



WHITE PAPER

Application Performance Management

**The Case for Adaptive Instrumentation in J2EE
Environments**

Why Adaptive Instrumentation?	3
Discovering Performance Problems.....	3
The adaptive approach	6
Step 1—Take the survey.....	6
Step 2—Analyze the survey data.....	7
Step 3—Instrument selected methods	7
Appendix A — Glossary.....	9

Why Adaptive Instrumentation?

Measuring and isolating performance bottlenecks in production J2EE applications requires an efficient data collection technology that provides detailed performance metrics with minimal overhead. Excessive overhead is usually the result of over-monitoring; therefore, deciding which components of the application should be monitored is a key issue. Traditionally, this has been a manual trial-and-error process, where the application structure is examined and components that could potentially cause a bottleneck are added to the list of monitored components. If undetermined performance issues still exist after all the components on the list are monitored, a manual analysis process is employed until the root cause of the remaining performance problems is found. During this process, if too many components are monitored, the associated overhead will affect the system. Some components will have to be removed from the list of monitored components in order to maintain reasonable overhead.

This process is tedious, time-consuming and requires a significant amount of developer experience. It is typically repeated during application testing as well as after the application is deployed in production, since the test environment is often different from the target production environment.

Also, changes in an application's Java code, workload patterns and database behavior frequently introduce new performance problems. Any of these changes will require additional manual refinement to the process described above.

The question is, how can we improve this process? The answer is Adaptive Instrumentation.

Adaptive Instrumentation is a patent-pending technology that automatically discovers and isolates the major contributors to poor performance in Java™ 2 Platform Enterprise Edition (J2EE) applications while minimizing overhead. This paper reviews the traditional approach to monitoring J2EE applications, lists problems associated with this approach, and explains how Adaptive Instrumentation resolves these problems. A glossary of basic terms used throughout this document is included in Appendix A.

Discovering Performance Problems

Most J2EE application performance monitors gather information on applications by placing probes in key locations throughout the application code. The probes gather performance data on specific components as the application executes. By gathering and analyzing the data collected by the probes, an application performance monitor can provide accurate information on the overall behavior of the application, point to bottlenecks in the application and recommend necessary adjustments to alleviate those bottlenecks.

Inserting probes in the right places to balance the need for detail with acceptable overhead is a key factor in measuring performance correctly. For J2EE applications, the most elementary unit of work is a Java method. To determine the amount of time spent in each method, measurements are taken before and after the method is invoked. By looking at the difference in measurements (e.g., time, memory usage) before and after a method is invoked, we can understand the individual method's contribution to the overall performance of the application. We can then record the data in a performance database for further analysis.

The process of inserting the probes into the application code is called *instrumentation*. Precise for J2EE pioneered the use of a technique called *byte-code instrumentation* which allows insertion of probes into classes' and methods' binary code. This technique removes the need to access the application's source code and thereby allows instrumentation of any Java application. Precise byte-code instrumentation is done at run-time as the classes are loaded.

Which methods should be monitored?

A comprehensive approach applies instrumentation to all methods in the application so that the maximum amount of information is collected on the performance of that application. This approach, however, is limited to small applications in development environments since data collected by the various probes does not come for free. Taking measurements and recording them consumes expensive system resources and affects the executing applications. If all methods are instrumented, the application may suffer severe performance degradation. In some extreme cases, more time may be spent on recording performance data than on executing the application itself.

A dilemma arises between the need to maximize visibility into the application's performance and the need to minimize the overhead created by taking the measurements. As we increase the instrumentation level, more time is spent on measuring performance which results in a slower application. We need to strike a balance between the overhead caused by performance management and the visibility needed to identify and isolate performance bottlenecks.

How to manually find the right balance?

Most J2EE applications use well-known interfaces that are part of the J2EE specification. Some of these interfaces are Java Server Pages (JSPs), Enterprise JavaBeans (EJBs) and Java Database Connectivity (JDBC) drivers. Generally, byte-code instrumentation of these interfaces provides automatic identification of the response time contributions related to the use of relational databases, EJBs, and Internet user interfaces. Precise for J2EE and similar J2EE performance monitors usually instrument all or most of the standard J2EE interfaces.

However, J2EE applications often spend significant amounts of time performing operations outside the scope of common J2EE interfaces, such as communicating with legacy applications, using third-party packages, and executing logic that is unique to the application. To measure the performance of these components, we need the flexibility to apply instrumentation to a broader set of methods and classes. Precise for J2EE provides this capability through a feature called *custom instrumentation*.

