREADS AND A BUFFER SIZE OF FIVE

Thread	Response Time (ms)	Critical Time (ms)	Critical State	Utilization (%)
Producer-1	274.2	3.4	Waiting	0.127
Producer-2	271.7	8.5	Started	0.055
Consumer-1	273.3	3.4	Waiting	0.127
Consumer-2	219.3	3.2	Started	0.056
Consumer-3	219.2	3.2	Started	0.067
Consumer-4	186.6	0.8	Waiting	0.067

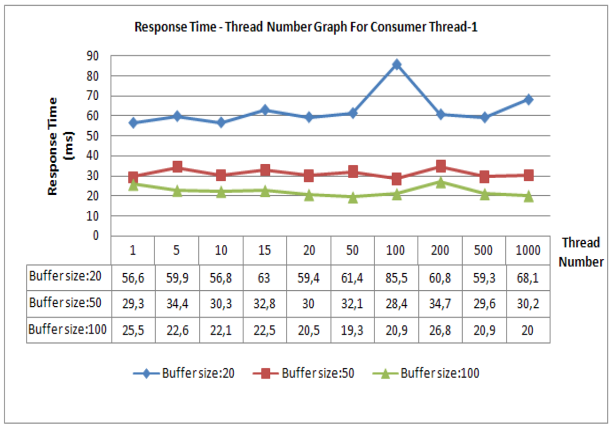
Experiment Set 2: The second set of tests were run using varying numbers of producer and consumer threads with varying buffer sizes to discover how metrics behave and whether they are useful in analyzing thread behavior. The figures below show the results of metrics for a single thread; the first producer thread and the first consumer thread are selected as representatives. The numbers of threads of each kind were changed, starting with a single thread and increasing the number to 5, 10, 15, 20, 50, 100, 200, 500 and 1000. Each experiment was repeated for three buffer sizes, 20, 50 and 100. The results are depicted in Figures 4-9.



	1	5	10	15	20	50	100	200	500	1000
Buffer size:20	57,2	60,7	57,5	64,1	60,1	62,2	84,7	62	59,4	68,3
Buffer size:50	30	35,1	31,1	33,4	30,7	32,7	29,1	35,3	30,3	30,9
Buffer size:100	19,9	23,3	23	23,2	21,2	19,9	21,6	21,3	21,5	20,6

Figure 4. Response Time—Thread Number Graph for First Producer Thread with Different Buffer Sizes

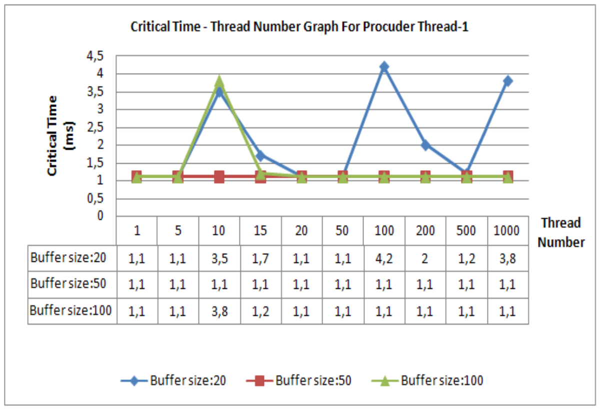
In Figure 4, the response time results are shown for Producer Thread-1 with different buffer sizes. Increase in the buffer size results in an improvement in performance, decreasing thread response time.



Thread Number	1	5	10	15	20	50	100	200	500	1000
Buffer size:20	56,6	59,9	56,8	63	59,4	61,4	85,5	60,8	59,3	68,1
Buffer size:50	29,3	34,4	30,3	32,8	30	32,1	28,4	34,7	29,6	30,2
Buffer size:100	25,5	22,6	22,1	22,5	20,5	19,3	20,9	26,8	20,9	20

Figure 5. Response Time—Thread Number Graph for First Consumer Thread with Different Buffer Sizes

