Design of JFluid: A Profiling Technology and Tool Based on Dynamic Bytecode Instrumentation

Mikhail Dmitriev

Design of JFluid: A Profiling Technology and Tool Based on Dynamic Bytecode Instrumentation

Mikhail Dmitriev

SMLI TR-2003-125 November 2003

Abstract:

Instrumentation-based profiling has many advantages and one serious disadvantage: usually high performance overhead. This overhead can be substantially reduced if only a small part of the target application (for example, one that has previously been identified as a performance bottleneck) is instrumented, while the rest of the application code runs at full speed. Such an approach can also beat scalability issues caused by a high volume of profiling information generated by instrumented code running on behalf of multiple threads. The value of such a profiling technology would increase further if the code could be instrumented and de-instrumented as many times as needed at run time.

In this report we describe in detail the design of an experimental profiling system called JFluid, which includes a modified Java HotSpot™ VM and a GUI tool, and addresses both of the above issues. Our JVM™ supports arbitrary on-the-fly modifications to running Java methods, and can connect with a profiling tool at any moment, without any startup time preparation. Our tool collects, processes and presents profiling data on-line. To perform CPU profiling, it instruments a group of methods defined as an arbitrary "root" method plus all methods that it calls (a call subgraph). It appears that static determination of all methods in a call subgraph is difficult in the presence of virtual methods, but fortunately, with dynamic code hotswapping available, two schemes of dynamic call subgraph revelation and instrumentation can be suggested.

Measurements that we obtained when performing full and partial program profiling using both schemes show that the overhead can be reduced substantially using this technique, and that one of the schemes generally results in a smaller number of instrumented methods and better performance, especially for large applications.

M/S MTV29-01
2600 Casey Avenue
Mountain View, CA 94043

email address:
mikhail.dmitriev@sun.com

© 2003 Sun Microsystems, Inc. All rights reserved. The SML Technical Report Series is published by Sun Microsystems Laboratories, of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

TRADEMARKS
Sun, Sun Microsystems, the Sun Logo, Java, Java HotSpot, JDK, JVM, J2EE, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the US and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@sun.com>. All technical reports are available online on our Website, http://research.sun.com/techrep/.

Design of JFluid: A Profiling Technology and Tool Based on Dynamic Bytecode Instrumentation

Mikhail Dmitriev
Sun Microsystems Laboratories
UMTV29-112, 2600 Casey Avenue, Mountain View, CA, USA
Mikhail.Dmitriev@sun.com

November 17, 2003

1 Introduction

Growing size and complexity of modern software applications increase the demand for tools that automate collecting data about the dynamic behaviour of programs, and thus allow developers to identify performance bottlenecks in their applications with minimum effort. The process of automatic collection and presentation of data characterizing performance of running programs is called profiling. For an object-oriented language such as the Java™ programming language, that features automatic memory management, built-in multithreading and thread synchronization mechanisms, etc., several forms of profiling are useful in practice. CPU profiling determines how much time the program spends executing various parts of its code; memory profiling determines the number, types and lifetime of objects that the program allocates; hot lock profiling determines congested monitors, etc. For applications built according to higher-level standards, for example Enterprise Java (EJB) applications, specialized, high-level kinds of profiling exist, such as measuring the EJB transactions throughput.

Instrumentation-based profiling works by inserting, or *injecting*, special packets of code, called instrumentation code, into the application to be profiled (we will call it *target application* or TA). When the injected code is executed, it generates events, such as method entry/exit or object allocation. This data, usually in the processed form (e.g. the total time spent in each method, or a total number of allocated objects of each type), is eventually presented to the user.

The main advantage of instrumentation-based profiling over other known techniques — sampling-based profiling and, in case of programs running on top of a virtual machine (VM), the so-called VM hooks[1], is its flexibility. Virtually any kind of data, ranging from low-level events to high-level data, such as EJB security check occurrences, or GUI event firing, can be obtained using this technique. For high-level data, instrumentation (whether performed by a special tool or embedded into the application itself) is the only way of data collection. For CPU performance measurements, the advantage of instrumentation compared to its main rival, sampling-based profiling, is that instrumentation records the exact number of events such as method invocations; is capable of measuring precise timings (not a statistical approximation, as it happens with sampling); and can be used selectively (for example, on just a few methods, or when instrumented methods are called with a specific parameter value only, etc.). Thus, instrumentation profiling has an advantage when it is required to profile a number of short-running and infrequently executed methods, or in a system where total code profiling, even using sampling, will result in an unacceptable overhead. Furthermore, the fact that instrumentation records all events as they happen while the program runs, can help to restore the important details of program execution history, such as what methods a given method called and in what order, a critical execution path, and so on. Last but not least, instrumentation mechanism is relatively portable, i.e. with it one can implement many useful features without any special support from the VM.

However, instrumentation-based profiling is often associated with high temporal overhead. Injected instrumentation code takes its own time to execute; its presence may prevent some optimizations, such as method inlining, that could have otherwise been made to the target application code; finally, additional code injected into the original code may result in many more CPU cache misses than when the original code is executed as is. Finally, for multithreaded

[1] The outcome of recent evaluation of JVM profiling hooks can be found in [9]. Based on these findings, the Expert Group established to define the new standard JVM profiling specification has recently decided that in this specification [1], many of the VM-generated events, including method entry/exit and object allocation, will be optional and not required to be supported by all conforming JVMs. Bytecode instrumentation is the recommended mechanism for their generation.

applications the amount of generated profiling data grows proportionally to the number of threads, and online processing of this information may raise additional thread contention issues, that need to be specially addressed. For these reasons, total execution time overhead measured in "factors" rather than per cent is not uncommon [3, 5, 7], and overheads in the range of 10,000 per cent (100 times slower) have been reported [7]. Those who used modern profiling tools such as OptimizeIt [8], JProbe [15] or VTune [12] in the "instrumentation profiling" mode can probably confirm this observation. The problem is made worse by the fact that presently most of the tools use static or class load time instrumentation. That means that the code remains instrumented, and thus incurs execution overhead, during the whole run time of the target application.

One way to address this problem is to perform selective instrumentation. For example, the overhead of instrumenting only a few known key pieces of the application may be just one or two per cent, making it possible to use this technique even for applications running in the field, as reported, for example, for the Wily's Introscope monitoring tool [25]. The problem is that minimal and nonobtrusive instrumentation is usually good only to confirm that there is indeed a performance problem, or, at best, to roughly identify the code area responsible. On the other hand, even heavyweight instrumentation of the part of the application is likely to be tolerable in many situations, as long as it is there only when needed to pinpoint the real performance problem.

In this report, we describe an experimental technology and the same-named tool, called JFluid, that address the above issues by allowing users to select and profile a limited subset of methods of a Java application, and repeat that as many times as needed while the application is running. The underlying mechanism enabling these features is *dynamic bytecode instrumentation*, that supports injecting (and removing) instrumentation bytecodes into/from methods of a running program. In its present shape, JFluid is more suitable for development-time profiling rather than for production-time problem diagnostics. However, it appears that even at development time the approach that we have taken is very useful, as confirmed by our own measurements and the feedback from numerous JFluid users.

Our present implementation consists of the JVM™ (currently based on Java HotSpot™ VM version 1.4.2) that has been modified to support dynamic bytecode instrumentation, and the GUI tool. The JVM allows the user to start it without any special preparation (such as special command line options, presently needed in most of the industrial JVMs to enable various profiling-related capabilities), then attach the tool to it at any moment, instrument the application, collect and analyse the data "on line", and finally detach the tool, allowing the target application (TA) to run at full speed again. The tool currently supports CPU and memory (object allocation) profiling. Both the modified JVM and the tool are available for free download and evaluation at http://research.sun.com/projects/jfluid.

This report makes the following contributions:

1. We describe in detail the key components of our profiler, which can probably be used as a "reference implementation" by developers of similar tools. We demonstrate that a profiler for Java can be written mostly in Java itself.

2. While the idea of using dynamic bytecode instrumentation for profiling is not new, we make a contribution by suggesting how it can be used for CPU profiling in a new, more efficient way. Firstly, we suggest to instrument methods of a *call subgraph* (defined as a single chosen method plus all methods that it calls, directly or transitively), that the tool automatically reveals, based on a single "root" method specified by the user. Secondly, we show how dynamic bytecode instrumentation itself helps to solve the problem of fast and highly-precise identification of all methods in a call subgraph.

3. We experimentally evaluate our technology on different types of applications, showing in what circumstances our approach is most beneficial.

The rest of this paper is structured as follows. In Section 2 we describe the general architecture of our system. In Section 3 we describe the attachment mechanism used to attach our tool to a running JVM. The next section describes how method hotswapping works in our modified JVM. In Section 5 we discuss how the generated profiling events are recorded, and the resulting data is transferred to, and processed by, the tool. In Section 6 we describe two schemes for dynamic revelation of a call subgraph that is triggered by the "instrument method transitively" command. In the next section, we present the results obtained by running our tool on several benchmarks, using both of the above schemes and either full or partial application profiling. In Section 8 we discuss related work, and finally, in Section 9 we present conclusions and some thoughts about future work.

2 JFluid Architecture

.

