An Aspect Oriented Performance Analysis Environment

Jonathan Davies, Cambridge University,
jjd27@hermes.cam.ac.uk
Nick Huismans, Imperial College,
nick@huismans.com
Rory Slaney, Edinburgh University,
rss@dcs.ed.ac.uk
Sian Whiting, Imperial College,
sian_whiting@yahoo.co.uk

Matthew Webster, matthew_webster@uk.ibm.com
Robert Berry, brobert@uk.ibm.com
IBM Corporation, Hursley, UK

Abstract

Performance analysis is motivated as an ideal domain for benefiting from the application of Aspect Oriented (AO) technology. The experience of a ten week project to apply AO to the performance analysis domain is described. We show how all phases of a performance analysts' activities – initial profiling, problem identification, problem analysis and solution exploration – were candidates for AO technology assistance – some being addressed with more success than others. A Profiling Workbench is described that leverages the capabilities of AspectJ, and delivers unique capabilities into the hands of developers exploring caching opportunities.

1. Performance and the Software Development Process

Performance measurement, analysis and improvement are central activities in the software development lifecycle. Ideally, performance considerations play an early role, e.g., at design time, as recommended in performance-oriented design methodologies [6]. But most often, they factor into the later stages of the development process. This is unfortunate, since design problems detected late in the cycle are more expensive to fix - and may not even be fixed at all because the risked or real schedule impacts are considered as too severe. A very important improvement to this situation now exists thanks to the development and adoption of development environments such as JProbe[4], Eclipse[5] and also VTune[1]. These development environments bring performance awareness and analysis technology closer to the developer by making it easier to explore the performance of code as it is being written. They do so by integrating analysis tools into the development workbench. In this paper we describe our experiences in using Aspect Oriented Software Development (AOSD) techniques in helping to further narrow this gap.

1.1 Performance Profiling

When creating an application, a developer's focus is on program functionality. The program is designed to work - specifically to pass its functional verification testing. Considerations for reliability, serviceability and performance may factor into the design, but unfortunately are not often first order considerations. In one common development model, the functional code is ultimately made available to a performance measurement team which must then go through the following steps to speed up the application:

1. Establish Performance Objectives for the application (CPU time, throughput, response time)
2. Identify workloads to test these objectives
3. Test if the application performance is within the objectives
4. If not, profile the application with performance tools to determine reasons for missing the objective
5. Modify the application to bring the application within performance objectives (Repeat 3-5)

Steps 1 and 2 are essential and can be quite difficult, but for this paper we assume them as given and do not address them further. Step 3 requires the ability to exercise the application in a tractable manner. Fortunately, application development environments such as those mentioned earlier typically provide this capability. Step 4 - analyzing the application - requires the developer to have an awareness of, and a level of facility with, complex performance profiling tools. Step 5 is certainly within the developer's grasp - but pinpointed areas of focus are needed, and this is where Step 4 is so important.

Putting performance profiling in developers' hands is certainly facilitated by existing application development environments - e.g., with hot method profiling, and call-flow profiling capabilities typified by tools such as tprof [3], gprof [2], Vtune [1], and JProbe [4]. Most of these profiling tools concentrate on identifying performance problems relating to program control flow; i.e., they focus on finding hotspots and areas of calling congestion. While these are very important, they are not the only source of optimisation opportunity.

Indeed, we find increasingly with object oriented systems that the flow of data can be just as important. A common type of performance improvement involves the introduction of caching logic into an application, or middleware library. Justification and motivation for the complexity of such caching logic is generally obtained through the introduction of specialised instrumentation. When the application is run the instrumentation can be used to confirm particular argument and return value combinations that support a caching solution. For example, suppose method **X(a)** were called 100,000 times, and for 80,000 of those invocations the input argument **a** took the same value, with the return value also being identical. This would be strongly suggestive of a caching opportunity.

However, in practice, identifying these opportunities can be very expensive. We chose to bring some assistance with this general class of problems through the development of a profiling tool based on Aspect Oriented technology. The promise of employing these technologies for non-invasive, flexible and adaptive instrumentation and subsequent behaviour modification suggested a compelling synergy well worth exploration.