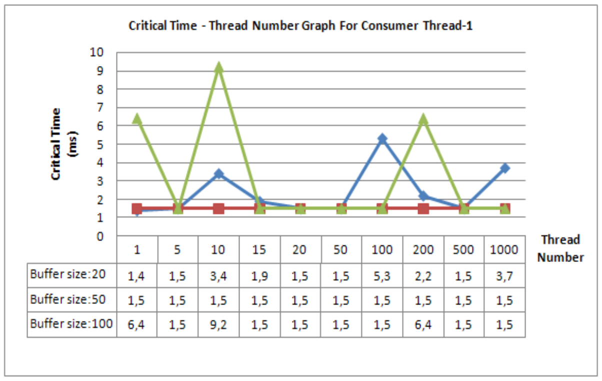
In Figure 5, the response time results are shown for Consumer Thread-1 with different buffer sizes. Similar to the Producer Thread-1, response time of the Consumer thread decreases with increasing buffer size.



Thread Number	1	5	10	15	20	50	100	200	500	1000
Buffer size:20	1,1	1,1	3,5	1,7	1,1	1,1	4,2	2	1,2	3,8
Buffer size:50	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1
Buffer size:100	1,1	1,1	3,8	1,2	1,1	1,1	1,1	1,1	1,1	1,1

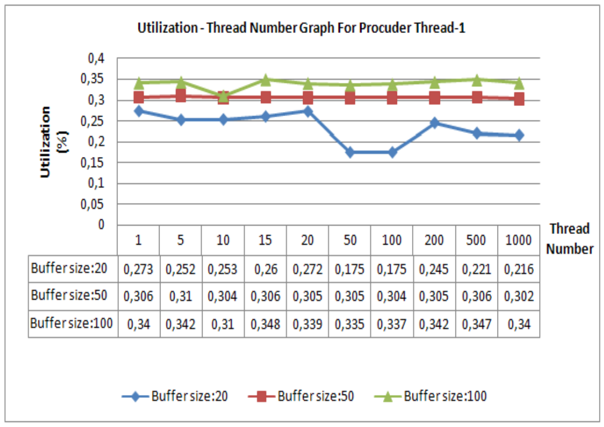
Figure 6. Critical Time—Thread Number Graph for First Producer Thread with Different Buffer Sizes

Figures 6 and 7 show change in critical time for the Producer Thread-1 and Consumer Thread-1 respectively, as the buffer size increases. The results show that buffer size does not have an impact on critical time.



Thread Number	1	5	10	15	20	50	100	200	500	1000
Buffer size:20	1,4	1,5	3,4	1,9	1,5	1,5	5,3	2,2	1,5	3,7
Buffer size:50	1,5	1,5	1,5	1,5	1,5	1,5	1,5	1,5	1,5	1,5
Buffer size:100	6,4	1,5	9,2	1,5	1,5	1,5	1,5	6,4	1,5	1,5

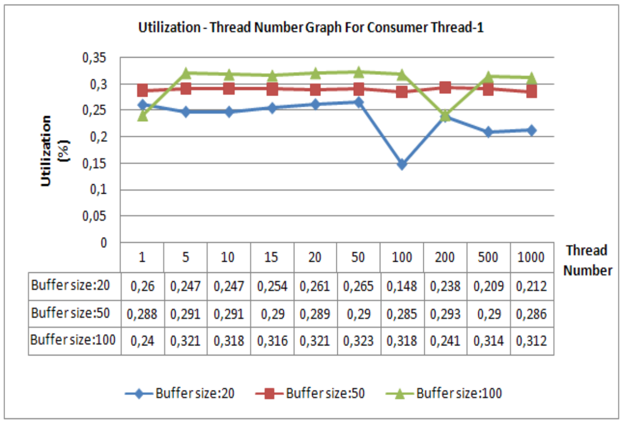
Figure 7. Critical Time—Thread Number Graph for First Consumer Thread with Different Buffer Sizes