Of the requirements to an introspection system, one of the most important is to keep its impact on the target VM as small as possible, in terms of CPU cycles, heap space usage and other internal resource consumption. On the other hand, ease of development, change and maintenance is also important, especially for a research tool. The present architecture of JFluid is an attempt to address both of these requirements.

In order to minimise the system's impact on the target VM, a client–server model, depicted in Figure 1, was adopted for JFluid. The target VM acts as a *server*, with the JFluid server, or back end, code, executing on it in parallel with the TA. The JFluid tool communicates with it using TCP/IP sockets and a shared-memory file (why we use both of these mechanisms will be explained a bit later).

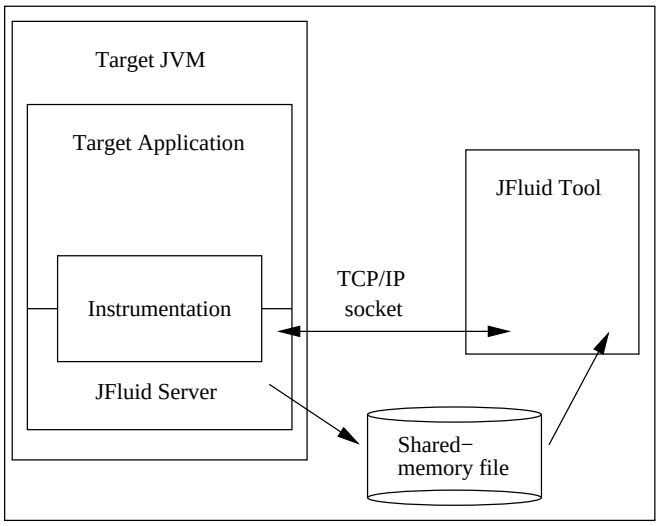

Figure 1: JFluid Architecture

The server is a relatively small, but most important part of JFluid, since its properties affect the performance and usability of the system to the greatest extent. Once activated, the server starts a single thread that handles the communication with the tool. It receives and executes the tool's commands, and also informs the tool of certain events that may occur while the TA is running. The injected instrumentation code executes on behalf of the TA, but structurally is also a part of the server code. It collects and records the profiling data, and may trigger some events, that are handled by the server and then, in some cases, by the tool.

The JFluid server is written in Java, with only a small amount of C code. This code acts simply as a bridge between the server and our special VM-internal API, which is similar to JVMDI (JVM native debugger interface [21]). This API can be viewed as a subset of JVMDI: it provides a small number of calls to get VM-internal information (for example, to get all loaded classes or running threads), handle some VM events (e.g. class loading), and to hotswap the code of running methods (which is how dynamic bytecode instrumentation is performed). The rest of the server code is written in Java. That includes, in particular, the code that records profiling events and transfers this data to the tool, and the code responsible for general communication between the server and the tool.

The advantages of the back end written in Java are: ease of development, increased portability, and even, possibly, a better performance of the instrumentation code. The latter is due to the fact that crossing the boundary between Java and native code is a relatively expensive operation. Thus, frequently executed instrumentation code written in Java and compiled with a sufficiently smart dynamic compiler, such as the one available in HotSpot, may execute even faster than the native code.

Writing the back end of an introspection tool in Java does not, however, seem to be a universally accepted approach, since, for one thing, the Java classes that belong to the tool, and the data structures that they create in the Java heap, inevitably distort the normal memory layout, and possibly the performance, of the TA. While this is generally true, we believe that for large applications that are the primary target of JFluid, the impact of this additional code and data is negligible. To ensure that, we took the following measures:

- Retained as much profiler code as possible at the tool side;
- Designed the system in such a way that the size of the data generated by the server and retained at the target VM side remains limited during the TA execution lifetime (see Section 5.2).
- Implemented our wire protocol avoiding the use of Java Object Serialization (see below).

Another consideration is that, when judging whether or not Java code is appropriate, we should distinguish between the types of introspection tools. Specifically, in debuggers a back end written in Java would be problematic due to a high probability of deadlocks. They will most probably become an unsurmountable problem, since both the back end and the TA would use Java threads and compete for the same VM internal resources, such as an internal class dictionary or various heap locks, while the back end would need to perform some VM internal functions, such as asynchronous thread suspension, that a normal Java application can never use. In our profiling code, however, this does not appear to be a problem, since its interaction with the target VM internals is quite limited and is mostly in the form of queries. Asynchronous thread suspension/resumption is simply not used. Thus, so far we did not have any serious problems due to the JFluid server being written in Java, and the time spent on fixing occasional bugs in this area is fully compensated by the increased maintainability and other adavantages of the Java code. The performance of our system also looks comparable to that of several commercial profiling tools that we tried.

To transmit information between the tool and the server we use both a custom, packet-based wire protocol, similar

to e.g. JDWP [22] over TCP/IP sockets, and a shared memory file. While commands, responses and the data that they carry (the amount of which is relatively small) is transmitted using TCP/IP sockets, the acquired rough profiling data is transmitted using a shared memory file. This is due to the high volume of this data and the need to transmit it as fast as possible to minimize the run time profiling overhead (see Section 5.2 for further discussion). We use both sockets and shared memory as opposed to just shared memory, because in future we plan to extend our system such that the target VM and the tool can run on separate machines. In that case, TCP/IP sockets will be the only universal communication mechanism. However, to retain advantages of processing rough profiling data outside the target VM, we will still have a small "proxy" component running on the same machine, that would get the rough data through the shared-memory file, process it and send relatively compact intermediate representation to the tool over the network.

To minimise the impact of the communication code written in Java on the target VM, we implemented our own custom mechanism of data transmission over TCP/IP sockets, which does not use standard Java Object Serialization. Commands of our wire protocol are mostly simple, each consisting of a few integers and strings, or at most a few arrays of those, so it was not difficult to write our own custom serialization code. This helped us to avoid "polluting" the Java heap with many additional classes that are loaded and even automatically generated by the standard Serialization mechanism. The performance of our mechanism is also superior to that of the Java Object Serialization.

Finally, the JFluid GUI tool is a Java application taking advantage of automatic memory management, portable graphics libraries, and other merits of the Java platform. It can either start the target VM or attach to an already running one (see Section 3 for details). As the TA executes, the tool processes the rough profiling data generated by the server and builds a compact and informative representation of profiling results (see Section 5), that can be viewed by the user at any moment.

3 Establishing Connection with the Target VM

Like most other profiling tools, JFluid can work with the target VM/application in two modes: by controlled application startup and by attaching to a running VM. Obviously, the first mode is much easier to implement. In our implementation, the tool would simply start the target JVM with its own `ProfilerServer` main class, passing it all the information about the actual target application (that the user previously entered in the tool) through the command line. The server class' `main` method first spawns a thread that establishes a connection with the tool and keeps listening on the communication port, and then invokes the main method of the target application.

It appears, however, that attaching to an already running JVM is presently much more popular with JFluid users. "Ergonomics" is the main reason for using this mode even at the development phase, since many large Java applications are started with a very long and cumbersome set of command line options, class path entries, and so on. These options may be spread among several script files, and therefore copying them all into the tool for controlled startup, even if done just once, may be quite painful. It is much more convenient to just start the application as usual, and then attach the tool to it. In future, as we mentioned in the introduction, we hope that the capability to attach the diagnostic tool to any running JVM, started without any special preparation, will become one of the key pieces of the general "emergency servicing" concept.

To our best knowledge, no existing introspection tools for Java that need to perform some activity within the running JVM, support attachment without preparation. Instead, they require that the target JVM is started with special command line options that typically activate some additional code (an equivalent of our profiler back end) right from the beginning. This code would then listen on a port or an alternative communication channel, waiting for the tool to come up and initiate an active session. One good reason for these tools to activate the back end very early, is that it is the only way for them to intercept and instrument all application classes.

In JFluid, we do not have the latter concern, and we wanted to avoid any special JVM startup-time preparation completely. Therefore, we came up with a solution, which is simple, and yet, to the best of our knowledge, is not presently used in any other Java VMs. To activate the introspection back end (in fact, this mechanism can be used for any other code as well) we use an OS signal handler in the VM. An OS signal in UNIX$^{(R)}$ does not carry any additional information with it, so in order to make this mechanism flexible enough, we suggest the use of a convention, according to which, once the VM receives a predefined OS signal, it goes to some predefined file system directory, and looks for a file with a predefined name. If the file is not there, nothing happens. If the file is there (it is the tool's responsibility to generate it, and then delete it when the connection is established), it should contain the full path to the native library to start, and any parameters to the function (with the predefined or specified name) that serves as an entrypoint. The VM then would load the library and start the function with the given parameters in a specially created thread.

The implementation of this mechanism appeared to be straightforward. The HotSpot VM already has some sig-

nal handlers. One of them, for the UNIX signal SIGQUIT, can also be activated by pressing `Ctrl-\` (on Solaris) or `Ctrl-Break` (on Windows) on the keyboard. This signal handler presently dumps the stacks of all Java threads currently running in the VM. We simply added code to it that would, after dumping threads, look for a file and perform other operations as described above.

