*J: A Tool for Dynamic Analysis of Java Programs[*]

Bruno Dufour, Laurie Hendren and Clark Verbrugge
School of Computer Science
McGill University
Montréal, Québec, CANADA H3A 2A7
[bdufou1,hendren,clump]@cs.mcgill.ca

ABSTRACT

We describe a complete system for gathering, computing and presenting dynamic metrics from Java programs. The system itself was motivated from our real goals in understanding program behaviour as compiler/runtime developers, and so solves a number of practical and difficult problems related to metric gathering and analysis.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics; D.2.11 [**Software Architectures**]: Domain-specific architectures

General Terms

Experimentation, Design, Languages, Measurement, Performance

Keywords

Dynamic Metrics, Software Metrics, Program Analysis, Java, Profiling, Execution Traces

1. INTRODUCTION

This work describes a complete system for gathering trace data from runs of Java programs, calculating dynamic metrics based on that input (offline), and presenting the output in a usable fashion. The system itself was motivated from our real goals in understanding runtime program behaviour as compiler and runtime system developers, and so solves a number of surprisingly difficult problems:

- Dynamic metrics do not yet form a well-defined corpus of calculations. Individual metrics were motivated from ongoing investigations into program behaviours, and so considerable flexibility must be built into the system to allow for new metric calculations to be easily added.
- Existing dynamic data gathering systems did not provide all the data required for the metrics we wanted to compute. For future metrics, as well as comparative features, we needed the ability to calculate metrics from a wide variety of input sources, including Java's built-in profiling interface, instrumented programs and virtual machines, and more. This genericity impacted both trace format and how calculations could be done.
- Trace data from running programs can be quite voluminous, even under compression. Handling such large inputs as well as the performance limitations of some trace generators severely constrains the design of the system.

Section 2 describes the overall pipeline design of the software tool, describing each of its 3 basic components. Related systems are discussed in section 3, and future work and conclusions are given in sections 4 and 5 respectively.

2. DESIGN

Conceptually, *J is a pipeline composed of three major components. The *trace generator* produces a stream of program events from a program execution, which is then fed to the *analyzer*. The analyzer calculates the actual metrics and produces summary information as an XML file which can then be incorporated into a *database* or viewed using different means.

2.1 Trace Generator

Traces in *J are formed of records of runtime events, and use a simple, generic and extensible trace format. This simplifies the task of collecting data from multiple sources. This design also makes it easy to add new metrics because they can be calculated by reprocessing old data, thus avoiding costly trace generation phases.

Currently, our main trace generator uses the built-in Java Virtual Machine Profiling Interface (JVMPI) [1] to dynamically receive events from a JVM. Command-line options can modify the agent's behaviour, and in particular, a specification file can be used to select which events and which of their fields have to be recorded in the trace file, without requiring any modification to the agent. This agent runs on any platform implementing JVMPI.

Unfortunately, while JVMPI is reasonably ubiquitous it has several drawbacks. The most obvious one is the fact that JVMPI's callback mechanism is inherently slow for frequent events, which results in a very significant increase in execution time even for simple benchmark programs. JVMPI also does not guarantee complete trace data—some events may be skipped during VM startup and have to be explicitly requested in order to obtain a precise execution trace; this requires non-trivial internal state in order to track and handle missing events. A further limitation of JVMPI is in the data it provides. Not only are event types fixed (limiting metric possibilities), but even the event data can be insufficient: JVMPI reports instruction executions using code offsets, and so locating the actual opcode of the executed instruction requires classfile parsing.

Complete traces of even small programs can be very large. *J thus supports splitting trace files into more manageable sizes. *J is also designed to make use of a pipe or socket between the agent and the analyzer, eliminating the need to store trace files on disk. Ap-

[*]This work was supported, in part, by NSERC, FCAR and McGill FGSR. Special thanks to Tobias Simon for the web pages and Marc-André Dufour for designing the icons for the metrics.

Copyright is held by the author/owner.
OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.
ACM 1-58113-751-6/03/0010.

propriate compression techniques and alternative trace generators are part of active future work.

2.2 Analyzer

The analyzer in *J processes the generated traces and computes appropriate metrics. This component is itself a pipeline of modular metric computations and other components, and so provides flexibility in trace processing as well as easy modification to the set of analyses.

Analysis operations in *J are organized hierarchically as `Pack`s and `Operation`s. `Pack` objects are containers for other `Pack`s or `Operation`s, whereas `Operation`s perform computations. This hierarchical formation is conceptually clean and allows for modular use of operations or packs.