Thread Number	1	5	10	15	20	50	100	200	500	1000
Buffer size:20	0,273	0,252	0,253	0,26	0,272	0,175	0,175	0,245	0,221	0,216
Buffer size:50	0,306	0,31	0,304	0,306	0,305	0,305	0,304	0,305	0,306	0,302
Buffer size:100	0,34	0,342	0,31	0,348	0,339	0,335	0,337	0,342	0,347	0,34

Figure 8. Utilization—Thread Number Graph for First Producer Thread with Different Buffer Sizes

Figure 8 and 9 show change in utilization for the Producer Thread-1 and Consumer Thread-1 respectively. In both cases, we observe that as the buffer size increases, the utilization of the threads generally increase as well. According to the challenges in thread profiling there can be some different situations on results like the experiment made using hundred producer and consumer threads. There is a dip in the figure and there may possibly be a thread switching or a lock operation that changes every thread to waiting. These situations can be seen when working with threads so they must be ignored and the general result graph made by all different experiments must be used to understand thread behavior.



Thread Number	1	5	10	15	20	50	100	200	500	1000
Buffer size:20	0,26	0,247	0,247	0,254	0,261	0,265	0,148	0,238	0,209	0,212
Buffer size:50	0,288	0,291	0,291	0,29	0,289	0,29	0,285	0,293	0,29	0,286
Buffer size:100	0,24	0,321	0,318	0,316	0,321	0,323	0,318	0,241	0,314	0,312

Figure 9. Utilization—Thread Number Graph for First Consumer Thread with Different Buffer Sizes

In general, results obtained from the profiler show that thread response times are closely related to buffer size. Critical time is generally higher when the thread number is low, as expected. Thread utilization decreased when the buffer size and the thread number increased. Based on these observations, the application developer may choose the right number of threads and the best size for the buffer suitable for its needs. The experiments show that the metrics defined can be useful to understand the behavior of threads in a classical problem by collecting profiling information.

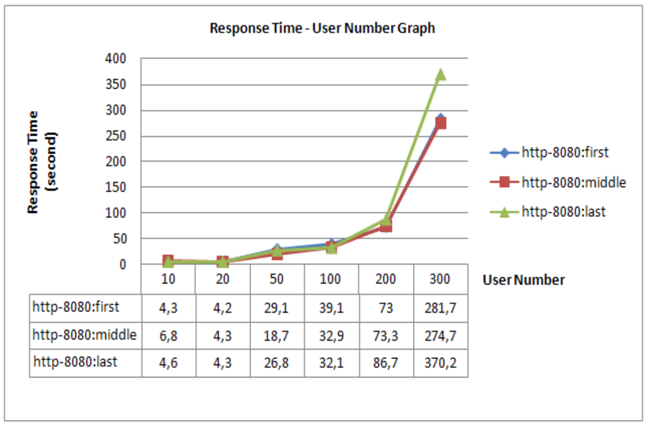
Experiment set 3: The third test uses Apache Tomcat web server with LightPortal web application deployed. The test functional behavior is provided with Apache JMeter, used to simulate the user number. In Table VI, the profiling results of JAOTP are presented on Tomcat web server with 5 users simulated by JMeter. As can be seen in the table, Tomcat creates certain internal threads such as http-8080-Acceptor and TP-Processor, along with user threads that are named as http-8080-i, where i vary between 1 and 5, the number of users present in the experiment. To show the utility of presented metrics, we have repeated the tests with different numbers of users and report the results for the first, last and the middle user. That is, if 10 users are present, results for http-8080 threads 1, 5 and 10 are presented.