One remaining question is how to choose the directory that the target JVM would use to lookup the session startup file. We initially made our JVM use the directory from which it was started, thus requiring the tool user to specify both the JVM process PID and this directory to establish connection. Then we came up with a better, though non-general solution, which uses the non-standard mechanism known as `jvmstat`[24], available in HotSpot starting from version 1.4.1. This mechanism makes any HotSpot JVM create on startup a special, one-per-user directory, that is accessible only to that user. The VM then places a file into this directory, that contains information allowing the `jvmstat` tools to read certain performance data that the VM dumps periodically. This is in fact another "introspection always on" mechanism available in HotSpot, although it supports only data reading, in contrast with JFluid's active interaction with the JVM. The bottom line is that for any running HotSpot VM there is now a write-protected directory, that can be unambiguously identified by both the target VM and the tool, given the user ID and the target JVM PID — which is exactly what we need.

As a final note, we would like to add that our mechanism is as secure as the underlying OS. In particular, in UNIX only the user who started a process, or a super-user, can send any signal to it. The signal can travel only within the same machine. Furthermore, the directory which we use to pass the information is unaccessible to other users.

4 Dynamic Bytecode Instrumentation

4.1 General Approach

At the time when the JFluid project started, the *dynamic class redefinition* capability, in the form of the `RedefineClasses()` call in JVMDI [21, 10], was already available in the HotSpot VM. However, it turned out that the general class redefinition functionality provided by this call is technically not quite suitable for massive, yet individually small changes to methods, that characterize bytecode instrumentation. `RedefineClasses()` takes the arguments that are, essentially, the pointer to the "old" class object, and the byte array representing the whole "new" class file. Before actually hotswapping the two classes, it parses the new class file, installs an internal class object for the new class version, compares the two versions to make sure that only supported changes have been made in the new version, etc. (see [10] for a detailed description). So if, for example, we need to instrument 1000 methods in 1000 different classes, with each of these classes containing a few tens of other methods, the overhead due to generating 1000 whole rewritten class files, transferring them over the wire, and then redefining each of these classes completely, may be very significant.

Fortunately, if it is known in advance that modifications that we are going to apply are injection of method calls into existing bytecodes, or, generally, any modifications that obey certain technical constraints, an optimized solution can be suggested. Given the way in which classes are represented in memory in the HotSpot JVM [10], and other properties of this system, these constraints are:

1. Nothing in a class except the code of existing methods is modified. This saves us a lot of run time work needed to restructure the internal class object if, for example, a method or a data field is added (see [10] where the process of full-fledged class evolution is described).

2. No existing *constant pool* [13] entries are changed or removed, so that the same constant pool entry index *x* in the original and the instrumented method code refers to the same constant value. This saves us time parsing the whole new constant pool, and also guarantees that both old and new method versions can operate with the same new constant pool.

If the new code satisfies the above constraints (which are easy enough to obey if we only instrument the code, preserving its original semantics and actual bytecodes), an optimized API can be proposed, that consists of just two calls: `RedefineMethods(methodIds[], byte[][] newMethodBytecodes)` and `ExtendConstantPool(classId, byte[] addConents)`. We chose to support extending only one constant pool in one go, since this operation is relatively rare and does not have any effect until methods that may use the added constant pool contents are changed. In contrast, atomic redefinition of multiple methods is often necessary. Furthermore, methods may be redefined frequently and in large groups, and therefore aggregating multiple redefinitions into one call, that performs some common operations only once, reduces the overhead significantly.

4.2 Dealing with Compiled and Inlined Code

Technically, run time hotswapping of a method is not such a challenging issue as it might seem. The task is simplified by the fact that our current policy for switching between the original and the instrumented method version is: "all active invocations of the original method complete as is, and all calls to this method that occur after hotswapping go to the

instrumented method version". In other words, if a method that we want to instrument is currently running, we don't attempt to switch execution to the instrumented code immediately. The latter feature does not currently seem to be very valuable in practice, since, in particular, methods that never exit because of e.g. spinning in an endless cycle are rare and can be addressed separately if needed.

Given this policy, method hotswapping means essentially locating all pointers to old method versions and switching them to their respective new method versions everywhere in the program. However, this may be a real challenge in a VM that, like the HotSpot VM, runs Java applications in mixed mode, when initially the program is interpreted, and gradually the most actively used methods are compiled into native machine code [23]. One compiled method can call another using a direct `call` machine instruction; furthermore, some methods may get inlined into their callers. In the latest versions of HotSpot, inlining is performed aggressively, which means, in particular, that virtual methods can be inlined if, at compile time, only a single method implementation can be called from a particular call site.

Fortunately, the so-called *deoptimization* mechanism (initially introduced in the SELF system [11]), that solves this problem, is already available in HotSpot. It works essentially by recording, at compile time, the information about methods called directly, and about "caller - callee" pairs when a callee is inlined. Actual deoptimization is performed if we need to switch to interpreted execution of some compiled method — for example to step through it in a debugger, or because it is a virtual method that cannot be inlined anymore (a class in which this method is overridden has been loaded). Deoptimization happens in two phases. In the first, eager phase, the execution is suspended, the stacks of all application threads are scanned, and all compiled stack frames for methods that should be deoptimized are patched in a special way. The execution then resumes, and the second phase happens for each affected stack frame individually, when a callee method is about to return to the method that was deoptimized, say `m()`. At that time, the stack frame of `m()`, which is now the topmost, is converted into the format of the interpreter, and the current instruction pointer is set to the point in the bytecode that corresponds to the current point in the machine code. The procedure is complicated by the fact that `m()`itself, and possibly its callers, may be inlined; thus a single native stack frame may be replaced with several interpreted stack frames.

4.3 Method Hotswapping Procedure

The procedure of replacing methods with their instrumentation copies, performed by our `RedefinedMethods()` call, consists of the following two phases:

1. *Parse method bytecodes and create internal method objects*. This operation is performed in an ordinary Java thread, without suspending other application threads, and therefore causes minimal slowdown, especially if the VM runs on a multiprocessor machine. Each byte array representing a new method version is parsed using the standard JVM class file parser, and verified as usual. If these operations are successful, a new method object is created.[2] New method objects are made reachable from a temporary array.

2. *Activate the instrumented code.* Once the previous operation is performed for all methods scheduled for instrumentation, we suspend all application threads, bringing them to the safe point. Then we deoptimize all methods depending on our instrumented methods. Finally, we switch pointers in all relevant objects on the heap, so that they now point to instrumented method versions. Places where such pointers may occur include constant pools of classes that reference these methods, and virtual method tables of the class that defines an instrumented virtual method, plus all its subclasses.

After the second operation is complete, the application threads are resumed, and the next call the target application makes to an instrumented method, goes to its new version. The invocations of original method versions that are currently active complete as is.

Hotswapping of constant pools are performed in much the same way. In this case, however, we don't need to perform any method deoptimizations, and we only need to switch pointers from methods of one class object and its methods to the new constant pool.

5 Data Collection and Transmission

Currently JFluid supports collection of two kinds of CPU profiling data. The first, more sophisticated kind (which, however, imposes a much higher run time CPU overhead) is a Context Call Tree (CCT) (explained below), plus a sorted list of accumulated net times for individual methods, that are built for a chosen call subgraph of the TA. The second kind of profiling data (usually collected with virtually no overhead) is gross time for a single code region. Since this profiling technique is very simple (essentially equivalent to injecting two `getCurrentTime()` calls into a chosen TA method, calculating the difference between the two results, and recording it in a fixed-size ring buffer), in the following

[2] Internally in HotSpot, methods are separate objects, that are attached to class objects. Further description of how various structures constituting a Java class are represented internally in HotSpot can be found in [10]

discussion, we will only cover various aspects of how instrumentation and other parts of the JFluid system support call subgraph profiling.

The *calling context tree* (CCT) format of profiling results representation was first introduced by Larus et al. in [3], and is presently used, for example, in the popular OptimizeIt [8] commercial profiling tool. The reader can find a detailed explanation for this and other formats in the above work; a short example in Figure 2 illustrates it and is hopefully self-explanatory.

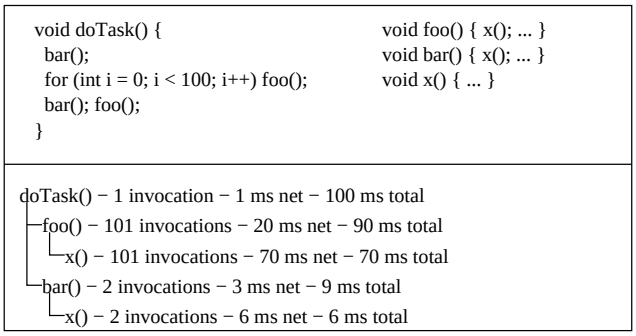

Figure 2: A program and its Calling Context Tree (CCT)

We chose this presentation format for JFluid since in our opinion it is the most compact and yet informative one. Its convenience is further increased by the fact that in the GUI the user does not initially see the entire tree, which can be quite large. Instead, they first see only the top-level nodes, and then can follow interesting paths down the tree by clicking on, and thus "expanding", subnodes.

When collecting data in this format, we inject instrumentation bytecodes into all methods that are reachable from an arbitrary root method specified by the user (`doTask()` in Figure 2). Section 6 explains how these methods are identified. Our instrumentation consists, essentially, of calls to the following two methods: `methodEntry(char methodId)` and `methodExit(char methodId)`. Both of them are `public static void` methods, defined in our own `ProfilerRuntime` class. We arrange that the target VM loads it using the bootstrap class loader, to ensure that there will not be any problems with calling our instrumentation methods from any class loaded by the target VM, including the bootstrap classes. Each instrumented method of the TA is assigned its own `char` identifier, which is hard-coded into the calls to `methodEntry()` and `methodExit()`.