1.2 Related Work

Many existing profilers indicated above focus on CPU time and program flow. Tools such as Vtune also enable analysts to profile on the basis of other hardware activity, e.g., L1 cache misses. However, we knew of no tool that allows applications to be profiled on the basis of data flow – a key requirement for caching opportunity detection. We believed this to be important because as analysts we spend a great deal of time in application and middleware software improving this aspect of performance - yet we lack general tooling to help in this important area.

We sought to develop a technique and environment to selectively gather information on certain methods, argument values, return values, to conduct correlation analysis between these, and to couple that information with timing information. While some of this information could certainly be gathered manually, e.g., Java^{TM} debuggers based on JVMDI are able to track arguments and return values –this is typically a single-step debugging operation. Manually inserted instrumentation (e.g., using **System.out.println** statements, or through the use of a logging or other api) is also possible. Alternately, specialised instrumentation can be developed based on byte-code modification techniques – indeed this is where we began – but the flexibility of AOSD techniques quickly suggested a different approach.

AOSD technology, and in particular for our project, AspectJ [7], made this kind of instrumentation feasible to explore, and provided a basis for enhanced analysis and application performance improvement. Further, and perhaps of more significance, when coupled with Eclipse the result puts technology to identify, investigate, and prototype performance improvements (Step 5 above) within reach of the developer.

In section 2 we discuss goals for the profiler, with section 3 introducing an overview of the design. This is followed by a detailed description of the implementation. Our experiences with using AOSD techniques and AspectJ in particular, for this effort are discussed in Section 5.

2. Goals and Approach

2.1 General Project

Our project had several ambitious aims. We wanted first to explore the value of AOSD technology to a real problem domain; we chose performance analysis. We also deliberately chose to do this with a team initially lacking familiarity with AOSD. With this approach we expected to gain experience with the adoption of this relatively new technology.

2.2 Profiler Focus

We also had specific goals for the profiler. Performance measurement/problem identification, analysis and improvement were selected as the domain of application. The focus was further narrowed to a class of performance problems characterised by solutions involving caching. We did this in part because caching is an area of current interest in our efforts to improve the performance of our software. Another connection to AOSD is the very natural view of caching as distinct concern [9] - or indeed, more generally, of performance improvement as a distinct concern. We felt this created a particularly compelling reason for selecting this combination.

2.2.1 Phases of Performance

For the broader experience of AO technology we sought to apply AO techniques to all phases of performance – measurement, problem (or opportunity) detection, analysis (in this case, of caching opportunities), and exploration of actual performance improvement. We decided to use AspectJ, although similar techniques to the ones described here could also be used with other AO tools such as HyperJ [8] and HyperProbe[11].

We saw value in developing AspectJ aspects to extract information that would otherwise need handwritten code to be explicitly inserted by the programmer. We wanted to be able to identify interesting methods in a non-invasive manner, so the programmer didn't have to understand the profiling process. This required everything to be done within a graphical environment, a Profiling Workbench. By having an easy to use interface it would also allow ordinary developers as well as performance experts to profile code, and apply performance improvements.

For measurement and performance problem identification we create instrumentation aspects that determine which methods have a high invocation count or high execution time within an application. In some respects this is the domain of more traditional profilers. However, unlike other profilers, we were motivated to use Aspect Oriented Programming (AOP) in this phase of performance because we sought the benefits of a tight integration with the other phases of the performance process (analysis and then improvement). Another critical dimension of measurement, also addressed through the use of generated aspects, is in effectively limiting the quantity of data collected.

Problem analysis was the next area for AOSD technology application in the Profiler Workbench. In this case, aspects were created to capture argument types and values, as well as return types and values, for selected methods.

