MEProf: Modular Extensible Profiling for Eclipse

Marc Hull, Olav Beckmann, Paul H. J. Kelly

Department of Computing, Imperial College London
180 Queen's Gate, London SW7 2AZ, United Kingdom
Email {mfh02,ob3,phjk}@doc.ic.ac.uk

Abstract

This paper presents a profiling plug-in for IBM's Eclipse development environment. Our approach characterises profiling as an interactive exploration of a large virtual database of information about the execution of a program. We define a high-level, graphical environment for programming a profiling pipeline, which specifies how profiling data is collected, filtered and visualized, and allows the user to write custom Java code that can intercept and manipulate the profiling data passed between stages. We use Aspect-Oriented Programming (AOP) to program the collection of profiling information, allowing this process to be tailored to particular program contexts and domain-specific program characteristics.

1 Introduction

The goal of profiling is to understand the performance characteristics of a program and to attribute such characteristics to particular source code features. Programmers' profiling requirements are highly varied. In some cases, a detailed understanding of the performance of key routines, in terms of hardware performance counters at bytecode or assembler level is required. In other cases, a lightweight, high-level profile of a running application may be required. Furthermore, for many applications, the actual information which the programmer requires is domain-specific, or may be information that is derived from lower-level profiling data. This variety of profiling requirements has resulted in many different tools, each usually focussing on a particular, restricted aspect of a program's performance. IBM's Eclipse development environment allows such tools to be supplied as separate plug-ins. Taken on its own, the weakness of this approach is that it may be hard for programmers to collate, match and compare data that has been obtained from different plug-in tools.

MEProf introduces an additional framework on top of Eclipse's plug-in based architecture that contains an extensible suite of profiling tools and allows users to tailor the set of tools they deploy in accordance with their requirements. MEProf has a modular software architecture, which separates common tasks performed by profiling tools, such as data collection and logging, in order to reuse code and allow modules to be focussed on data processing or visualisation tasks. MEProf also allows data to be shared across multiple profiling tools in order to reduce unnecessary delays caused by gathering repeated information.

1.1 Contributions of this Paper

- We characterise profiling as an online exploration of a large virtual database of information about the execution of a program.
- We define a *profiling pipeline* as a specification of how profiling data is collected, processed and visualised.
- We propose a high-level graphical interface for programming profiling pipelines. This has been designed to support demand-driven

evaluation – ensuring that only data that is actually required is recorded at runtime.
- We demonstrate our approach with a prototype implementation of MEProf.

2 The Profiling Pipeline

We model the profiling process as a data-flow graph, or *profiling pipeline*. We refer to nodes in the profiling pipeline as *modules*. Modules are operations that either collect data (input nodes), process data or perform data output, including visualisation. Edges denote data flow.

2.1 Programming Profiling Pipelines

MEProf provides users with a high-level, graphical interface for specifying profiling pipelines by combining profiling modules and controlling the flow of data along each edge of the pipeline. We have classified these modules into six distinct types based on the task they perform.

Trigger Modules capture specific events in a program's execution, such as method invocation, garbage collection, increases in heap size, or regular time intervals.

Collector Modules gather information about a running program, such as the name of the current method, the number of bytecode instructions executed since the last collection, or the contents of a certain variable in the program.

A trigger combined with one or more collectors constitutes an input to the profiling pipeline. As we outline in more detail in Section 2.3, triggers and collectors together specify a profiling aspect. Roughly, triggers can be viewed as corresponding to pointcuts and collectors as corresponding to the advice that is deployed. Each combination of a trigger module with one or more collector modules supplies a stream of records of profiling data into the profiling pipeline.

Aggregator Modules process profiling data, for example by calculating statistical summaries, such as the number of times an event has occurred.

Adapter Modules convert Java objects into a form in which they can be visualised, such as converting a stack trace into an array of strings.

Filter Modules allow the user to select only a subset of the collected data to be passed on to later pipeline stages. This allows, for example, the data displayed to be reduced to a particular region of interest within a program.

Visualisation Modules display data on screen in a particular format, such as bar charts, trees, or plain-text listings. Visualisation modules constitute the outputs of profiling pipelines.

Although the order of components in a profiling pipeline typically follows the order of the above list, the user is not restricted to this sequence. MEProf allows any number of each type of module to be added to the pipeline, subject to triggers and collectors being combined to form inputs and visualisation modules being used as outputs. In addition, as we discuss in the next section, we control carefully how data-flow between modules can be programmed in order to ensure that the results of the pipeline as a whole are coherent.