Because of the large amount of information that is to be processed for each trace file, an event dispatch system that simply propagates events through the `Pack` tree is impractical. Instead, *J preprocesses this tree and generates a mapping from events to sets of `Operations,` effectively "flattening" the hierarchy while keeping the relative ordering of operations. Only operations interested in an event will therefore receive it, thus eliminating the cost of recursively traversing the tree for each event.

*J's default `Pack` tree contains several operations grouped into different packs based on their functionality:

- *transformers*: Transformers modify the event objects in order to provide services to subsequent operations. For example, one analysis will map JVMPI identifiers to unique identifiers.
- *metrics*: This pack contains all of the analyses that are used as part of our dynamic metrics project.
- *triggers*: Triggers are used to invoke other specific operations when a particular criterion is met, for example the execution of 1000 bytecode instructions. Triggers allow metric values to be computed over various intervals by modification of a simple counting trigger operation rather by having to modify or create new, complex metric operations.
- *printers*: Printers are responsible for converting the trace file to other formats. For example, one printer generates a dynamic call graph based on the recorded method invocations.

2.3 Output Processing

In *J, each operation and/or pack is responsible for generating its own output. This is flexible, but not always convenient, and so the default metric pack is capable of aggregating all metric results and generating a single XML file for the entire set of results.

We provide an XSL style sheet that allows XML files to be viewed in common web browsers by transforming them into standard HTML files. More interestingly, we have implemented a parser for the XML files that is designed to insert the metric analysis data into a MySQL database. This database is then used to generate dynamic HTML pages on our website using PHP. The website supports arbitrary user-defined database queries. This dynamic website has the advantage of being very easy to update since its contents immediately reflects changes made to the database.

3. RELATED WORK

Dynamic metrics themselves are discussed in [6, 4].

There are several JVMPI-based profilers for Java programs, e.g. JInsight [7], JProbe [2], and Optimizeit [3]. These profilers tend to provide a fixed set of performance metrics, and do not allow one to define new calculations or metrics.

Several offline Java profilers exist. "Caffeine" [5] uses Java's built in debugging interface (JVMDI) to produce traces of program behaviour. Caffeine includes a Prolog engine that can answer queries about program execution, with the goal of reconciling high level model conjectures about a program with its runtime behaviour. "Shimba" [9] traces Java programs for reverse engineering purposes. In comparison we focus on generating diverse summary information relevant to compiler/runtime developers.

Online Java profiling systems such as [8] also exist; such approaches of course have very different engineering requirements.

4. FUTURE WORK

There are dynamic metrics that are not possible to compute using the JVMPI agent in *J, e.g. stack and memory layout are necessary for some memory-related metrics. In order to address this problem, we are working on modifying SableVM, an open-source JVM, to generate *J traces. Other possibilities include program instrumentation using SOOT, a Java bytecode transformation framework.

In order to handle large traces more efficiently, integration with trace compression systems are also planned. We are also investigating the use of summary information to encode execution data instead of our current stream-based approach.

Of course, more metrics are continually being added.

5. CONCLUSIONS

We have described the design of a system that solves several problems arising from the need to investigate dynamic behaviour of Java programs. Data collected from this system for a wide variety benchmarks is available on the web at `www.sable.mcgill.ca/metrics`, as well as *J itself. We are actively improving this software, and welcome feedback including metric suggestions.

6. REFERENCES

[1] URL: http://java.sun.com/products/jdk.

[2] URL: http://java.quest.com/jprobe/jprobe.shtml.

[3] URL: http://www.optimizeit.com.

[4] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, 2003.

[5] Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No Java without Caffeine – a tool for dynamic analysis of Java programs. In *17th IEEE Conference on Automated Software Engineering*, pages 117–126, Edinburgh, UK, September 2002.

[6] Sherif M. Jacoub, Hany H. Ammar, and Tom Robinson. Dynamic metrics for object oriented designs. In *Sixth IEEE International Symposium on Software Metrics*, pages 50–61, 1999.

[7] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. Visualizing the execution of Java programs. In *Software Visualization 2001*, number 2269 in LNCS, pages 151–162, 2002.

[8] Steven P. Reiss. Visualizing Java in action. In *Proceedings of the ACM 2003 Symposium on Software Visualization*, pages 57–65, San Diego, California, USA, 2003.

[9] Tarja Systä. Understanding the behavior of Java programs. In *Seventh Working Conference on Reverse Engineering (WCRE'00)*, pages 214–223, 2000.