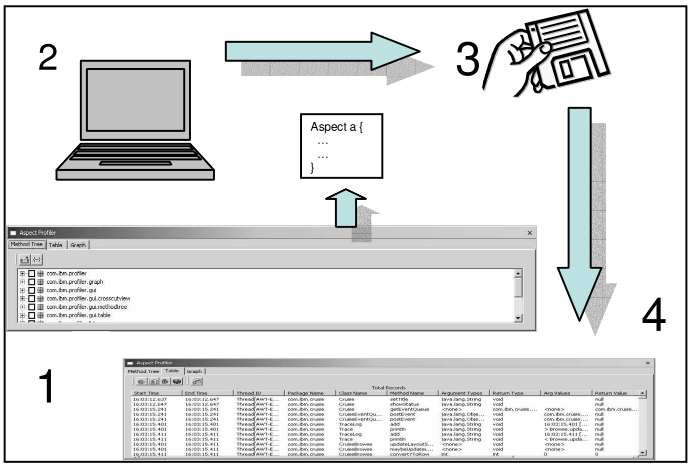
A further aim of the profiler included demonstrating how AO techniques could be used to actually *solve* the performance problems experienced by a particular class or method. Whether this involves automatic caching or pooling, a simple strategy should be able to be put in place within the Profiling workbench. This may not produce the most efficient solution but should be a good indicator as to whether it is worth implementing a more sophisticated or specialised caching technique.

2.2.2 The Developer Experience

Another important goal was creating a solution that would enhance the ability of *developers* to explore and analyse performance problems. This goal is challenging for two reasons. First, as introduced earlier, developers are not generally familiar with the discipline of performance analysis and the use of its tools. Being able to analyse profiler data within the user's development environment was important. This would enable the code to be easily accessible when an interesting method was found. We needed to be able to show the programmer what was happening in the code both for each individual method execution and on a method by method basis.

A second challenge for developer acceptance came from the desire to actually hide AOSD from the developer, the end user of the profiler. While we planned to exploit AO techniques to facilitate problem detection, analysis and improvement, we needed to hide the details of this from the user.

3. The Profiling Workbench



```
com.ibm.profiler
com.ibm.profiler.graph
com.ibm.profiler.gui
com.ibm.profiler.gui.crosscutview
com.ibm.profiler.gui.methodtree
com.ibm.profiler.gui.table
```

Total Records

Start Time	End Time	Thread ID	Package Name	Class Name	Method Name	Argument Types	Return Type	Arg Values	Return Value
16:03:12.637	16:03:12.647	Thread[AWT-E...	com.ibm.cruise	Cruise	setTitle	java.lang.String	void		null
16:03:12.647	16:03:12.647	Thread[AWT-E...	com.ibm.cruise	Cruise	showStatus	java.lang.String	void		null
16:03:15.241	16:03:15.241	Thread[AWT-E...	com.ibm.cruise	Cruise	getEventQueue	<none>	com.ibm.cruise....	<none>	com.ibm.cruise...
16:03:15.241	16:03:15.241	Thread[AWT-E...	com.ibm.cruise	CruiseEventQu...	postEvent	java.lang.Obje...	void	com.ibm.cruise....	null
16:03:15.241	16:03:15.241	Thread[AWT-E...	com.ibm.cruise	CruiseEventQu...	postEvent	java.lang.Obje...	void	com.ibm.cruise....	null
16:03:15.401	16:03:15.401	Thread[AWT-E...	com.ibm.cruise	TraceLog	add	java.lang.String	void	16:03:15.401 [...	null
16:03:15.401	16:03:15.401	Thread[AWT-E...	com.ibm.cruise	Trace	println	java.lang.String	void	> Browse.upda...	null
16:03:15.411	16:03:15.411	Thread[AWT-E...	com.ibm.cruise	TraceLog	add	java.lang.String	void	16:03:15.411 [...	null
16:03:15.411	16:03:15.411	Thread[AWT-E...	com.ibm.cruise	Trace	println	java.lang.String	void	< Browse.upda...	null
16:03:15.401	16:03:15.411	Thread[AWT-E...	com.ibm.cruise	CruiseBrowse	updateLayoutS...	<none>	void	<none>	null
16:03:15.401	16:03:15.411	Thread[AWT-E...	com.ibm.cruise	CruiseBrowse	maybeUpdateL...	<none>	void	<none>	null
16:03:15.411	16:03:15.411	Thread[AWT-E...	com.ibm.cruise	CruiseBrowse	convertYToRow	int	int	0	0

The above figure depicts at a very high level the use of the Profiling Workbench. It shows key stages of the developer's workflow while engaging in a caching exploration. At (1) the developer is presented with a list of candidate methods. The contents of this list can be modified by varying the selection criteria. When the developer indicates a desire to profile (2), the workbench will *generate* the instrumentation aspects needed to profile the selected methods. These aspects are applied by the framework (the AspectJ compiler is an integral part of the Workbench). Initial profiling results are obtained (3), and presented to the developer (at 4). These results indicate areas of potential improvement based on invocation frequency or time-in-method.