The instrumentation code is relatively complex, since, in addition to its main responsibility — taking the current timestamp and recording the relevant event — it needs to perform a number of other operations necessary for correct data collection in the CCT format. The main reasons why these additional operations are needed are:

1. A limited subset of the TA methods is profiled, but any method that is within a call subgraph can also be called from a site which is outside this subgraph. Therefore, the instrumentation has to detect whether it is called in the profiled subgraph context or not, and in the latter case, has to return as quickly as possible.
2. Instrumentation itself may call some Java core classes code, which also may be instrumented. We need to prevent recursive instrumentation calls to avoid infinite recursion, misleading measurements, and other problems, such as deadlocks.
3. The TA code may throw exceptions, which, if measures are not taken, would result in `methodEntry()` calls with no matching `methodExit()`.
4. In order to measure exact elapsed time for profiled methods (see Section 5.3), we need to guarantee that each invocation of `methodEntry()` and `methodExit()` takes a constant amount of time, and that any deviations from this constant time can be predicted and measured in turn, to be compensated in the data presented to the user.
5. Profiling data should be available for presentation at any moment, so that we can monitor the performance of an application as it runs.
6. Collected data should not take a noticeable part of the target JVM heap, and, ideally, should be of small constant size.
7. If the data is dumped on disk, it should also take a relatively small and, ideally, constant space.

In the following subsections, we discuss how each of these requirements in turn is addressed in JFluid.

5.1 Context-sensitive Activation of Instrumentation

As we explained above, the instrumentation methods should emit their respective events only if they are called in the proper context: when the TA method that calls instrumentation is called from within the profiled subgraph, i.e. directly or transitively by the subgraph root method, and not recursively by any instrumentation method. To address this and other issues, we maintain a special per-thread data structure in the JFluid server, called `ThreadInfo`. It contains, in particular, two boolean fields (flags) called `inCallGraph` and `inProfilerRuntimeMethod`. Both of them are initially set to false. They are further used in the instrumentation code as shown in Figure 3, where (in a slightly simplified form) the relevant parts of the `methodEntry()` method are presented. The comments in the code make clear how these

flags are utilized. The structure of `methodExit()` method is largely the same, except that, after determining that we are inside the call graph, we first check if the simulated stack depth is zero. If so, it means that we are in the topmost invocation of the root method, and thus will leave the call graph when this invocation completes. Therefore if `ti.stackDepth == 0`, we set the `inCallGraph` flag to false before leaving `methodExit()`.

In order to handle exceptions properly, we add to each instrumented method the bytecode equivalent of the Java `try{ ... } finally { methodExit()}` statement. In this way, even if a method exits abruptly, our instrumentation is still called by it, ensuring the consistency of collected data — that is, the exact match between the real and the simulated stacks.

5.2 Recording and Transmitting Profiling Data

Now we will describe how the requirements of on line data presentation, constant execution time for instrumented methods, and minimum disturbance to the target JVM heap, are addressed in JFluid. In the light of these requirements, none of the traditional simple solutions would work. For example, writing each event into a file as it happens, or storing it into a memory buffer and dumping it periodically into a disk file, which the tool can read on-line, would involve a performance penalty and may create a large (up to tens of GB) files for applications running for long enough time. An alternative solution, which is used by some commercial tools, for example OptimizeIt [8], is to process the data at the target VM side and keep only the final compact representation (a CCT, as in our case) permanently. We initially implemented this solution, but unfortunately, for realistic applications even the compact representation of the profiling data may grow to a considerable size. For example, in one of our tests, a run of the `javac` Java compiler compiling a relatively small (a few thousand lines) Java application caused more than 3 million method invocations, and the resulting CCT appeared to contain several thousand nodes that occupied more than 3 MB on the Java heap. Not only the heap image was polluted by the node objects, but also the performance data was distorted because of creation and garbage collection of many thousands of objects allocated and thrown away while the CCT was being built. Some tools seem to avoid at least the heap pollution problem by building the CCT in the C memory using the native C code. We, however, wanted our code to be portable, and also wanted to avoid additional overhead due to making native calls. Another consideration is that, although at this time we present only the CCT to the user, in future we may want to add more metrics, such as timeline or a critical execution path, and therefore, on line data processing may become too complex and obtrusive to be done on the target VM side.

The solution that we came up with, that satisfies all of the above requirements, is a combination of rough data collection and periodic data dumping at the target VM side, and its online processing in the tool. To speed up transmission of rough profiling data to the tool, we dump the above buffer into a shared-memory file (using the API defined in the `java.nio.MappedByteBuffer` class), which works much faster than writing and reading an ordinary file. The tool reads the buffer and builds the next portion of the CCT upon each dump, and can also asynchronously request the contents of the not yet full buffer.

To store the data as it is generated we use a relatively small (currently 1 MB) byte buffer allocated on the Java heap. This object, although it distorts the heap image to a certain extent, does not affect the garbage collection time significantly (the generational garbage collector used in the HotSpot VM is likely to promote this object into the old generation soon, and pay little if any attention to it afterwards). Each call to `methodEntry()` and other instrumentation methods stores a corresponding event record into this buffer.[3] For example, for the method entry this event record consists of an event code, thread identifier, and a timestamp. If the event recording code discovers that a buffer is full, it remembers the current time (let us call it $time_0$), and then makes a call, which dumps this buffer into the shared memory file, and notifies the tool. The server then waits for the tool, that processes the data and notifies the server when it is finished. When processing is complete, the thread that initiated the dump takes the current time ($time_1$) again, and then records a special "adjust time" event, with the argument equal to $time_1 - time_0$. Note that other threads, before attempting to write something to the buffer, also need to check if it is currently being processed. If so, they use the same mechanism to "factor out" the time spent in data processing from the profiling results.

The described mechanism, by avoiding any operations that may take noticeably varying and unpredictable amounts of time, ensures that the execution time for `methodEntry()`, `methodExit()` and other instrumentation methods is constant, and possible deviations are recorded and handled properly. This property allows us to factor out the time spent in instrumentation itself from the profiling results, as explained in the next section.

[3] The actual design is more complex: small thread-local event buffers are used to avoid locking the main buffer at each event write, which can cause high thread contention. Dumping a thread-local buffer into main buffer once in a few hundred method invocations practically eliminates this problem.

```
public static void methodEntry(char methodId) {
  ThreadInfo ti = getThreadInfo();           // Get record for current thread
  if (ti != null && ti.inCallGraph) {
    if (ti.inProfilerRuntimeMethod) return;  // Avoid recursive instrumentation invocation
    ti.inProfilerRuntimeMethod = true;
    ti.stackDepth++;                         // Increase the simulated stack depth for this thread

    // ... Obtain current timestamp and record the "method entry" event

    ti.inProfilingRuntimeMethod = false;
  } else if (methodId == rootMethodId) {  // Entered root method from outiside this call graph
    if (ti == null) {  // This thread never entered this call graph before
      ti = ThreadInfo.newThreadInfo();
      // ... Record the new profiled thread creation event
    }
    ti.inProfilerRuntimeMethod = true;
    ti.inCallGraph = true;

    // ... Obtain the current timestamp and record the "root method entry" event

    ti.inProfilerRuntimeMethod = false;
  }
}
```

Figure 3: General structure of `methodEntry()` instrumentation method

5.3 Obtaining Exact Elapsed Time for Profiled Methods

One consequence of the fact that instrumentation injected into the TA consumes CPU cycles is that, if measures are not taken, the CPU performance data presented to the user may become significantly distorted. It appears that the cost of high resolution timer functions available in Solaris (`gethrtime()`) and in Windows NT/2000 (`QueryPerformanceCounter()`) is quite high. For example, on a Sun UltraSPARC(R) II 450 MHz processor machine running Solaris™ 2.8, each `gethrtime()` call costs approximately 0.2 microsecond. The cost of `QueryPerformanceCounter()` call on Windows seems to be much higher relative to the processor speed. In our experiments, on a 600 MHz Pentium III machine, each such call takes approximately 1.67 microseconds. Many other operations, for example integer arithmetics, take much less time. Therefore for an average instrumented method, the time spent in instrumentation cannot be ignored if we want to obtain performance measurements that are realistic, as illustrated in Figure 4. If we ignore the time spent in instrumentation, the time spent in the method that we obtain is equal to $(y_1 + t_{exact} + x_2)$. If t_{exact} is small compared to the other two components, the cost of the method as we measure it will be dominated by the cost of the injected instrumentation. If the number of calls to such a method is high, the obtained results may become misleading, showing that the application spends a large part of its time in this method — whereas in reality this time is spent in instrumentation code.