TABLE VI. TOMCAT WEB SERVER LOAD TESTING WITH FIVE USERS

Thread Name	**Response Time (ms)**	**Critical State**	**Utilization (%)**
Container Background Processor	4337.5	Sleeping	0.00005
http-8080-Acceptor	3185.1	Running	0.67421
TP-Processor1	2147.9	Waiting	0.00014
TP-Processor2	2149.1	Waiting	0.00045
TP-Processor3	4301.6	Waiting	0.00150
TP-Processor4	2152.0	Running	0.99786
TP-Monitor	2157.6	Waiting	0.00002
http-8080-1	4296.5	Running	0.99962
http-8080-2	4572.8	Running	0.93923
http-8080-3	4302.7	Running	0.99820
http-8080-4	4297.0	Running	0.99952
http-8080-5	2405.8	Running	0.89647

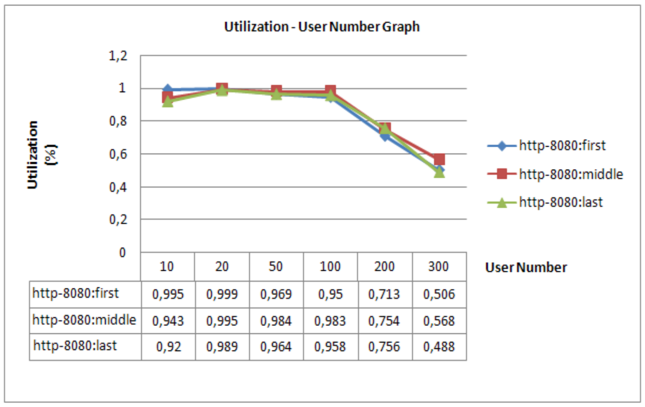
The web server creates a new thread for each user (client) and the thread fulfills the user request. The utilization rates are very high, showing that the web server is working healthy and its performance is good with these numbers of users.

The results of response time and utilization metrics are shown in Figures 10 and 11.



	10	20	50	100	200	300
http-8080:first	4,3	4,2	29,1	39,1	73	281,7
http-8080:middle	6,8	4,3	18,7	32,9	73,3	274,7
http-8080:last	4,6	4,3	26,8	32,1	86,7	370,2

Figure 10. Response Time— User Number Graph



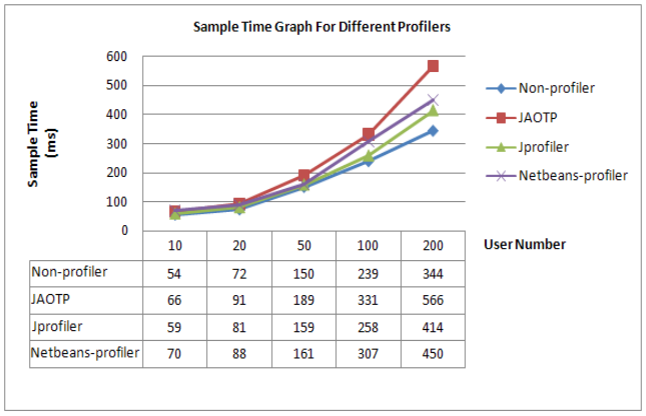
	10	20	50	100	200	300
http-8080:first	0,995	0,999	0,969	0,95	0,713	0,506
http-8080:middle	0,943	0,995	0,984	0,983	0,754	0,568
http-8080:last	0,92	0,989	0,964	0,958	0,756	0,488

Figure 11. Utilization—User Number Graph

The results show that the performance of the web application starts to decline after 100 users. This is clearly seen with increase in the response time and decrease in the utilization of the threads. This implies that the threads spend more time in waiting state, rather than doing useful work. Hence, the utilization metric provides a valuable input to the assessment of behavior and can be used to tune the application for better performance.

D. Comparison of the JAOTP With Other Profilers

One of the most important values for a profiler is how much overhead it brings. Figure 12 shows a comparison of JAOTP with the other well-known profilers JProfiler [19] and NetBeans Profiler [20]. The comparison uses the JMeter average sample time results with the LighPortal web application. Results are listed in milliseconds.



	10	20	50	100	200
Non-profiler	54	72	150	239	344
JAOTP	66	91	189	331	566
Jprofiler	59	81	159	258	414
Netbeans-profiler	70	88	161	307	450

Figure 12. Profiler Overhead Comparison

Compared to the other well-known profilers, JAOTP seems to bring a higher overhead. However, as it provides the software developer with more detailed results at the level of thread state change, it may be more desirable to use JAOTP to make certain design decisions if the application requires profiling at such levels.