This process can then be repeated, starting again at (1) but generating new aspects aimed at collecting method argument types and values (and return types and values). Again, the aspects are generated, applied to the application codebase and new data is collected. This data is then available for graphing and correlation analysis.

In correlation analysis, the Workbench presents the developer with graphical feedback on the behaviour of the hot methods. Here it is possible to (manually) identify caching opportunities by observing argument values and return values for methods.

When a potential caching opportunity is identified, the developer can then select from a set of caching aspects (instead of profiling aspects) to apply to a particular method. A caching aspect, customised to that target method, is automatically generated, and the application can then be rerun. The performance analysis loop is completely self-contained – from identification, to analysis, and then to performance improvement prototyping and measurement.

4. Profiler Implementation and Usage Detail

4.1 Introduction

The Profiling Workbench is written in Java as a plugin for Eclipse. It depends on three other Eclipse plugins: AspectJ Development Toolkit (AJDT) which provides the AO types for Eclipse, AspectJ Development Environment (AJDE) which provides the compiler and is required by AJDT, and Draw2D which is used for drawing graphs. The profiler is also dependent on the IBM High Resolution Time Stamp Facility [12] in order to get high resolution event timing information on the Windows 32 platform. The plugin consists of one view containing three tabs: Profiling Rules, Table and Graphs.

4.2 Profiling Rules:

The Profiling Rules View is used to select methods to profile, parameters to log, sampling techniques and handlers to use. Methods can be selected for profiling and then the Workbench produces an appropriate AspectJ pointcut.

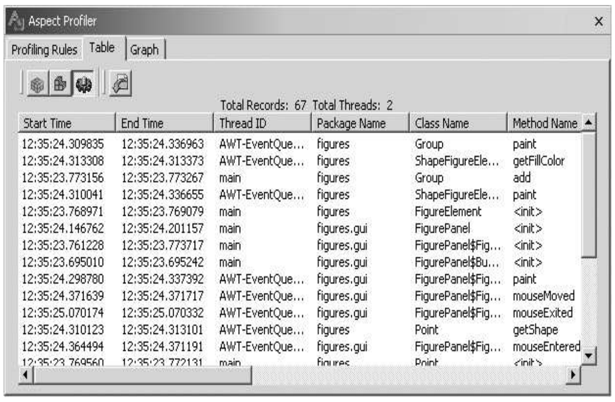
A “Logging Parameters Wizard” allows the user to select method parameters to log. This is an orthogonal mechanism because logging argument and return values is likely to cause distortions to timing values. The advised practice is to run the profiler twice with the same test case. On the first run, timing information should be gathered and on the second, values and types. The results of the first run allow for the identification of a small set of methods of interest. On the second profiler run, logging all parameters for this more restricted set of methods provides a reasonably accurate and general overview.

The logging parameters wizard contains a handlers page. A handler facility is provided so that the user can extract meaningful values from complex data types. Several pre-defined handlers are provided with the option and documentation for the user to write their own. The logging parameters wizard also provides an option to profile library packages. Execution time is calculated by taking away the time within method values for all sub calls from the time within method value for the method. Profiling library packages means that execution time values are more accurate for methods where library calls are made.

A “Launch Filtration Wizard” enables the user to apply sampling techniques. If the project is large or run for a considerable time the user will probably want to employ some sampling to avoid using a lot of disk space. A “Generate Aspect” step automatically generates an aspect to perform the profiling with all the options the user has selected. An extra package is added to the project containing an aspect called Profiling.java. This contains the pointcuts created with the wizards and the necessary logging advice.