2.2 Type-Based Module Composition

The above model could lead to the following pitfalls, which have to be avoided:

- Type errors: we have to ensure that users cannot connect modules that expect a stream of numbers to modules that produce a stream of strings or Java objects, etc.
- Coherence: when an aggregator module is used, for example, to perform a join operation on two streams of records, we have to ensure that these records share a field over which the join operation can be performed.

We avoid these potential problems by controlling the data-flow through the profiling pipeline via a connection-based system, similar to Eclipse's own extension point system for passing information between plug-ins. Each module defines a set of input and output connectors, where each definition includes a description of possible types communicated via that connector, and of whether or not the connection is optional.

2.3 Basic Example

We illustrate the specification of a profiling pipeline with a simple example in Figure 1, which counts the number of times each method in a Java applet is called and plots the result as a bar chart.

MEProf supplies pre-written modules which can be combined to perform this task:

- `MethodEntry`: This trigger intercepts all method invocations. We show the AspectJ code corresponding to `MethodEntry`.
- `MethodName`: This collector logs the signature of the current executing method.

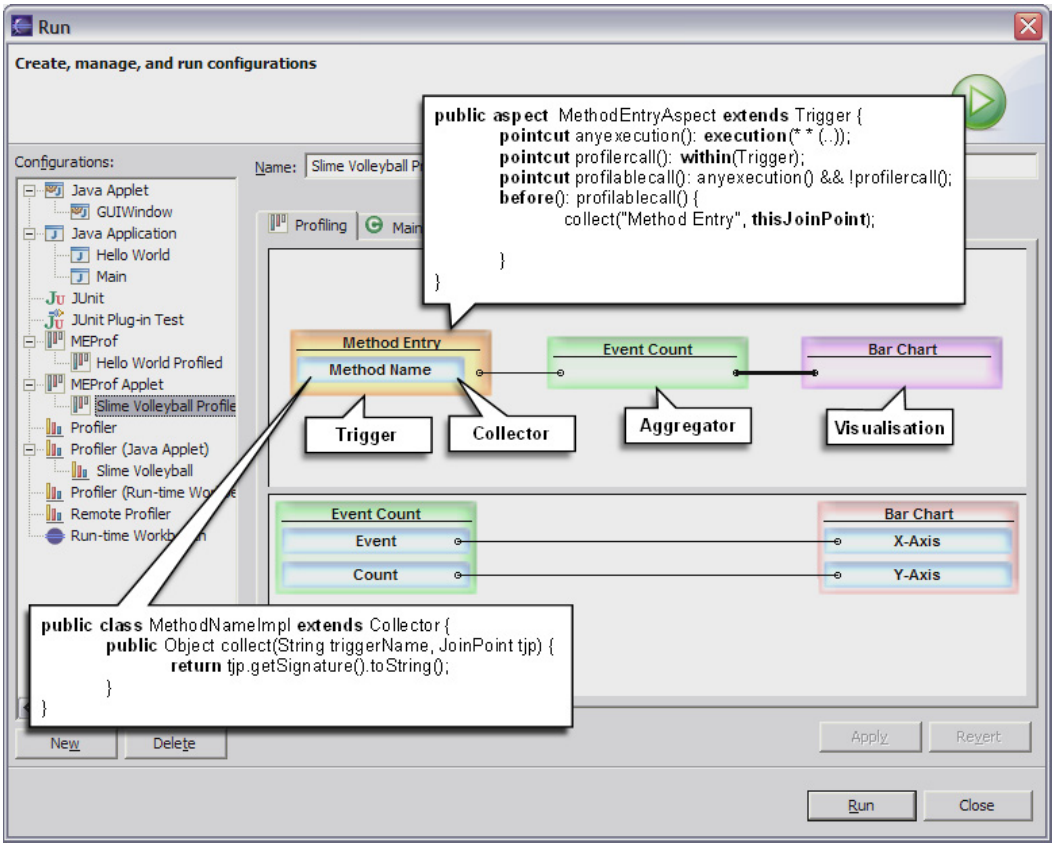

Figure 1. Pipeline for profiling the number of times methods in a Java applet "Slime Volleyball" are called.

- The `EventCount` aggregator counts the number of times each method signature occurs in the stream of events.
- The `BarChart` visualisation module plots the table of methods and counts as a bar chart.