VIII. Conclusion

In this paper an aspect-oriented thread profiler was developed and a method to find and record exact time information was discussed. The profiler was tested on a producer-consumer problem and Tomcat web server. Results show that the aspect-oriented approach can be easily used, and the metrics-based analysis method gives developers results regarding how their threads behave in the development phase of their programs. In addition, Java needs to access some machine level information for such profiling; the paper also presents the methods used to get such information in the Java context. The overhead introduced by JAOTP is measured to be slightly higher than that of other profilers. However, JAOTP adopts a metrics-based approach, computing specific metrics for each thread, different from the other profilers. The information collected that consists of native method calls and detailed state information is valuable for thread profiling. The native calls use a shared library so that platform dependency decreases according to the platforms that the shared library supports.

The decision to implement the profiler with AOP approach, using AspectJ, has proven to be helpful. This is mainly because separating the nonfunctional profiling logic of an application from its functional logic has helped to achieve an efficient and effective profiler. As a result, JAOTP is an easy to use thread profiler, with an easily extensible modular structure.

JAOTP functionality provides several metrics that help analyze thread performance and behavior. With these metrics, the application developer can analyze various aspects of program execution, in order to find out potential bottlenecks or failure conditions. For example, JAOTP may be used to monitor producer-consumer relationships. Also, for the case where threads are dependent on resources, with JAOTP it is possible to track the life cycles of threads, determining the periods of time they are in a blocked state, waiting for a resource to become available. In "input-processing-output oriented" applications, such as a Web application, *utilization* and *response time* metrics of JAOTP allow to assess the performance of the system, showing how long it takes to process user requests and produce results.

Having these conclusions, JAOTP, with its design and implementation decisions, proves the suitability of AOP techniques and the use of AspectJ for thread profiling.

IX. References

[1] D. G. Waddington, N. Roy, D. C. Schmidt, "Dynamic Analysis and Profiling of Multi-threaded Systems," Proceedings of the 2nd International Workshop on Social Computing Behavior Modeling and Prediction, Phoenix, AZ, March 31-April 1, 2009.

[2] J. K. Hollingsworth, B. P. Miller, "Parallel program performance metrics: A comparison and validation," in Proceedings of Supercomputing, November 1992.

[3] D. G. Waddington, B. Yao, "High Fidelity C++ Code Transformation," Proceedings of the 5th Workshop on Language Descriptions, Tools and Applications, Edinburgh, Scotland, UK, 2005.

[4] IBM Corporation Whitepaper: Develop Fast, Reliable Code with IBM Rational PurifyPlus, 2003.

[5] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, "The Paradyn Parallel Performance Measurement Tool," IEEE Computer, vol. 28., n.11, p. 37-46, 1995.

[6] G. Kiczale, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, "An Overview of AspectJ," Lecture Notes in Computer Science, 2072, p. 327-355, 2001.

[7] Aspect Oriented Programming with Spring, http://static.springsource.org/spring/docs/2.5.x/reference/aop.html.

[8] JBoss AOP – Aspect-Oriented Framework for Java, JBoss AOP Reference Documentation, http://www.jboss.org.

[9] AOPBenchmark, http://docs.codehaus.org/display/AW/AOP+Benchmark

[10] Time_Stamp_Counter, http://en.wikipedia.org/wiki/Time_Stamp_Counter

[11] CPUID, http://en.wikipedia.org/wiki/CPUID

[12] Apache Tomcat web server, http://tomcat.apache.org/

[13] Apache JMeter, http://jakarta.apache.org/jmeter/

[14] LightPortal, open source portal server and social collaboration software, http://www.lightportal.org/

[15] Java Server Pages, http://java.sun.com/products/jsp/

[16] Spring Framework, http://www.springsource.org/

[17] Hibernate, http://www.hibernate.org/

[18] MySQL database, http://www.mysql.com/

[19] JProfiler, http://www.ejtechnologies.com/products/jprofiler/overview.html

[20] NetBeans Profiler, http://profiler.netbeans.org