4.3 Table:

In order to gather profiling data the project is built and run as normal. The results are presented through the Table View. This view shows method entry/exit statistics, execution frequencies, argument types and values. Data can be grouped in a variety of ways – e.g., by class, by method:

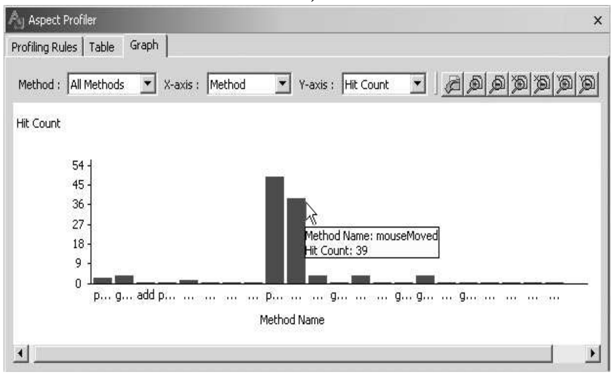


Total Records: 67 Total Threads: 2

Start Time	End Time	Thread ID	Package Name	Class Name	Method Name
12:35:24.309835	12:35:24.336963	AWT-EventQue...	figures	Group	paint
12:35:24.313308	12:35:24.313373	AWT-EventQue...	figures	ShapeFigureEle...	getFillColor
12:35:23.773156	12:35:23.773267	main	figures	Group	add
12:35:24.310041	12:35:24.336655	AWT-EventQue...	figures	ShapeFigureEle...	paint
12:35:23.768971	12:35:23.769079	main	figures	FigureElement	<init>
12:35:24.146762	12:35:24.201157	main	figures.gui	FigurePanel	<init>
12:35:23.761228	12:35:23.773717	main	figures.gui	FigurePanel$Fig...	<init>
12:35:23.695010	12:35:23.695242	main	figures.gui	FigurePanel$Bu...	<init>
12:35:24.298780	12:35:24.337392	AWT-EventQue...	figures.gui	FigurePanel$Fig...	paint
12:35:24.371639	12:35:24.371717	AWT-EventQue...	figures.gui	FigurePanel$Fig...	mouseMoved
12:35:25.070174	12:35:25.070332	AWT-EventQue...	figures.gui	FigurePanel$Fig...	mouseExited
12:35:24.310123	12:35:24.313101	AWT-EventQue...	figures	Point	getShape
12:35:24.364494	12:35:24.371191	AWT-EventQue...	figures.gui	FigurePanel$Fig...	mouseEntered
12:35:23.769560	12:35:23.772131	main	figures	Point	<init>

4.4 Graph:

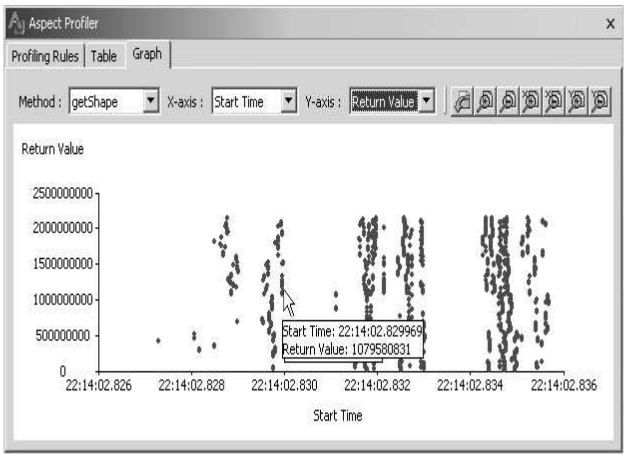
The Graph View can be used for the initial selection of hot methods, as shown below.


Method : All Methods X-axis : Method Y-axis : Hit Count

Method Name: mouseMoved
Hit Count: 39

Aspect Profiler
Profiling Rules
Table
Graph
Hit Count
54
45
36
27
18
9
0
p... g... add p... ... ... p... ... ... g... ... ... g... g... ... g... ... ...
Method Name

Detailed argument type and value data can also be graphed for correlative analysis purposes in the Graph View, as illustrated in the next figure. This figure illustrates a graph of execution (wall clock) time against return values. It appears to suggest that return values vary more or less uniformly across the range from [0..2,000,000,000], and so a caching solution in this case might not be advantageous.


Method : getShape X-axis : Start Time Y-axis : Return Value

Start Time: 22:14:02.829969
Return Value: 1079580831

4.5 Caching:

A basic general caching solution was implemented in the Workbench by adding a set of caching aspects. Application methods observed with appropriate argument/return value characteristics can be easily explored for the benefit that a caching might afford. One such generic caching aspect uses a hashtable with keys based on arguments and return values stored. If argument values for the current method have been used before, the value from the hashtable is returned. Otherwise the value is calculated, stored in the hashtable and returned. For an expensive method this should demonstrate the value of adding a permanent caching solution.