To connect these components, the user simply clicks on the output point of one module, followed by the input point of another. We also illustrate in Figure 1 how the user can configure the data flow between modules by clicking on the module connection and then specifying the mapping of fields between the input and output streams of the connected modules.

The run button in Eclipse then causes MEProf to generate and deploy the instrumentation for this pipeline and start running the program. Once the program terminates, the collected data is sent to the profiler server running in Eclipse, which updates the bar chart view of the results. The bar chart resulting from the profiling pipeline in Figure 1 is shown in Figure 2.

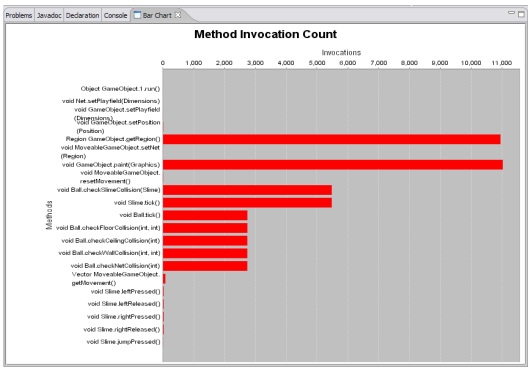

Figure 2. Barchart visualization resulting from the pipeline in Figure 1.

2.4 Optimising Profiling Pipelines

Although the user is given the freedom to define the data that is to be collected and processed, the underlying implementation is hidden by MEProf. This opens an opportunity for reducing the per-

formance overhead suffered as a result of profiling by evaluating the profiling pipeline in a demand-driven manner. If, for instance, method call counts are to be visualised by a bar chart, then MEProf can determine that it is not necessary to log all method call events and that a simple table of aggregate totals is sufficient. On the other hand, in cases where an additional visualisation forces all method calls to be logged, MEProf can identify the optimum point for data logging as being before the `EventCount` aggregator, delaying the call to the aggregator until after the program has terminated. These simple optimisations significantly reduce the memory and performance overhead of profiling.

2.5 Extensible Profiling Architecture

Although the modular system used to construct profiles allows for a large degree of flexibility, users may wish to profile a specific feature of the behaviour of a program that is not covered by the pre-installed modules. To cater for this, MEProf has a plug-in based architecture that allows new profiling modules to be easily written and integrated into the program.

This extensible architecture facilitates domain-specific profiling, where specialised trigger and collector modules capture events in a program that either could not be obtained with existing tools or that would be hard to isolate from the large volume of data produced by existing tools.

2.6 Example: User-Defined Profiling

We demonstrate MEProf's extensible architecture with a second example. The "slime volleyball" applet is a simple game where two players try to knock a ball so that it lands on the other player's half of the screen. We implement a profiling pipeline which logs the velocity of the ball.

The first task is to write a new trigger aspect, "Ball Speed Change", which specifies the event when the ball's movement vector is updated. Next, we need a collector which logs the ball's speed. We show the source code for these in Figure 5. Once written, these files are jarred up along with a description of their connection points and dropped into MEProf's modules directory. The next step is to complete the profiling pipeline. We combine our custom trigger and collector with an "Incrementor". This collector simply outputs an integer that increments on each event, and will allow us to maintain a rough estimate of time for plotting ball speed against time. We show the corresponding profiling pipeline in Figure 3 and the resulting visualisation in Figure 4.

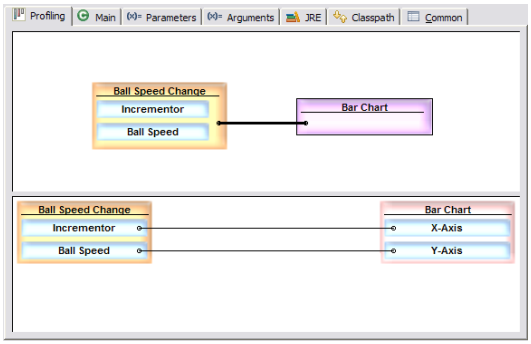

Figure 3. MEProf profiling pipeline for speed of the volleyball.

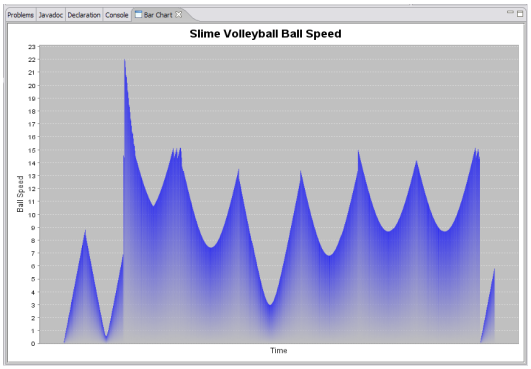

Figure 4. Visualisation of the velocity of the volleyball over time.