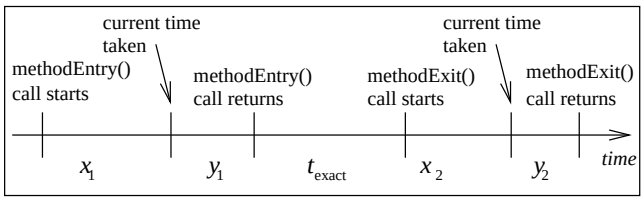

Figure 4: Time distribution in an instrumented method

The simplest way to factor out the time spent in instrumentation so that the final results presented to the user represent only the time spent in the TA itself, is to measure in advance the average time it takes to execute each of the instrumentation methods (calibrate them), and then subtract this value from the rough data. That is why it is so important for these methods to be written such that their cost is as close to constant as possible (see Section 5.2). It is also worth noting that on a VM that has a dynamic compiler, the execution time for instrumentation will vary significantly, depending on whether or not this method and

other methods that it calls are running interpreted or compiled. To make sure that instrumentation always runs compiled, and thus that each call's cost is the same, our calibration procedure, that is performed on the server side every time before starting the TA, executes both `methodEntry()` and `methodExit()` methods for a large number (currently 10,000) of times, measuring the average execution time periodically (for every 100 pairs of calls). It returns the minimal measured result as the final value that will be used in data processing.

To obtain the exact time spent in a TA method t_{exact}, we need to know not just the total time spent in `methodEntry()` and `methodExit()`, but the part of this time equal to $(y_1 + x_2)$, which we call "inner time", t_I. It appears that in practice we cannot simply assume that this value is a half of the total time — this is generally not true, and for some short-running methods this assumption can result in gross errors in results. Fortunately, in our case it is easy to measure the inner time, since both `methodEntry()` and `methodExit()` record the event timestamps. Thus, if we execute a pair of the above calls, without any code between them, for n times, we will get a recorded sequence of n pairs of timestamps: $t_{11}, t_{21}, t_{12}, t_{22}, ..., t_{1n}, t_{2n}$. It is clear that the average inner time t_I can be calculated as:

$$t_I = ((t_{21} - t_{11}) + (t_{22} - t_{12}) + ... + (t_{2n} - t_{1n}))/n$$

Note further, that this correction technique cannot be used with an arbitrary form of method CPU time accumulation and presentation. Specifically, if we only measured accumulated time for each method, registering just the time between entering and exiting it, and not taking into account the time spent in methods that it calls, this technique will not compensate for the time spent in instrumentation invoked by this method's callees. In our case, however, we accumulate the profiling results in the form of a CCT, where for each method in each context it is known how many outgoing calls were made. When we build a CCT, we "stop the clock" for the caller method, and "start the clock" for the callee when processing a method entry event, and do the opposite when processing a method exit event. This way we obtain a "net" elapsed time for the caller method. However, this net time, even after subtracting t_I from it, still remains inexact, because it includes the x_1 and y_2 components of instrumentation that was invoked by its callees. We call the sum $(x_1 + y_2)$ the "outer time", or t_O. Knowing t_I (see above), it is trivial to calculate t_O. Taking into account all the effects of instrumentation that we can predict and measure in advance, the final formula for the exact time for a method that made m direct calls to other methods, and whose total measured time is equal to t_{meas}, looks like:

$$t_{exact} = t_{meas} - t_I - mt_O$$

6 Dynamic Revelation and Instrumentation of a Call Subgraph

Given the inevitable complexity of instrumentation code that we previously explained, it is quite important to be able to instrument an interesting group of methods rather than all of them in order to reduce the overhead. However, to the best of our knowledge, in all existing profiling tools for Java reduction of profiled code volume is implemented quite primitively. These tools, for example OptimizeIt [8], JProbe [15], VTune [12] or JRockit Management Console [6] allow the user to limit the number of methods to profile by manual selection only. The user may pick up methods that they want (or don't want) to profile by hand, or by specifying whole classes or packages. Manual selection can become very tedious and error-prone as soon as the number of interesting methods exceeds a few tens, whereas more coarse-grain selection often leads to increased run time overhead with no additional useful information.

A better alternative in many situations could be the tool itself "discovering" interesting methods to profile, based on some user-specified criterion, that is more sophisticated than class or package name. Of such criteria, one seems to us to be of a particular sense, and that is a *call subgraph* defined as a set of all methods called directly or transitively by an arbitrary, user-specified root method. A technique where the user specifies just a single root method, and the tool determines all methods reachable from it directly or transitively is very easy to use, and, most importantly, looks quite useful, since a call graph seems to match the typical idea of a "task" in the program. Indeed, one method typically performs some task, with other methods that it calls performing, logically, sub-tasks of this task. It is natural to ask questions like "how much time this task takes to complete", or "how many objects have been allocated by that task", then go to the most expensive sub-task (a bottleneck), ignoring the cheaper ones, etc. Call graph instrumentation allows one to do exactly this sort of thing, avoiding the run time overhead due to instrumented methods that are not called within the selected task. Furthermore, instrumenting a call subgraph and presenting data in the CCT format fit together very well: we instrument only those methods that we want to see measured.

However, it appears that identifying and instrumenting methods within an arbitrary call subgraph of Java methods is not simple. To understand the reasons for that, consider how we can actually determine all the methods that can be called, directly or transitively, by an aribtrary method. This can be done statically, by scanning and analysing bytecodes. But to begin with, static bytecode analysis will not

allow us to determine what methods are called using the Java reflection mechanism. Furthermore, if virtual methods and call sites are present in our code, special analysis, such as *class hierarchy analysis* (CHA) or *whole program class analysis* will be required to determine concrete methods that are reachable from a given virtual call site (see e.g. [16], where an overview and extensive bibliography on these techniques is presented). It appears [16], that all of these techniques generally give a greater number of virtual methods than the program would actually call. In the case of simple techniques (CHA), overestimation may be as high as 10-20 times. More complex techniques, such as *points-to* analysis, which takes into account what classes are actually instantiated in the program and where, result in a higher precision. However, they would not work for objects allocated using reflection (or returned by native methods), whose exact types thus can not be determined by static analysis of Java code. Also, in our opinion, these techniques are unlikely to scale well with the growth of the analysed code (the tests used for evaluation in [16] seem to us unrealistically small).

Thus, if we can instrument methods in any class only once, before it has been loaded by the VM (which is the only option in most of the contemporary industrial Java VMs), precise (and exhaustive) call graph revelation for an arbitrary root method done in acceptable time seems highly problematic. That is probably why call subgraph instrumentation and profiling is not used in conventional profiling tools for Java.

However, as we show here, if a VM (or, generally, a runtime system) provides a way to instrument code at run time, a novel technique for call graph revelation and instrumentation can be implemented. In the following two subsections we briefly describe the two variants (schemes) of this technique. Both of them use simple conservative class hierarchy analysis (which simplifies the implementation a lot), and yet — at least the second scheme — demonstrate quite acceptable precision (see Section 7) on even highly-polymorphic tests.

6.1 Scheme A

In this scheme, we establish the following conditions under which a method `m()`is instrumented:

1. Class `C` to which `m()` belongs has been loaded by the VM.
2. Using class hierarchy analysis, `m()` is determined to be reachable directly or transitively from the root method.

The actual procedure works as follows:

1. The procedure starts when the subgraph root method's class is loaded (we can intercept this event using the standard class load hook mechanism of the JVM). Blocking further class loads temporarily, obtain the list of all classes currently loaded by the VM. For each class a special "mirror" data structure is created that will hold certain information relevant to our analysis. To speed up further analysis, for each class we can also determine and record in the mirror the names of all of its subclasses. Thus, for any given loaded class, all of its subclasses can be obtained instantly. After determining all loaded classes, install our own class load hook that performs operations described in step 3, and resume the TA.

   Then, for the root method m(), perform step 2.

2. Scan the bytecodes of method `m()`, examining each call instruction. If a call instruction is for a static method `SC.s()`, mark this method as "reachable unscanned static" in the `SC` class mirror. Then check if class `SC` is currently loaded by the VM. If it is, mark method `s()` as scanned and perform step 2 for it in turn.

   If a call instruction is for a virtual method `VC::v()`, mark this method as "reachable unscanned virtual". Then check if class `VC` is currently loaded by the VM. If it is, mark the method as scanned and perform step 2 for it. Next, check all of the subclasses of VC that are currently loaded by the VM. If in any of them `v()` is overridden, perform the same procedure for this method, i.e. mark it as scanned and repeat step 2 for it.

   When this step exhausts, we have a number of methods in loaded classes marked as "reachable scanned" and a number of methods in the mirrors for not yet loaded classes that are marked "reachable unscanned". Now, instrument each scanned method using method hotswapping mechanism, and mark them accordingly.

   The next step 3 is performed every time a new class is loaded by the VM.

3. Upon loading of a class `C`, check if a mirror data structure has already been created for this class, and if not, create one. Check if there are any reachable methods in the mirror. If yes, mark each of these methods scanned and perform step 2 for each of them.

   Next, determine all of `C`'s superclasses. For each superclass, check if it contains any reachable virtual methods. If so, check if any of these methods are overridden in `C`. Mark each such method as "reachable scanned virtual" and perform step 2 for it.

In our implementation, all bytecode scanning and other code analysis operations are performed at the tool side. The code at the target VM (server) side sends to the tool messages such as "class loaded", "initial list of classes loaded by the VM", etc., and receives messages containing lists of methods to instrument. It is worth noting that once any code has been instrumented and thus profiling started, further code analysis and instrumentation upon class loading, which may be relatively time-consuming, could have affected profiling results quite significantly. To prevent this, every time a class load event occurs, our server-side code records a special "thread suspend" profiling event. The "thread resume" event is recorded just before returning from the class load hook to the TA code. This compensates for the time spent in code analysis and instrumentation.