Custom instrumentation

The custom instrumentation feature of Precise for J2EE allows users to monitor the performance of and obtain the drilldown invocation context of any Java method. However, when customizing the list of instrumented methods, users must explicitly specify a method name, which means they must know the methods to instrument. Discovering the right methods to instrument is a tedious, iterative, and time-consuming process that requires detailed knowledge of the application.

Custom instrumentation involves adding specific methods to the instrumented methods list. Typically, the underlying J2EE application must be restarted whenever the list of methods to be instrumented changes. Restarting the application server interrupts running applications, thereby

impacting availability—not an optimal scenario for mission-critical web-based applications. This re-instrumentation process must be repeated until the root cause of a performance problem is isolated.

Overhead increases as additional methods are custom instrumented. However, there is no way to quantify the amount of overhead associated with each additionally instrumented method. The only way to know that the overhead is too high is to over-instrument, watch the effect, and then back out some or all of the new instrumentation. To back out instrumentation, we need to decide which instrumented methods should be removed from the list, thereby reducing the overhead to an acceptable level. This process of trial and error imposes serious limits on our ability to monitor applications. It requires prior experience and understanding of the anticipated effects of instrumenting each method.

Summary of Problems with the Manual Approach

As we have seen, performance management of J2EE applications is conducted in an environment that poses significant challenges to developers, performance analysts, and others charged with the responsibility of providing smoothly running applications with acceptable performance and service levels. We summarize these challenges here, before describing how Adaptive Instrumentation minimizes or even eliminates them:

- **Limited visibility**—Default instrumentation does not necessarily cover the important application components. There is a need to manually discover bottlenecks and key components using intimate knowledge of the application.
- **Slow discovery**—An iterative discovery process is required by which you slowly reveal the “hot spots” in the application.
- **Manual adjustments**—The need to manually add each discovered method to the list of instrumented methods using Custom Instrumentation.
- **Impact on availability**—The need to restart applications and/or application servers for the new instrumentation to take effect, thereby interrupting availability.
- **Overhead assessment**—The inability to understand the overhead associated with the instrumentation of each additional method. In the absence of prior knowledge of problem areas in applications, IT organizations tend to over-instrument; this trades off greater visibility at the expense of overhead.

The adaptive approach

Precise for J2EE now employs Adaptive Instrumentation to address all of the problems described above. Adaptive Instrumentation uses advanced (patent pending) technologies to measure and collect a high-level view of an application's activity. It then uses this information to:

- Automatically choose the best candidate methods for instrumentation
- Filter the candidate method list to limit the overhead to a user-specified level
- Automatically generate a list of candidate methods for instrumentation
- Apply the instrumentation without restarting the application

The adaptive approach significantly reduces the time and effort necessary to isolate and resolve application performance issues, minimizes the need for detailed application knowledge, and avoids any negative impact on application availability.

How does Adaptive Instrumentation work?

Adaptive Instrumentation is a three step solution:

1. **Survey**—Monitor all of the methods in the application with ultra-low overhead and record key information that allows Adaptive Instrumentation to evaluate the contribution of each monitored method to the overall performance.
2. **Analyze**—Analyze the information gathered on all the methods in the application and determine which method should be monitored, given an overhead budget.
3. **Instrument**—Dynamically instrument the selected methods, avoiding the need to restart the application.

These three steps allow the Adaptive Instrumentation feature to choose which methods should be instrumented given a certain overhead budget. To help it make the right decisions in choosing which methods to instrument, we need to provide Adaptive Instrumentation with the amount of overhead (expressed as a percentage of the application's response time) that we consider acceptable.

Step 1—Take the survey

The purpose of the survey is to map the contribution of each and every method in the executing application. During the survey, all invoked methods are measured for their contribution to overall performance. Since instrumenting all of the application methods may result in high overhead, the survey relies on the following principles:

Only collect data that is absolutely essential to the evaluation process. This data includes the total response time and the invocation count for each method, along with enough data to construct a call graph of the application logic.

Run for a limited amount of time to reduce the impact of the survey's overhead. The amount of time that the survey runs is determined by the user. It must be long enough to allow all important components of the application to execute. The survey cannot measure methods that were not invoked.