3 How it Works

MEProf manages the data flow through the profiling pipeline by maintaining a simple database which contains a separate tables for each module. The declaration of input and output connection points for a module corresponds to the specification of the format of these tables. Each connection point represents an individual column in a table, so when a module has two input points A and B, the data received will be a table with two columns named "A" and "B". Similarly, if there are two output connection points, C and D, then the module's constructor will be passed a table with two columns "C" and "D" which it may write to in order to pass information to other modules.

Each module consists of a JAR containing two Java class files, one of which is a description of the module, including its connection points, the

```
public aspect BallSpeedChangeAspect extends Trigger {
    pointcut vectorchange(Vector v):
            call(* Vector.change*(..)) && target(v);
    pointcut vectorset(Vector v):
            call(* Vector.set*(..)) && target(v);
    pointcut ballvectorchange(Vector v):
            vectorchange(v) && within(slimevolleyball.Ball);
    pointcut ballvectorset(Vector v):
            vectorset(v) && within(slimevolleyball.Ball);

    before(Vector v): ballvectorchange(v) {
          collect("Ball Speed Change", thisJoinPoint);
    }

    before(Vector v): ballvectorset(v) {
        collect("Ball Speed Change", thisJoinPoint);
    }
}

public Object collect( String triggerName,
                       JoinPoint thisJoinPoint) {
    Object target = thisJoinPoint.getTarget();
    if (target instanceof Vector) {
        return new Double(((Vector)target).modulus());
    } else {
        return new Double(0);
    }
}
```

Figure 5. Aspect code and collector method for logging the speed of the volleyball.

other contains the code to be executed when the module is invoked. Both extend classes inside MEProf that handle integration with the rest of the system, leaving programmers to concentrate on code specific to the module they are developing.

When configuring the profiling pipeline, the user specifies data flow in two levels: high-level module connections must be configured at a lower level to specify which data fields from the source will map to which data fields in the destination. This can be seen in Figure 1. MEProf implements this by making a structural copy of the output table from the source module, renaming the columns as specified by the data connection and then passing this table to the target module. Since copying the table is a shallow clone of its column names and pointers to the start of the data, the overhead of this process is very low.

MEProf differs from most other Java profiling tools in that it does not make use of JVMPI to collect data. Instead, trigger modules that capture program events are written and deployed using aspect-oriented programming, allowing data to be captured at any program point that is advisable by the AspectJ syntax. This also allows users of the system to write their own triggers to capture events specific to their programs. To perform a profiling run, MEProf recompiles the program using the AspectJ compiler. This weaves the trigger aspects into the project, which in turn invoke other modules to propagate data through the profiling pipeline.

4 Related Work

The PROSE group at ETH Zuerich have been exploring alternative dynamic aspect weaving technologies [1]. Paradyn [3] uses dynamic instrumentation for performance analysis and implements a callgraph-based bottleneck search. EVolve [2] is an infrastructure for implementing visualisation tools for custom profiling data. STEP [4] is designed to provide a general mechanism for encoding program trace data.

4.1 Acknowledgements

This work was funded by an IBM Eclipse Innovation Grant on “extensible domain-specific performance analysis using dynamic aspect weaving”.

4.2 About the Authors

Marc Hull is a student on the Undergraduate Research Opportunities (UROP) scheme at Imperial College London. Olav Beckmann is a post-doctoral researcher at Imperial College London. Paul H. J. Kelly is a reader in the Department of Computing at Imperial College London and leader of its Software Performance Optimisation Research Group.

References

[1] A. Popovici, G. Alonso, T. Gross. Just-in-time aspects: efficient dynamic weaving for Java. AOSD 2003.

[2] Q. Wang, K. Driesen, L.J. Hendren. EVolve: An Extensible Visualisation Environment. Software Visualisation, OOPSLA 2001.

[3] http://www.cs.wisc.edu/paradyn

[4] R. Brown, K. Driesen, D. Eng, L.J. Hendren, J. Jorgensen, C. Verbrugge, Q. Wang. STEP: A Framework for the Efficient Encoding of General Trace Data. PASTE 2002.