Peeling the Layers of the "Performance Onion"

John Murphy
CEO, Crovan

The end of an IT project is often dogged by failure to meet system performance goals. To make matters worse, it often takes multiple fix-test cycles before the software is ready to ship and the project manager is finally happy. These multiple fix-test cycles are analogous to peeling back the layers of an onion: when one problem is fixed, then (and only then) another is found - a process that is frustrating to all involved. This paper examines a unique prediction tool from Crovan that has been developed to identify all the problems in the first test cycle (peeling all the layers of the onion at once), saving substantial time and trouble.

Introduction: The Problem

Have you ever noticed that as freeways are extended and built to relieve congestion the bottleneck simply moves to another part of town? This is not due to poor insight of the traffic planner, but because often the first bottleneck simply obscures the second and subsequent bottlenecks. Upgrading a congested freeway from 3 to 5 lanes may simply mean that more people get to the traffic-light queue quicker.

Unfortunately such problems are not simply the domain of traffic planners, but also those of the IT professional. The "performance bottleneck" that appears in the performance test simply masks additional bottlenecks that lie further along the path to reaching your system's performance goals - akin to peeling the layers from an onion.

These layers of performance bottlenecks cause a series of fix-test cycles where the testers fail the system and the developers identify and fix the bug, only for a subsequent bottleneck to appear resulting in another round of fixing and testing.

To make matters worse, performance testing is often left to the end of the development cycle where the cost of redesign is high and time is critical. Program managers need to be certain that the performance bugs can be fixed in a definable timescale. Any prolonged fix-test cycle will destroy customer confidence that the system will ever be finished.

An example

Here is an example based on the author's experience; it concerns a company that built a large IT system based on a software vendor's product, which it customized using a system integrator and its own staff.

With 2 months to go before the launch date, the performance test phase began and was expected to last 1 month. Within the first week of the testing it was clear that the system did not meet its performance goals. Unfortunately, but not uncommonly there followed a 2-week blame-storming exercise between the customer, integrator and software supplier. Finally, the data centre upgraded its hardware by doubling the CPU capacity. However, the next round of performance tests still failed to meet the performance goals and a further week was spent changing configuration options and reconfiguring the operating system kernel. The next round of performance tests still failed to meet the performance goals and so a performance code review was ordered to identify and remove inefficient coding methods. This review and debugging took 3 weeks. The next round of performance tests still failed to meet the performance goals. Developers discovered several calls to off-line information providers that were performed in series rather than parallel and special code was developed to perform these steps in parallel, which took another 3 weeks. The next round of performance tests still failed to meet the performance goals. The software supplier promised a full code review of the underlying code, instrumentation and a performance improvement plan that would take 3 months. Eventually, 5 months later the system passed its performance goals, but left a tattered relationship between customer, integrator and supplier.

Is there a better way?

Following a failure in performance, traditional approaches have centered on instrumentation and profiling tools, but these are often only aimed at identifying the first performance bottleneck. Typically, a profiler will identify where a program spends most of its time because this part of the code is often the inefficient section that needs improvement. However, the profiler will not tell you by how much the code needs to be improved, or whether there are any additional sections of code that also need to be optimized in order for the performance goal to be met.

© Crovan 2004

Furthermore, as well as identifying the performance bottleneck for a particular system configuration at a particular workload, engineers often also want to know how it performs under different workload conditions and on different hardware configurations. A profiler cannot help with these either.

Part of the research programme for the Performance Engineering Laboratory (PEL), a joint research group across University College Dublin and Dublin City University, has been to investigate solutions to the problems of the "Performance Onion" discussed above. Arising out of this research, Crovan has developed a tool with 4 analysis levels to address the performance issues outlined here:

InSight: Detecting software performance bottlenecks (unraveling the Performance Onion)
MoreSight: Identifying ways of correcting the bottlenecks
ForeSight: Predicting performance for different hardware
ClearSight: Predicting performance for different usage scenarios

Crovan Approach

Crovan has developed and built PredictorV to solve the problems of the Performance Onion in real systems.