Step 2—Analyze the survey data

Once the survey ends, we have a list of all methods that were invoked in the application, the amount of time spent in each method and the number of times each method was invoked. Using the call graph collected by the survey along with the response times and invocation counts, the Adaptive Instrumentation technology now examines the results and decides which methods should be instrumented. A well defined set of principles guide Adaptive Instrumentation in choosing which methods to instrument. Some of these principles are:

- Do not instrument methods with high invocation counts.
- Instrument methods in descending work time order.
- Instrument call tree paths throughout the application to isolate the largest contributors to response time in the context of the application logic.
- Do not exceed the user-defined overhead budget.

Step 3—Instrument selected methods

Method instrumentation is applied starting with the default instrumentation, which includes the most common J2EE interfaces: Java Server Pages, Enterprise JavaBeans, and JDBC calls. We then instrument methods that were selected in Step 2. Instrumentation of methods stops as soon as the allocated overhead budget is reached. With very small overhead budgets, it is possible to have only a small portion of the default instrumentation applied. The ability to control the amount of overhead caused by instrumentation is a unique feature of Precise for J2EE known as over-instrumentation protection.

Over-instrumentation Protection

Over-instrumentation protection allows users to specify a maximum acceptable overhead beyond which instrumentation should not be applied, regardless of whether it originates in the default instrumentation (e.g., JSPs) or in an Adaptive Instrumentation survey. It also constantly monitors the overhead level and will automatically disable instrumentation if the calculated overhead exceeds the predefined thresholds. This unique capability protects applications from unexpected load and from changes in the code that were unknown at the time the Adaptive survey was taken and that may result in excessive overhead.

Using multiple surveys

The ability to run an automatic survey of an executing application is very powerful and paves the way for newer and more sophisticated ways of using instrumentation. Some possibilities are:

The Case for Adaptive Instrumentation in J2EE Environments

- **Fine-Tuning**—Experiment with different overhead budgets to further fine tune the level of instrumentation. Users can run multiple surveys with different budgets and find the best balance between detail level and amount of overhead.
- **Multiple Instrumentation Sets**—Use different instrumentation sets for different times, since applications may behave differently at different times. For example, there may be a difference between the behavior of an application during the week and its behavior during weekends, or even differences between morning and afternoon peaks on the same day. Users can use Adaptive Instrumentation to run multiple surveys and apply different instrumentation sets at different times.
- **Pre-Survey**—Run the survey in a preproduction environment and apply the results to the production environment. In a staging environment, users can run the adaptive survey in the preproduction system with different budgets until they are satisfied with the instrumentation set. The results are then copied to the production environment and used there.

Summary

Adaptive Instrumentation is a state-of-the-art technology that allows developers, administrators, and performance analysts to maximize their visibility into applications while minimizing application overhead. It allows users to:

- Automatically discover and isolate the “hot spots” in a J2EE application’s business logic context
- Instrument methods without restarting applications
- Protect against over-instrumenting applications

Adaptive Instrumentation is a key component of Precise for J2EE, a comprehensive performance management solution for J2EE applications.

Appendix A — Glossary

Byte Code—The machine language interpreted by the JVM.

Byte-Code Instrumentation—Byte code, variables, and methods inserted into a Java class to collect information on or modify program behavior.

Hot Spots—Methods in an application that are the highest contributors to the overall performance of the application.

Invocation Count—The number of times a method was invoked by other methods.

JVM—The Java Virtual Machine, a program that executes byte code. Sometimes referred to as VM.

Overhead—The amount of time spent on gathering and recording performance data versus executing business logic.

Over-Instrumentation Protection—The ability to automatically disable instrumentation that exceeds a given overhead budget.

Response Time—The total amount of time spent inside a method or an application.

Work Time—The amount of time spent by a method or an application executing, excluding calls it made to other methods/services.

About Precise Software Solutions

Commitment, Focus, Experience

For over 15 years Precise Software Solutions has helped our Global 2000 customers manage business performance in complex, heterogeneous environments, assuring availability and business continuity. Precise offers a complete solution – from discovery through ongoing management – that allows our customers to focus on their core business. We offer the broadest platform support, in terms of enterprise application, operating system, database, and development environment coverage. Precise is the solution of choice for IT as an organization-wide standard for application management.

Visit our Web site

www.precise.com

To speak with a Product Specialist in the U.S.

Call toll-free 1 (877) 845 1886. For specific country offices and contact numbers, please visit our Web site.