Note further that, although the VM may perform concurrent class loads, all class load, analysis and instrumentation operations in the JFluid back end are made serial. This is done by making the class load hook method, which effectively initiates these operations, synchronized on a lock. In this way, we guarantee that at any point our analysis operates on a fixed set of classes, and therefore no classes can get missed or instrumented twice.

This scheme works much better than static Class Hierarchy Analysis, since only the classes actually loaded by the VM are analysed, and their number is usually significantly smaller than the number of all classes on the class path. Furthermore, we can easily cope with method invocations via reflection, by instrumenting the code of the `java.lang.reflect.Method.invoke()` method itself. Every time `Method.invoke()` is called, we check to see if its argument method has already been instrumented. If not, we dynamically process and instrument it as yet another reachable method.

However, quite often this scheme may result in a large number of methods instrumented but never called (see Section 7). Thus, we eventually came up with another scheme, which uses a more lazy approach, presented in the next section.

6.2 Scheme B

To overcome the shortcomings of the previous scheme, we extend the set of conditions necessary for method `m()` to be instrumented. Two new conditions, in addition to the first two presented in the previous subsection, are added:

1. Method `m1()` which calls `m()` directly has been instrumented;
2. Method `m1()` is about to be executed for the first time.

The fact that a method is instrumented only when its chances to be called are good, since its direct caller has been called, is likely to reduce the number of unnecessarily instrumented methods quite significantly. However, to make this algorithm work, the instrumentation in every target application method that has callees, should check, every time it is invoked, whether this is the first invocation of the given method. If so, it triggers the procedure of instrumentation of its direct callees. Fortunately, for the most time-sensitive profiling kind, CPU profiling, this check can be implemented within the same injected `metodEntry()` method that performs CPU time measurements operations. The additional check takes very little time compared to the rest of the instrumentation code. Before instrumentation starts, we allocate a global array of booleans, called `methodInvoked[]`, where each element corresponds to a method in the profiled subgraph. During instrumentation, methods that are found reachable are assigned integer identifiers that grow monotonically. Thus each method is associated with an element of the above array, and to check if an invocation of the given method is the first one, it is sufficient to check if `methodInvoked[methodId] == false`.

The algorithm itself works as follows:

1. The procedure starts when the root method's class is loaded (we can intercept this event using a standard JVM class load hook). Obtain the list of all classes currently loaded by the VM and create mirror data structures in the same way as in Scheme A. Also create a growable global boolean array `methodInvoked[]` that indicates whether any instrumented method has been invoked at least once.
2. Instrument the root method `m()`, and mark it as instrumented in the `C`'s mirror.
3. This step is performed every time any instrumented method `m()`is entered. Check if this method is being executed for the first time. If this is true, perform step 4 for this method.
4. Scan the bytecodes of method `m()`, examining each "call" instruction. If a call is for a static method `SC::s()`, and method `s()` is not marked as instrumented, mark it as "reachable uninstrumented static" in the `SC` class mirror.

   If a call is for a virtual method `VC::v()`, and this method is not marked as instrumented, mark it as "reachable uninstrumented virtual" in the `VC` class mirror. Next, check all subclasses of `VC` that are currently loaded by the VM. If in any of them `v()` is overridden and not marked as instrumented, also mark it as "reachable uninstrumented virtual".

   When this step is exhausted, we have reached the end of `m()` and marked a number of methods directly reachable from it as "reachable uninstrumented". Now,

using the hotswapping mechanism, instrument those of the above methods that belong to classes which by this time have already been loaded by the VM.

5. This step is performed by the class load hook, upon loading of any class `C`. Check if a mirror has already been created for class `C`, and if not, create one. Next, determine all of the `C`'s superclasses. For each superclass, check if its mirror contains any reachable virtual methods. If so, check if any of these methods are overridden in `C`. Mark each such method in `C` as "reachable uninstrumented virtual".

   Finally, instrument all methods in C marked as "reachable uninstrumented" and mark them as instrumented.

In our tests, this scheme generally works much better, resulting in a close match between the number of instrumented and actually executed methods (see the next section). As an only drawback, for short-running applications it may result in a higher overhead due to a large number of `RedefineMethods()` calls, each of which carries just one or two instrumented methods.

7 Evaluation

This section presents experimental results obtained from our VM and tool executing several benchmarks. Our goal was to evaluate the efficiency and accuracy of our system, and to compare the two instrumentation schemes described in the previous section. As the first set of benchmarks (the "small" ones), we used the standard SPECjvm98 [18] suite, executed with 100%-sized inputs. To these, we added one non-standard benchmark, `Notepad`, which is a demonstration GUI program distributed with Sun's JDK.

As the second, "large" benchmark set, we used the Sun PetStore [20] sample application running on top of Sun™ ONE Application Server 7, Standard Edition. J2EE™ [19] and Web applications, together with the code of application servers on top of which they run, are probably the largest Java applications, that can be easily obtained at present. PetStore was chosen as representing a (hopefully) typical J2EE application that manages an online shop and supports selecting items from a list, adding them to the shopping cart, paying with a credit card, and creating/updating user accounts. With this application, we exercised two scenarios, resulting in two benchmarks, `PetStore 1` and `PetStore 2`. In the first scenario the "user" (our workload emulation tool) would just browse through the store, add a few items to the shopping cart, and leave. In the second scenario, the user would retrieve and then update their persistent account details.

In all tests except Notepad we did not profile Java core class methods to avoid results dependency on the JDK (core class internal code can change between JDK versions). For Notepad, core classes were included (they are mostly Swing graphics classes in this case) to demonstrate the spectacular difference between two profiling schemes in a "real life" situation.

All experiments were performed on the Sun E420ER server with four 450MHz UltraSPARC™ II processors, with 4GB of main memory, running the Solaris™ Operating Environment, version 2.8. We executed each small benchmark once to "warm up", and then three more times to gather results. The recorded results were calculated as an average for these three runs. In case of `PetStore`, we generated the workload using the OpenSTA tool [2]. Each testing sequence consisted of 10,000 consecutive recorded interactions, preceded by two warm-up sessions of 100 and 1000 interactions, respectively.

All measurement results are grouped into two identically structured tables. The results in Table 1 are for the whole program profiling — that is, each benchmark was instrumented transitively starting from its `main` method. In the second table, we present results of partial instrumentation of each of the benchmarks, performed in the following way. For each of the SPECjvm98 benchmarks, we picked up (essentially arbitrary) method `m()`, that, together with its callees, takes about 10% of the total charged execution time of the fully profiled benchmark. `m()`would then be used as a call subgraph root for measurements that are presented in Table 2. In `Notepad` a root method was one called in response to a user action ("New File"), and we discarded the results of the first invocation, when most or all of the method hotswapping operations are performed — as developers profiling their program would normally do. Finally, in `PetStore` we profiled the call graph originating from the business method `processEvent()` of the `ShoppingClientControllerEJB` class, which, as we observed, is effectively the single entrypoint into the "business" code of `PetStore`.

In each table, there are two lines for each benchmark: they correspond to results obtained using instrumentation scheme A and B (see previous section), respectively. The columns in the tables have the following meaning:

- *Calls* is the total number of calls to instrumented methods.
- *Bare time* refers to the time taken by the benchmark running without instrumentation. *Root time* in Table 2 is the time, measured using our lightweight code fragment profiling mechanism, for the method that is used as a call subgraph root in the partially profiled run.
- *Charged time* is the time charged by the profiler against the profiled application subset. An ideal profiler taking proper account of all kinds of overhead caused by

itself, should show charged time equal to bare (root) time. Thus, the difference between the bare and the charged time can be viewed as the profiler's accuracy. Note though, that some of the side effects of instrumentation are very hard to account for — this is discussed a few paragraphs below. For `PetStore`, charged time is not given when full profiling is done, since in that case we get results for many threads doing different jobs, and there is no sensible way to compare this data with partial profiling results.

- *Total time* is the total execution time for the instrumented benchmark.
- *Overhead* is the percentage increase of the total time with respect to the bare time.
- *Instrumentation, Hotswapping, Tool 1, Tool 2* are the relative (to the total execution time) times spent, respectively, in the injected instrumentation code (i.e. `methodEntry() and methodExit()` methods); in the VM performing method hotswapping operations; in the tool performing bytecode analysis and rewriting; and in the tool performing profiling data processing. The last two values also include time spent exchanging data between the system components. For `Notepad` in partial profiling mode, and `PetStore`, Hotswapping and Tool 1, times are not given, since the respective operations were performed during the warm-up execution(s).
- *Total instrumented methods* and *called instrumented methods* are the total number of methods that were instrumented, and the number of instrumented methods that were actually called at least once during execution.

Let us first consider the full profiling results. When profiled fully, most of the benchmarks make a large number of calls, which results in instrumentation consuming very significant time and the resulting high overhead (the maximum value of nearly 50 times is observed in `compress`). The cost of transferring the rough profiling data to the tool and processing it on that side ("Tool 2") is 2-3 times lower than the cost of the instrumentation, and, by coincidence or some deeper reason, is 18-19% for the majority of the benchmarks. Method analysing and hotswapping time is negligibly small in all of the tests except for `Notepad` (discussed a few paragraphs below).