5. Exploitation of AOSD & AspectJ

The aspect-based profiling tool was programmed in pure Java exploiting several AOSD and AspectJ features and methodologies. This section details these exploitations; explains why they were deemed useful; where they were used in the profiler; and what the alternatives would have been, had the profiler been developed using traditional programming techniques.

While many of the following observations apply generally to the use of AspectJ, they are derived directly from our experience in using this technology to develop and experiment with the profiler.

5.1 Weaving of Cross-cutting Concerns

The fundamental feature of AOSD and AspectJ, is the ability to modularise cross-cutting concerns and to weave them into a program before or at run-time. AspectJ, in particular, allows the possibility for code in aspects' advice to be weaved into the target program achieving the same effect as if it were in-line code, despite it being stored in a separate modular unit. This feature may be exploited to insert one piece of code at an arbitrarily large number of positions within the source code of a program. Furthermore, with AspectJ, it is particularly easy to uniquely identify, and define behaviour to occur at, these positions within the execution of the program. Moreover, these points may be static (dependent of position in the source code) or dynamic (dependent on the control flow of the program).

5.1.1 Current Use

The aspect-based profiling tool gathers information regarding the performance of a Java program by inserting an aspect. This advises certain methods guided by a pointcut, which is custom-generated, based on the user's selections. Please refer to section 4 "Profiler Implementation and Usage" for a more detailed explanation of this process. The advice which is applied to these chosen methods provides simple logging behaviour, to gather the information regarding the details of the circumstances of execution of the method. This is an elementary example of a cross-cutting concern, logging, which can be woven into arbitrary methods across the entire program.

5.1.2 Alternatives

An alternative to having a separate module containing the cross-cutting logging code, which the profiler caused to be woven into the target program, would be to actually insert the logging code into each source file containing methods to be logged. In other words, the alternative would be to manually perform the same job that AspectJ does itself. Clearly, emulating AspectJ's behaviour in this way would be wasteful.

A second alternative would be to modify the JVM to explicitly extract and log the desired performance information whilst the program is running. While feasible, this is not an easy step, and it would lack the flexibility of our current approach.

5.2 Sufficiency of Expressiveness of Pointcut Language

AspectJ uses pointcuts to pick out well-defined points in a program. The language which is used to define pointcuts is a powerful combination of Boolean operators and potent pointcut designators, including regular expressions for succinct formation of complex pointcuts.

5.2.1 Current Use

Pointcuts are used in an aspect generated by the profiler, which is then inserted into the target program. These pointcuts pick out the methods for which performance information should be gathered. The "execution" pointcut designator, in partnership with before and after advice, is used to catch the start of the execution of these particular methods. The advice records information such as the name of the method; the name of the thread executing the method; the time duration spent within the method; and the argument and return values – such information can be used to build up a picture of the performance of this method.

We found the pointcut language sufficiently expressive to pick out these methods with simplicity and succinctness. The efficiency of pointcut expression was manifested primarily in the way in which wildcards can be exploited. For example, if the user wishes to log the performance information for every method within a certain class, then it is only necessary to generate an aspect containing the pointcut designator: execution(* className.*(..)).

In addition, the pointcut language is also powerful in terms of identification of particular sets of methods. For example, it is possible to pick out all methods with a particular return type, by using execution(returnType *..*(..)); or all methods with an argument of a particular type, by using execution(* *..*(.., argumentType, ..)).

Furthermore, the pointcut language is exploited by using the "call" pointcut designator to gather information about invocations of methods for which the user does not have access to the source code.

5.2.2 Alternatives

If wildcards were not available, then the alternative to a pointcut designator such as execution(* className.*(..)) would be a large disjunction of designators for each single method within the class. Clearly, this is inefficient and the resulting pointcut would prove difficult to read and is insusceptible to manual alterations.

Whilst it is reasonably easy to list all the methods within a particular class, it is a more complex task to list all the methods with a particular return type, or with an argument of a particular type, across the scope of the entire program, or perhaps within an arbitrary limited scope. If the possibility did not exist to use pointcut designators such as execution(returnType *..*(..)) or execution(* *..*(.., argumentType, ..)), then some search would need to be made across the scope of the entire source of the program to find the methods with the desired argument or return type. Clearly, this is a laborious process, which would require a great deal of manual effort, or a processor-intensive task to be executed.