The tool is based on a framework for Performance Monitoring, Modelling and Prediction of component-based systems. The aim of the framework is to capture the information from a system and automatically generate a performance model. Then the performance model is used to determine quickly where the performance problems are in the system.

One advantage of this approach is that it reduces the time and skill required to build a model, because information to build the model itself is captured directly from the system. Another advantage is that as much of the process as possible is automated, so it reduces the risk of human errors being added into the model.

The tool comprises 3 modules, shown in
Figure 1: a module to **monitor** the system, a module to **model** the transaction flow on that system, and a module to **predict** the future performance of the system. Having this predictive capability enables key performance questions to be answered and the Performance Onion to be unravelled.

Although the framework is generic, it was felt that to achieve a successful implementation it was important to reduce the scope of the project.

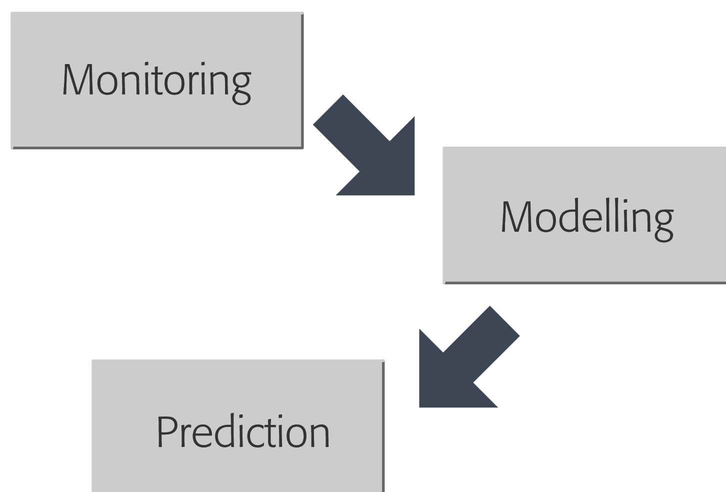

Figure 1: PredictorV Modules

Therefore the current tool development has concentrated on solving performance problems for J2EE systems. It was decided to concentrate on this platform as it is a technology well suited to, and widely used for, large-scale system developments in industry.

The next three sections give examples of each of the three main PredictorV modules in more detail.

Monitoring Module

The aim of the monitoring module is to collect enough information from the system under test so that a predictive model can be built. To do this it needs to obtain structure information (what the system does), represented by a call graph, and resource information (how much resource does it use) for each of the business transactions. The best method for collecting this information in a java-based system is to use profiling.

Profiling, in a broad sense, is the ability to monitor and trace events that occur during run-time, the ability to track the cost of these events, and the ability to attribute the cost of the events to specific parts of the J2EE program. For example, a profiler may provide information about which portion of the program consumes the most amounts of CPU time, or about which portion of the program allocates the most amount of memory. The profiler is used to collect information related to the resource usage of Java applications. Java applications are written in the Java programming language, and compiled into machine-independent binary class files, which can then be executed on any compatible implementation of the Java virtual machine. The Java Virtual Machine (JVM) is a platform-independent execution environment for compiled Java code. The profiler architecture is shown in
Figure 2.

© Crovan 2004

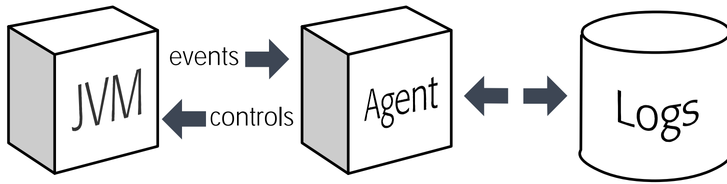

Figure 2: Profiler Architecture

The profiler makes use of the Java Virtual Machine Profiler Interface (JVMPI), a C interface to the JVM, where profilers may gain access to information relating to the JVM's current state. Using the JVMPI, the profiler collects memory and CPU information which is recorded in log files for each business transaction.