The charged time is reasonably close to the bare time in `db`, is 1.5 - 2 times greater in `jack`, `javac` and `Notepad`, and is more than twice greater in the rest of the benchmarks. We verified the correctness of this data by additionally profiling the instrumented applications using a simple form of sampling. That is very easy and cheap to implement in JFluid, given that the thread status — whether it is currently in the instrumentation code or the application code — is stored in the internal record that JFluid creates for each profiled thread. Following the technique described in [29], we implemented a sampling thread as just an additional Java thread spinning in an endless loop. Inside that loop, the thread would repeatedly sleep (using `Thread.sleep()`) for a specified sampling interval, then wake up and inspect the records for active thread. The resulting data confirms our measurements.

There are likely to be three reasons for the fact that the charged time is still greater than the bare execution time, even after the injected code execution time is factored out. The first is inlining prevention: injected code makes the size of some application methods larger than the inlining threshold, and thus these methods are not inlined anymore and run slower. The second is cache miss effects: injected code is likely to cause CPU cache misses that do not happen during normal program execution. It is hard to measure this effect directly, but in general cache misses are known to cost a lot (see e.g. [11, 4]). Finally, running changed and unchanged code interpreted after hotswapping, recompiling it, then deoptimizing it again and again may also cause substantial slowdown. The first two of the above factors apparently dominate in the benchmarks where a very large number of calls to very short-running (on the order of 0.1 microseconds) methods is made: `compress`, `jess` and `mpegaudio`. `javac` probably suffers from all three problems, whereas in `Notepad` the cost of repeated recompilations is likely to dominate.

Running benchmarks with partial profiling leads to a very considerable reduction in the overhead. However, for small, computation-intensive benchmarks, there is still a significant mismatch between the real and the charged time. Only `db` and `Notepad` with their relatively small number of calls practically don't suffer from that. We conclude that measuring exact execution time using total, or even partial (in our case, 10-50% of the total number of methods) instrumentation-based profiling is not suitable for small, computationally intensive applications running on modern highly optimized VMs and processors. Apparently, relatively heavyweight instrumentation that we employ breaks too many optimizations on various levels when it is injected into methods that, in the compiled form, consist of just a handful of instructions. On the other hand, lightweight instrumentation that we also have in our tool produces much more realistic results.

The situation is quite different for large benchmarks, for which JFluid imposes modest or very reasonable overhead despite a very large number of calls when doing partial profiling. The charged time is still greater than the root time, but not very substantially. We conclude that using instrumentation and partial profiling really pays off in large ap-

Table 1: Results for whole program instrumentation

Benchmark/ scheme	Calls	Bare time, ms	Instrumented time, ms charged	total	ovhd,%	Instr., %	H/S, %	Tool 1, %	Tool 2, %	Instr. methods total	called
compress	225.93M	23241	73038	1199465	5061	75	< 0.1	< 0.1	18	55	53
			73079	1199498	5061	75	< 0.1	< 0.1	18	55	53
db	70320	24714	25174	25838	4.5	1.1	< 0.1	< 0.1	< 0.1	52	46
			25219	25912	4.8	1.1	< 0.1	< 0.1	< 0.1	52	46
jack	1.86M	7756	10128	20630	165	36	3.3	4.2	9.2	274	264
			10211	20718	166	36	3.8	4.2	9.2	274	264
javac	46.7M	19575	33956	275625	1308	67	1	0.9	19	992	730
			34270	277399	1317	67	1.8	0.8	19	936	730
jess	77.62M	8965	27553	416021	4640	72	< 0.1	0.2	19	478	327
			27417	415044	4530	72	< 0.1	0.4	19	442	327
mpegaudio	91.7M	18204	41083	504528	2672	73	< 0.1	0.2	19	241	213
			40372	503576	2666	73	< 0.1	0.2	19	230	213
Notepad	144256	1373	3134	18589	1254	3.1	58	44	2.4	7486	2874
	126067		2348	24775	1704	2.0	55	36	1.8	3853	2706
PetStore 1	6.01M	3921025	N/A	4395469	12.1	1.2	N/A	N/A	1.5	5345	287
			N/A	4364105	11.3	1.0	N/A	N/A	1.5	979	213
PetStore 2	178M	9630732	N/A	12635520	31.2	8.7	N/A	N/A	5.3	8734	1124
			N/A	12568103	30.5	8.5	N/A	N/A	5.2	2691	1073

plications, where "call intensity" is generally not very high, and where it really makes sense for developers to instrument a small (maybe just a few per cent) proportion of application code, because even this amount of code may, in absolute terms, be quite large (a few hundred methods in our case) and worth optimizing. Furthermore, J2EE and Web applications may interact with the underlying Application Servers in complex ways, so it makes a lot of sense to profile code that belongs to different "code layers" together in such cases. Our technique, which automatically instruments methods based on the logical task that they perform, rather than, say, on their affiliation with one or another package (layer), directly addresses this issue.

Now consider the number of methods that our system instruments. In benchmarks that do not include many polymorphic calls, such as `compress` or `db`, the total number of instrumented methods and the number of called methods are equal or very close, and is the same or close for both call graph revelation schemes. This changes dramatically in `javac`, `Notepad`, and PetStore, where scheme A causes instrumentation of many more methods.[4] To our surprise, in the case of PetStore, even the "lazy" instrumentation scheme B results in a large number of methods instrumented but not called.

Instrumenting more methods than necessary causes two kinds of problems. One is that a method that is instrumented unnecessarily can still be called by the TA. Since the context for the call is wrong, that is, the user-selected root method is not on the stack, the instrumentation code does not do any useful work. However, it still consumes time (although it is smaller than when it is invoked in the correct context). These unnecessary invocations can contribute to the total execution time, as it happens with `mpegaudio` when scheme A is used.

Another issue is that instrumenting more methods means that more time is spent in hotswapping, and more methods get deoptimized, run interpreted, then recompiled, etc. The difference in the hotswapping (and also code analysis) time is what causes such a big increase in execution time when scheme A is used for `Notepad`. Although in the long term these effects get amortised, they may make a big difference if e.g. program startup is profiled. We conclude that the "lazy" scheme B is generally much more suitable for performance profiling applications that we consider appropriate for instrumentation-based profiling.

[4] The difference in the number of calls and called methods for `Notepad` and `PetStore` with different schemes is explained by the fact that in these benchmarks we instrument Java core and Application Server classes, respectively, which call native methods, which, in turn, can call Java code. Our code analysis schemes cannot follow the latter kind of calls; however if a method that has been instrumented for other reasons is called by a native method, JFluid registers this call. Since scheme A causes a very large number of methods to be instrumented, some of them may, by coincidence, be called by the native code. This is a shortcoming of our technique, which is not yet addressed.

Table 2: Results for partial program instrumentation

Benchmark/ scheme	Calls	Root time, ms	Instrumented time, ms charged	total	ovhd,%	Instr., %	H/S, %	Tool 1, %	Tool 2, %	Instr. methods total	called
compress	104.3M	6579	20290	574411	2285	73	< 0.1	0.1	18	7	7
			20861	573393	2281	73	< 0.1	< 0.1	18	7	7
db	569	2612	2620	25328	1.2	< 0.1	< 0.1	0.3	< 0.1	2	2
			2658	25411	1.5	< 0.1	< 0.1	0.3	< 0.1	2	2
jack	242267	273	443	29140	271	3.3	0.1	2.6	1.8	66	55
			458	28785	267	3.4	0.3	1.2	2.0	65	55
javac	2.70M	850	1575	44893	118	24	0.9	1.3	6.8	587	209
			1756	38901	91	28	1.3	1.9	7.8	289	209
jess	5.67M	357	1213	47471	403	48	0.2	0.5	12	33	4
			1179	46056	388	49	< 0.1	0.5	12	8	4
mpegaudio	8.96M	612	1844	80667	343	44	0.1	0.2	11	191	11
			1762	70901	289	50	< 0.1	0.2	12	21	11
Notepad	660	49.7	59.4	62.1	25	4.2	N/A	N/A	< 0.1	7882	31
	380		54.5	56.0	12	2.7	N/A	N/A	< 0.1	76	27
PetStore 1	3.93M	12979	17071	4012785	2.1	< 0.1	N/A	N/A	0.2	1231	128
			16858	4010253	2.0	< 0.1	N/A	N/A	0.2	439	125
PetStore 2	170.4M	1313700	1672218	10799213	11.9	4.5	N/A	N/A	4.2	8606	976
			1613429	10771041	11.8	4.4	N/A	N/A	4.2	2546	891

8 Related Work

BEA Systems' JRockit JVM [6] is probably the closest equivalent of JFluid. This JVM supports dynamic method instrumentation, and allows the user to select an arbitrary number of methods to instrument using the GUI tool called Management Console. Methods can only be picked up by hand, i.e. no equivalent of our call subgraph profiling is available. The VM has to be started in a special mode to enable dynamic method profiling.

In the SELF system [27, 11, 26] an instrumentation-based profiling technique was once used, that featured eager injection of instrumentation code into the TA, done by the source-to-bytecode compiler. Initially, however, the instrumentation library routines would be deliberately made empty, allowing the dynamic compiler to optimize away the calls to them, thus eliminating any performance impact. Dynamic activation of the instrumentation was done by hotswapping the empty library routines with the "real" ones. At this point, all the TA code would be deoptimized, and then gradually recompiled again, now with calls to the instrumentation library.