5.3 Power of `thisJoinPoint`

AspectJ provides an object, accessible within the scope of the advice, called thisJoinPoint. The object contains both static and dynamic information about the joinpoint which matched the pointcut causing the advice to be executed. This information is invaluable for gathering data relevant to inspection of performance of methods. For example, thisJoinPoint holds the name of the executing method, and the argument values which were passed to the method.

5.3.1 Current Use

To obtain information relevant to the performance of methods within a program, the aspect generated by the profiler contains one piece of before and after advice, each to be executed for all methods picked out by the pointcuts. Because this advice had to be general and be used by arbitrary methods, thisJoinPoint was invaluable for extracting the values of arguments, in particular.

5.3.2 Alternatives

The alternative to having one general piece of before and after advice, and using the magic of thisJoinPoint, would be to have specific advice for each method and use the “args” pointcut designator to extract argument values and “returning” keyword to obtain the return value. This would require knowledge about the signature of each method, to determine the layout of the args pointcut designator’s arguments, in each specific pointcut. Moreover, there would have to be an individual pointcut and an accompanying piece of advice, for each method to be profiled. Clearly, in a program of any reasonable size, this would lead to the generation of a huge aspect, which is difficult for a human to read and understand; and to amend or modify.

Traditional profilers do not extract the values of arguments and the results of methods. Using non-aspect-based profiling techniques, the user would be expected to insert logging code into his program (perhaps using System.out.println, or similar) to gather the values of arguments and results of methods the user is interested in. This is clearly not an ideal situation, because it involves extra effort on the part of the user. Also, this enforces the necessity that the program must be run twice – first to identify the methods susceptible to profiling; and second, once the logging code has been manually inserted, to gather the values of arguments and results of the methods under inspection. If the program is processor-, memory-, or time-intensive, then running the program more than once is undesirable. Using the aspect-based profiler, there is the possibility for values of arguments and results of methods to be gathered directly.

5.4 Power of Around Advice

AspectJ allows the implementation of a particular method to be replaced through the use of “around” advice. In addition the use of “proceed” allows new logic to be introduced before and after execution of the original method if required.

5.4.1 Current Use

Once a method has been identified as performing below its expected level of performance (in terms of execution time), the decision must be made regarding how to improve the performance of the method. Typical solutions may be to implement a caching or pooling policy. Caching aims to save time by avoiding repeating expensive sections of code; pooling aims to save time by avoiding expensive re-creation of objects when existing objects may be re-used.

The aspect-based profiler encompasses not only the identification of performance problems, but also aims to aid the user to address these problems, by implementing a simple, out-of-the-box caching algorithm, for expensive methods. This caching is achieved by inserting an aspect into the target program which contains a pointcut for a specific method and accompanying around advice. This advice implements simple caching by checking to see if the method's argument values have been used before – and, if so, returning the same return value; if not, calling proceed and storing the obtained return value in the cache. (Notably, such caching techniques will only be meaningful for, and will only improve the performance of, methods within a restricted set.)

Such simple caching is not intended to be a complete solution, but merely enables the user to identify quickly and easily whether some sort of caching technique may be useful for improving the performance of the method in question. If a performance gain is experienced, it is expected that the user would then modify the caching aspect to tailor its behaviour to suit the particular method. If no performance gain is experienced, then it is a simple matter to remove the caching aspect, to return to the original, uncached implementation of the method.

5.4.2 Alternatives

The alternative to using a separate caching aspect would be to modify the source code of the program. This has the obvious disadvantage that it is difficult to switch on and switch off to rapidly determine if a performance gain is realised. It is also considerably more error prone. Furthermore, the generic nature of the aspect means that it can be re-produced to suit any arbitrary method, with ease

6. Conclusions

The ability to identify, analyse and potentially solve performance problems within an IDE (Integrated Development Environment) is compelling. Both the flexibility (through the use of wildcards) and the control (using pattern and type matching) available with the AspectJ pointcut language make it possible to extract detailed information from an executing program without resorting to either handwritten instrumentation or JVM modifications. Furthermore the use of execution pointcut before/after advice allows the collection of data for the target application only rather than the whole system as is common in other profiling techniques. The use of call pointcuts before/after advice can also eliminate from results the cost of invoking library classes. Finally the use of an aspect to apply performance enhancements demonstrates a classic example of modularizing a cross-cutting concern, in this case caching or pooling.

