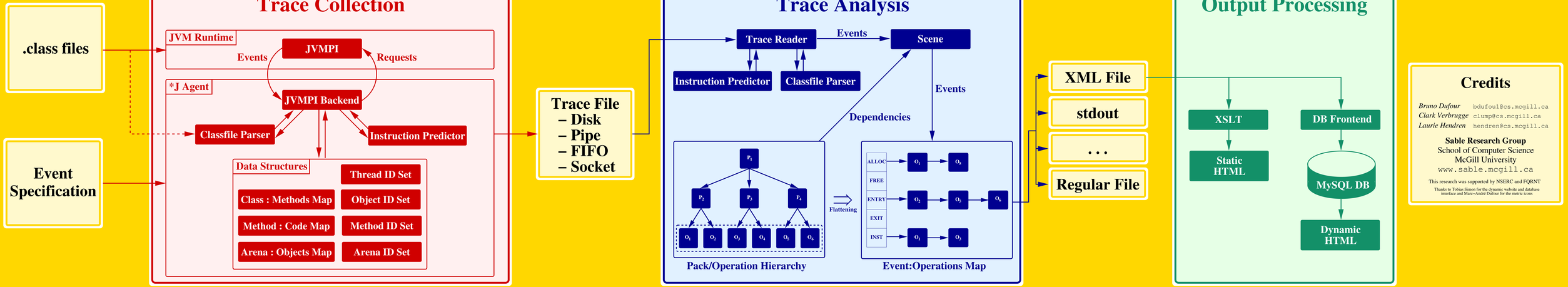
*J: A Tool for Dynamic Analysis of Java Programs



Credits

Bruno Dufour bdufoul@cs.mcgill.ca
Clark Verbrugge clump@cs.mcgill.ca
Laurie Hendren hendren@cs.mcgill.ca

Sable Research Group
School of Computer Science
McGill University
www.sable.mcgill.ca

This research was supported by NSERC and FQRNT

Thanks to Tobias Simon for the dynamic website and database interface and Marc–André Dufour for the metric icons

Overview

Conceptually, *J is a pipeline consisting of three major components. The trace generator produces and serializes a stream of events from a program execution in the form of a trace file. The trace is then passed on to the trace analyzer, which is primarily used to compute dynamic metrics. Finally, the metric data is emitted in the form of an XML file, which is then ready to be processed and incorporated into a database constituting the core of a dynamic website, or viewed using other means.

*J is distributed at:
`www.sable.mcgill.ca/~bdufoul/starj/`

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 trace generator uses the buit–in Java Virtual Machine Profiler Interface (JVMPI) to dynamically receive events from an executing Java Virtual Machine (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 the JVMPI specification.

Unfortunately, while the JVMPI is reasonably ubiquitous it has several drawbacks. The most obvious one is the fact that the 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. The JVMPI implementation 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 the JVMPI is in the data it provides. Not only are the event types fixed (limiting dynamic metric possibilities), but even the event data can be insufficient: the 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. While *J can output GZipped trace files, more appropriate compression techniques and alternative trace generators are part of active future work.

Trace File

While the JVMPI profiler is our main trace generator, we wanted to be able to collect data from other sources, such as instrumented VMs or instrumented programs. Therefore, the trace file format had to be generic enough to allow different data sources to generate traces with potentially different information. We designed the trace files to include space for arbitrary attributes in the header, which are very similar in nature to classfile attributes in Java. They allow a data source to encode arbitrary information about the events found in a trace in a portable way. Standard attributes which must be recognized by all trace consumers are provided to encode common information. For example, one attribute is reserved for encoding the bytecode prediction strategy. In addition, the trace header contains bit masks for each event that indicate which of its fields are present in the trace. Each trace file also gets tagged with a major and a minor version number in order to allow for major changes in the format that cannot be dealt with using simple attributes.

Trace 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 Packs or Operations. Pack objects are containers for other Packs, whereas Operations 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.

Packs and Operations can be added, removed or redefined. *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 transformer maps JVMPI identifiers to unique identifiers.
- **Metrics:** This pack contains all of the analyses that are used as part of the dynamic metrics project.
- **Triggers:** Triggers are used to invoke other specific operations when a particular criterion is met, such as the execution of 1000 bytecode instructions. Triggers allow metric values to be computed over various intervals by modification of a simple counting trigger rather than 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.

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 computations.

We provide an XSL stylesheet that allows such XML files to be viewed in common web browsers by converting them to static HTML files.

More interestingly, we have implemented a parser for the XML files that is designed to insert the dynamic 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. This also facilitates the distribution of dynamic metric results to a group of users, and can form the basis of a benchmark knowledge base.

Examples of Dynamic Metrics

size.appRun.value: This metric measures the number of bytecode instructions which are executed at least once, or *touched*, during the entire duration of a program's execution. This measurement is not affected by dead code. It is intended to capture the dynamic size of a program, and does not include startup.
HelloWorld: 4 Javac: 26267 Compress: 5084

data.floatDensity.value: This metric measures the number of floating–point operations per 1000 executed bytecode instructions. This measurement captures the relative importance of floating–point operations.
HelloWorld: 2.0 Javac: 0.1 Mtrt: 308.5

polymorphism.appReceiverArityCalls.bin: This metric measures the percentage of all calls that occur from a call site with one, two and more than two different receiver types. This measurement captures the amount of polymorphism in a program, and does not include startup.
HelloWorld: 100%,0%,0% Javac: 73%,15%,12% Jack: 90%,10%,0%

concurrency.lockDensity.value: This metric measures the average number of lock operations (monitorenter)per 1000 executed bytecode instructions. It captures the amount of synchronization found in a program.
HelloWorld: 0.19 Compress: 0.0 Jack: 2.13

memory.byteAllocationDensity.value: This metric measures the number of bytes allocated per 1000 executed bytecode instrucitons. It captures the "memory–hungry" aspect of a program.
HelloWorld: 1734 Compress: 11 Javac: 132

More metrics descriptions and values can be found at:
`http://www.sable.mcgill.ca/metrics/`