ParaDyn/DynInst (see the [28] Web site, where documentation and published papers can be found), is a suite of tools and APIs that allow developers to profile and monitor running parallel and distributed programs using dynamic insertion and customization of measurement code. It operates on native machine code programs and supports a variety of platforms, including SPARC/Solaris, Windows and Linux on x86 processors, and AIX on RS6000. ParaDyn can insert and remove instrumentation adaptively, and has support for automatically locating bottlenecks in the program.

The IBM Jinsight tool [14, 17] follows some of the same principles as JFluid. Performance data in this tool is gathered in *bursts*, defined as "a set of trace execution information, gathered during an interval of time, associated with a specific task in program". A burst is triggered by a user-specified method, much the same as profiling data collection in JFluid is triggered by some thread entering the root method. Jinsight also filters the data based on the user-specified threads, methods and other criteria. The implementation, however, is based on a special JVM which emits all events, such as method entry or object allocation, unconditionally, with the tool just filtering events that are relevant for the current task. The fact that the JVM always emits all events (or at least all events in one category) is likely to affect performance adversely.

The idea of accumulating and presenting data in the form of the calling context tree (CCT), which is both a very useful presentation format, and the data accumulation form that allows us to calculate and factor out the profiling overhead correctly for each method, was first introduced by Larus et al. in [3]. The same group has published a series of papers on different forms of profiling, but most of this work

concerns measurements at a granularity finer than a single method, so that the path of control flow within subroutines is recorded.

9 Conclusions and Future Work

Dynamic bytecode instrumentation mechanism that we implemented in the Java HotSpot VM, described in this paper, allows developers to instrument and profile a subset of a running application on-the-fly, and change this subset as many times as needed. Our VM-internal method hotswapping functionality should allow for any kind of instrumentation. However, in our profiling tool, called JFluid, we have so far concentrated mostly on a limited task — profiling a call subgraph in a running Java application. We believe that call subgraph is a very useful way of choosing a group of methods to profile, since, for one thing, it makes the tool instrument and profile a set of methods that is exactly the same as the one that the user typically wants to observe. It appears that in presence of virtual methods it is hard to precisely identify all methods in a call subgraph: it may be quite expensive and will not work in the presence of some Java mechanisms, such as Java Reflection. Therefore, we introduced a technique, where the tool reveals a call subgraph dynamically, while the application runs. We suggest two alternative schemes for doing that, and show experimentally that the second one generally gives better results.

We have shown that profiling a small part of the target application can significantly reduce the temporal overhead usually associated with instrumentation-based profiling. For CPU profiling, we found that absolute precision of measurements, even when instrumentation cost is determined and factored out, is not very high for computationally intensive benchmarks that make a large number of short-running calls. However, we also found that for large programs, such as J2EE applications running on top of an Application Server, the situation is very different, and the benefits of our technique are much more apparent. We also believe that on hardware/VM that do not perform as many optimizations as those (UltraSparc and HotSpot JVM) that we currently use, our technique may result in a much smaller perturbation to the results. Finally, dynamic bytecode instrumentation can be used in a very limited way: to trigger and stop sampling-based profiling, which in that case (when profiled code subset is relatively small) may be done with higher resolution and precision.

CPU profiling is not the only useful thing that can be done using dynamic bytecode instrumentation. Presently our tool also supports object allocation profiling (not described in this paper), which is done, in essence, by dynamically injecting calls to instrumentation at each allocation site. As a further continuation of this line of work, we consider object liveness analysis and, possibly, memory leak debugging, done in a relatively unobtrusive way, i.e. without interrupting the target application. We hope that "self-injecting" instrumentation techniques similar to the one that we use for call graph revelation can help us in that.

In a more distant future, it may be worth thinking about how to further reduce the profiling overhead and/or make it more controllable, to ultimately enable in-the-field use of our technology. Such an "emergency servicing" capability in a VM may be very attractive to developers and system administrators, who currently suffer from bugs in deployed applications, that may be hard or impossible to reproduce in the lab environment. We can imagine discovering a bottleneck or another problem in a running application, while keeping the overhead below some predefined (low) level at every moment, by gradually narrowing down the observation area and dynamically increasing the level of detail in the generated tracing data.

10 Acknowledgements

The author is indebted to Adrian Mos for his help with setting up and measuring the performance of large Java applications presented in this work, and to Mario Wolczko, for reading the early draft of the paper and giving very helpful comments. We are also grateful to numerous JFluid users inside and outside Sun Microsystems who sent us their feedback, that helped to better understand the real user requirements and concerns, and improve the quality of our tool.

References

[1] JSR 163 — Java™ Platform Profiling Architecture. http://www.jcp.org/jsr/detail/163.jsp, 2002.

[2] Open System Testing Architecture (OpenSTA). http://www.opensta.org, 2003.

[3] G. Ammons, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, pages 85–96. ACM Press, 1997.

[4] L. Bak, U. Hölzle, and D. Ungar. A Source-Level Profiler for Optimized Object-Oriented Programs. Draft, unpublished, 1992.

[5] T. Ball and J. Larus. Optimally profiling and tracing programs. In *ACM Transactions on Programming Languages and Systems*, volume 16, pages 1319–1360, 1994.

[6] BEA Systems. JRockit 8.1 SDK User Guide. http://edocs.bea.com/wljrockit/docs81/userguide/index.html, 2003.

[7] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1, March 1999.

[8] Borland Software Corporation. OptimizeIt Suite. http://www.borland.com/optimizeit, 2002.

[9] M. Dmitriev. Application of the HotSwap Technology to Advanced Profiling. In *Proceedings of the First Workshop on Unanticipated Software Evolution, held at ECOOP 2002 International Conference*, Malaga, Spain, 2001. Available at http://www.dcs.gla.ac.uk/∼misha/papers.

[10] M. Dmitriev. *Safe Evolution of Large and Long-Lived Java Applications*. PhD thesis, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, 2001. Available at http://www.dcs.gla.ac.uk/∼misha/papers.

[11] U. Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University, March 1995. Also published as Sun Microsystems Laboratories Technical Report SMLI TR-95-35.

[12] Intel Inc. Intel VTune Performance Analyzers. http://www.intel.com/software/products/vtune/, 2003.

[13] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[14] W. De Pauw, N. Mitchell, M. Robillard, G. Sevitsky, and H. Srinivasan. Drive-by Analysis of Running Programs. In *Proceedings for Workshop on Software Visualization, International Conference on Software Engineering*, May 2001.

[15] Quest Software. JProbe Java Profiling and Memory Debugging. http://java.quest.com/jprobe/jprobe.shtml, 2003.

[16] A. Rountev, A. Milanova, and B. Ryder. Fragment Class Analysis for Testing of Polymorphism in Java Software. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*. IEEE Computer Society Press, 2003.

[17] G. Sevitsky, W.De Pauw, and R. Konuru. An Information Exploration Tool for Performance Analysis of Java Programs. In *Proceedings of TOOLS Europe 2001 Conference*, 2001.

[18] Standard Performance Evaluation Corporation. SPEC Java Virtual Machine Benchmark Suite. http://www.spec.org/osg/jvm98, August 1998.

[19] Sun Microsystems Inc. Java 2 Platform, Enterprise Edition - Documentation. http://java.sun.com/j2ee/docs.html.

[20] Sun Microsystems Inc. Java Blueprints. http://java.sun.com/blueprints/code.

[21] Sun Microsystems Inc. Java Virtual Machine Debug Interface Reference. http://java.sun.com/products/jpda/doc/jvmdi-spec.html.

[22] Sun Microsystems Inc. Java Virtual Machine Wire Protocol Reference. http://java.sun.com/products/jpda/doc/jdwp-spec.html, 2000.

[23] Sun Microsystems Inc. Java HotSpot™ Technology. http://java.sun.com/products/hotspot, 2002.

[24] Sun Microsystems, Inc. Emerging technologies: jvmstat. http://developers.sun.com/dev/coolstuff/jvmstat, 2003.

[25] Wily Technology. Introscope 3.0. http://wilytechnologies.com/solutions_introscope.html.

[26] D. Ungar. Private communication.

[27] D. Ungar and R. Smith. SELF: The Power of Simplicity. In *OOPSLA'87 Conference Proceedings*, volume 22 of *ACM SIGPLAN Notices*, pages 227 – 241, Orlando, Florida, December 1987.

[28] University of Wisconsin, Madison. ParaDyn Project Home Page. http://www.cs.wisc.edu/ paradyn/, 2003.

[29] J. Whaley. A Portable Sampling-Based Profiler for Java Virtual Machines. In *Proc. of the Java Grande 2000 Intl. Conference*, pages 78–87. ACM Press, 2000.

About the author

Mikhail Dmitriev received his Diploma in Computer Science and Ph.D. in Computer Science from the University of Cambridge and the University of Glasgow, UK, respectively. The projects in which he was involved through his Ph.D. term and subsequent career at Sun Labs were: PJama (a persistent platform, effectively a “transparent” object database system, for Java), HotSwap (run time evolution of Java applications), and Javamake (a dependency checking, or smart recompilation, tool for Java). Mikhail is presently a Principal Investigator for the JFluid project, that investigates how run time code hotswapping techniques can be used to build smart, unobtrusive mechanisms for run time introspection of Java applications.