As the project developed some of the limitations of AspectJ became apparent. The current version of the compiler rebuilds any source code that is likely to be affected by an aspect, something that can be very time consuming for large applications. Future versions are expected to use incremental compilation which will greatly improve the usability of the profiler. In addition the extra pathlength (executed lines of code) associated with the profiling aspect can perturb results, especially for small methods, making "hot" method identification difficult. Whilst the use of sampling techniques can reduce data volumes, pathlength is still affected as the checks are made in the aspect not at the joinpoint.

The Aspect Oriented Profiler has been adopted by the eBusiness Integration Technologies performance team for analysing existing and future IBM products. Other groups at the IBM Hursley Laboratory have also expressed an interest is using the profiler.

The work so far has concentrated on only one area of performance analysis: pathlength measurement and reduction. However, the analysis of other factors impacting system throughput such as object lifetime, locking and the use of exceptions also lend themselves very well to the use of AOSD. Events such as object creation, monitor contention and exception handling can all be intercepted using the powerful pointcut definitions of AspectJ. Future enhancements could allow the profiler to address a broader range of performance concerns.

7. Extreme Blue and Project Roles

This project was conducted as an IBM Extreme Blue initiative. It took place over a 10 week period in the summer of 2002 at the IBM Laboratory, Hursley, UK. The mentors for the project were Matthew Webster and Robert Berry. Robert Berry has 20 years of experience with performance measurement and suggested the application of AOSD techniques to this field. Matthew conceived the idea of "whole lifecyle of performance analysis" within the Eclipse environment. The development of the plugin was split between the students. Jonathan wrote the Aspect generator and Rory developed the various sampling mechanisms. Nick created the tables for data selection and worked with Sian to design the graphing tools for data analysis. The project was an intensively collaborative effort benefiting from a tremendous level of teamwork.

Acknowledgements
We would like to thank our management sponsor, Dr. Mark Thomas for his strong support and direction. We gratefully acknowledge the assistance and encouragement of the IBM European Extreme Blue program director, Stuart Fawkes.

8.0 References

[1] VTune. Intel Corporation. http://www.intel.com/software/products/vtune/index.htm
[2] gprof. http://www.gnu.org/manual/gprof-2.9.1/gprof.html
[3] Rudy Chukran, "Accelerating AIX: Performance Tuning for Programmers and System Administrators", Addison-Wesley, Paperback, Published March 1998
[4] JProbe. Sitraka Corporation. http://www.sitraka.com/software/jprobe/
[5] Eclipse. http://www.eclipse.org
[6] Connie U. Smith, "General Principles for Performance Oriented Design", pp138-144, Proceedings of the Computer Measurement Group, 1987, Orlando Florida.
[7] G. Kiczales, E. Hilsdale, J. Hugunin, Mik Kersten, J. Palm, W. Griswold, "An Overview of AspectJ". In Proceedings ECOOP'01, Springer-Verlag, June 2001.
[8] H. Ossher, Peri Tarr, "Multi-dimensional Separation of Concerns and the Hyperspace Approach", In Proceedings of the Symposium on software Architectures and Component technology: The State of the Art in Software Development, Kluwer, 2001.
[9] Stanley M. Sutton, Jr., Isabella Rouvellou, "Concerns in the Design of a Software Cache", Proceedings Advanced Separation of Concerns in Object-Oriented Systems, Workshop at OOPSLA 2000, Minneapolis, Minnesota, October, 2000.
[10] SPECjbb2000, Java Business Benchmark, Standard Performance Evaluation Corporation, http://www.spec.org.
[11] D. Kimelman et al, HyperProbe - An Aspect-Oriented Instrumentation Tool for Troubleshooting Large-Scale Production Systems, Demonstration at AOSD 2002, Enschede, 2002.
[12] High Resolution Time Stamp Facility, alphaWorks, IBM Corporation, http://www.alphaworks.ibm.com/tech/ibmts