Modelling Module

The modelling module is responsible for displaying the call graph for each business transaction that has been collected using the monitoring module. To display the call graph the UML (Universal Modelling Language) Event Sequence Diagrams notation was chosen.

UML is a visual language that can be used for developing software systems. It can be used for the analysis and design of software systems. Various diagram types are available and it is possible to create profiles for the modelling of particular domains. UML is not a programming language or a development process.

UML emerged from the notation wars of the mid-1990's where software engineering gurus were promoting several slightly different notations for describing software systems. In 1995 the fathers of UML (James Rumbaugh, Grady Booch and Ivor Jackobson) joined Rational Inc and developed UML. In 1997 the UML partners and others submitted their proposal to OMG (Object Management Group) who took on the responsibility for developing the standard.

An Event Sequence Diagram shows the path of a transaction across the objects within the system. An example is shown in Figure 3. The objects are shown as vertical lines and the messages are shown as horizontal arrows. The blue rectangles are called Execution Occurrences.

Execution Occurrences in PredictorV are annotated with profile data showing the CPU and memory that are consumed when an execution occurrence is invoked.

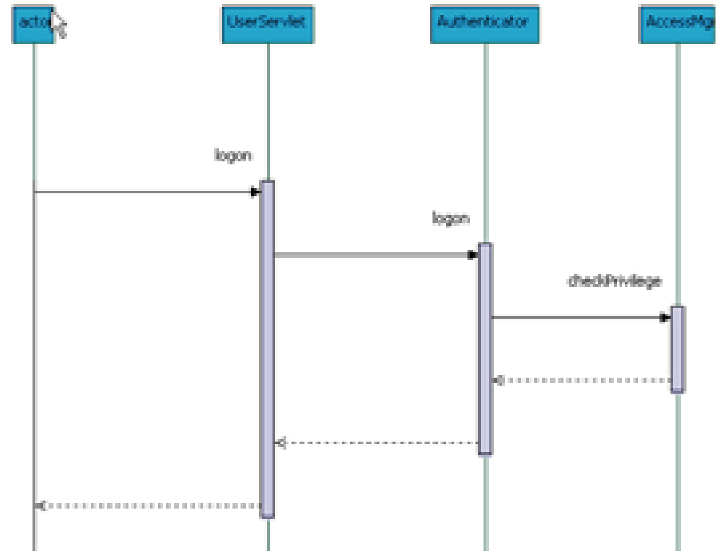

Figure 3: Example Event Sequence Diagram

Prediction Module

The prediction module takes the UML models and performs the various predictions mentioned above.

There are various different predictive techniques that could be used. The two prime candidates are analytical models and simulation models. Although an analytical model has the advantage that results can be generated very quickly, they have several disadvantages over simulation models as they cannot easily model:

· Spikes in workload, or bursts in arrival patterns
· Priority Schemes
· Parallel Processing
· Locks and Semaphores
· Percentile Statistics

Therefore, it was decided to use a discrete event simulation approach in the predictive module. Considerable effort was required to understand and model the complexity of the application server.

In addition to the simulation model, design tools for executing the InSight, MoreSight, ForeSight and ClearSight simulations have been produced as well as tools for displaying the result information. The user interface has been implemented as an Eclipse plug-in and a screenshot is shown in Figure 4.

© Crovan 2004

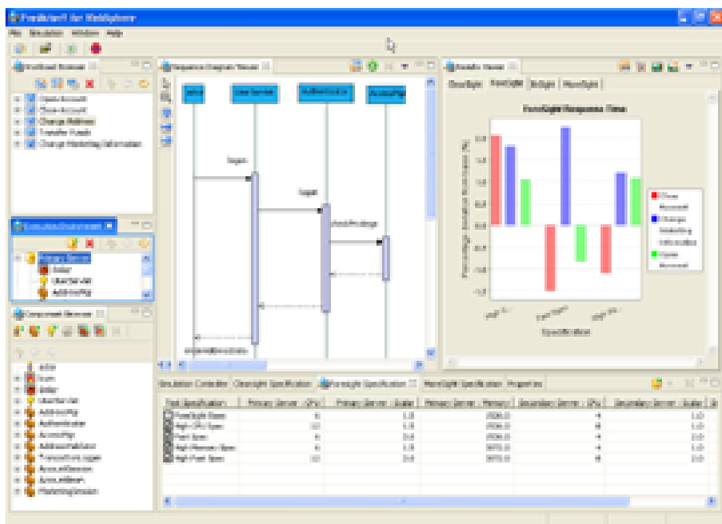

Figure 4: PredictorV Screenshot

PredictorV in Use

To illustrate how the tool can be used to peel the Performance Onion, a slightly contrived example is given below:

MegaCorp have just developed its new on-line Pizza ordering service and is about to conduct the performance test phase. MegaCorp is a progressive organization or, more importantly, its last "big" project was plagued by performance problems so project managers decided to use PredictorV.

Firstly, MegaCorp profiles each of the key business transactions that it expects to make up most of the workload. Using the profiler attached to the test system, each business transaction is executed and profiler files are saved. These files are then imported into the modelling module and the UML event sequence diagrams are displayed.

Now, using the predictive module, MegaCorp is almost ready to start peeling the onion. Next it must define the workload that it is interested in. To do this the number of concurrent users expected to execute the business transactions during the peak hour must be defined and any expected user think times must be added in.

Next the base simulation is performed and the results show that although the CPU and memory resources are under-utilized, the test fails to meet the performance goals. A further look at the results shows that there are not enough application server threads allocated to the server so these are increased.

Again, the base simulation is performed and the results show that the CPU and memory resources are still under-utilized, but the test fails to meet the performance goals. A look at the results shows that not enough bean instances have been created for a particular bean, so these are also increased.

The next simulation shows that CPU is now being fully utilized but the system still fails to meet the response time targets. Using InSight predictions the critical areas of code are discovered through the use of sensitivity analysis. However, just knowing the critical code does not tell the developers or managers how much of the code needs to be changed and by how much. Next using MoreSight the predictive model steps through a series of progressive "what-if" scenarios ranging from: what if the most critical method call is improved by 90%, through to: what if the top 10 most critical calls are improved by 10% (the search space can be specified by the user). After this analysis the user is shown what optimization strategies work, and hence how many layers of the onion must be peeled before performance goals are achieved. The advantage is that the simulation analysis can be completed in hours rather than months of fix-test cycles.

Luckily for MegaCorp its performance onion is only 3 layers deep and all code changes are made in one fix cycle. After this, as a progressive company, MegaCorp realizes that it may not estimate the workload for the system correctly and would like to know how it performs under different workload scenarios. Rather than spend days in the performance test laboratory, ClearSight predictions can be used as a virtual test lab to understand how the software performs. Using the virtual test lab results MegaCorp can then select the most interesting scenarios for actual testing in the lab.

Finally, before MegaCorp launches, it is offered a hardware upgrade deal from the hardware supplier. Using the ForeSight predictions the benefits of faster hardware on system performance can be easily quantified, and MegaCorp sees that a hardware upgrade is a smart business decision.

So MegaCorp manages to offer the world a first-rate on-line Pizza ordering service on time with no performance problems!

Summary

This paper has discussed the devastating effects that multiple fix-test cycles can have on a project not only in terms of development cost. Opportunity cost due to missed opportunities and delays can be even higher.

PredictorV not only solves the problems of multiple fix test cycle but also reduces the performance testing needed and provides confidence on how the system will scale on different hardware platforms.

For any project manager embarking on a large scale IT project PredictorV is a must have.

© Crovan 2004