xdProf: A tool for the capture and analysis of stack traces in a distributed Java system

John Lambert[a] and Andy Podgurski[a]

[a]Electrical Engineering and Computer Science Department,
Case Western Reserve University, Cleveland, Ohio 44106

ABSTRACT

We describe the design and implementation of xdProf: a tool that captures and analyzes stack traces sent at a fixed interval from Java Virtual Machines in a distributed system. The xdProf client uses the Java Virtual Machine Profiling Interface and works with any compliant implementation; no access to application source code is necessary, no library modifications are needed, and there is no run-time instrumentation of Java byte code. Configuration options given at virtual machine startup specify the interval for stack trace transmission and the remote xdProf server. The xdProf server collects information from multiple xdProf clients and provides an extensible interface for analysis. Current tools include a graphical user interface for viewing the most recent stack traces from multiple virtual machines and the generation of control flow graphs for each virtual machine. The performance impact of the xdProf client sending data over a local area network is minimal: less than a 8% increase in total elapsed time for a set of standard benchmarks. Future plans include real-time visualization, reliability estimation, trace capture, and performance analysis.

Keywords: Java profiling, distributed profiling, program visualization, Java Virtual Machine Profiling Interface

1. MOTIVATION

Java programs are becoming increasingly complex: they are executed on different hardware platforms and in different physical locations. Examining a local program during runtime is difficult; examining multiple programs running on different virtual machines, in different physical locations is even more difficult. Our goal was to create a tool that would send stack traces of a Java Virtual Machine to a remote location at a fixed interval. This stack trace data could be used in several different ways: interactive developer observation of the entire system, visualization of control flow in the system, or collection of the data for future analysis. This tool would have to run on different operating systems, different architectures, and different Java virtual machines, with minimal performance impact.

2. XDPROF CLIENT

The xdProf client runs on a Java Virtual Machine (JVM) and sends stack traces to a remote machine at fixed interval. It is a small dynamic link library (`xdProf.dll` or `libxdprof.so`) which can be used with the IBM Java Development Kit 1.3 and the Sun Java Development Kit 1.2 and 1.3; it is currently available on the Intel/Win32 platform, Intel/Linux, and Ultra/Solaris platforms. The xdProf client is invoked with the `-Xrun` command-line option:

`java -XrunxdProf:server=`*machine_name*`,port=`*port_number*`,refresh=`*milliseconds* *ApplicationToRun*

The Java applet viewer (`appletviewer`) and Remote Method Invocation Registry (`rmiregistry`) can use the xdProf client when the `-J-XrunxdProf:...` command-line argument is used. Also, on some Java Virtual Machines, the environment variable `_JAVA_OPTIONS` can be set to the `-XrunxdProf:...` argument so all programs running on the Java virtual machine will automatically use the xdProf client.

Further author information:
J.L.: E-mail: `jlambert@jlambert.com`
A.P.: E-mail: `andy@eecs.cwru.edu`

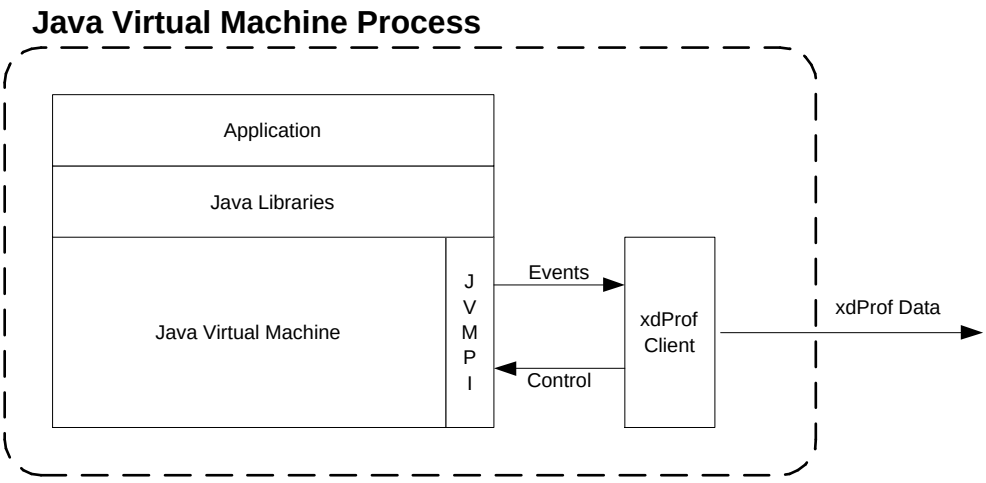

Figure 1. xdProf client architecture.

2.1. Java Virtual Machine Profiling Interface

The xdProf client uses the Java Virtual Machine Profiling Interface (JVMPI), which was proposed as a "general-purpose and portable mechanism for obtaining comprehensive profiling data from the Java virtual machine... it is extensible, non-intrusive, and powerful enough to suit the needs of different profilers and virtual machine implementations."[1] Currently, both IBM and Sun support the JVMPI specification on various platforms: Windows, Linux, Solaris, Macintosh OS X, etc. The JVMPI eliminates the need for an instrumented Java Virtual Machine, and allows one profiler to work with many different virtual machines. In the current version of the JVMPI, only one profiler agent per virtual machine can be supported. The JVMPI sends events to a profiler that has registered its interest in specific events via JVM callbacks. xdProf uses seven of these events, listed in Table 1. The details for each event are available in Ref. 2.

xdProf will store or remove information about the thread or class upon receiving the appropriate JVMPI event. Once the JVM initialization done event is received, xdProf will create a background thread, described in Sect. 2.2, to communicate with the xdProf server and notify this thread when the JVM shutdown event is received.

xdProf also enables the object allocation event for a short period of time. The JVM initialization thread will allocate objects before a thread start event has been sent for it. xdProf uses the object allocation event to discover this thread, and request a thread start event for it via the JVMPI. After xdProf has retrieved all applicable information about the thread, the object allocation event will be disabled, increasing performance.

Table 1. JVMPI events used by xdProf.

Event Name	Description	Information used by xdProf
`THREAD_START`	A thread is started in the VM	Thread name, group and parent name, thread identifier
`THREAD_END`	A thread ends in the VM	Thread identifier
`CLASS_LOAD`	A class is loaded in the VM	Class name and identifier, source file, methods in the class
`CLASS_UNLOAD`	A class is unloaded in the VM	Class identifier
`JVM_INIT_DONE`	VM initialization is done	None
`JVM_SHUT_DOWN`	VM is shutting down	None
`OBJECT_ALLOC`	An object is allocated	Object identifier, class identifier, thread identifier

```
Connect to server; if not, disable all events and end this thread
While there is not a shutdown pending
      Wait the user-specified delay using a JVMPI monitor
      Disable garbage collection
      Suspend all running threads in the virtual machine
      Gather information about the thread call stacks, methods, and classes
      Resume the threads that were suspended by xdProf
      Enable garbage collection
      Send information to remote server
Disconnect from the xdProf server
```

Figure 2. Algorithm for communication thread.

2.2. Communication Thread

Once xdProf has been notified that the JVM is initialized, xdProf starts a background communication thread (Figure 2). This thread initializes communications by attempting to connect to the xdProf server; if it cannot connect to the server specified in the command-line arguments, it will disable all future event notification, effectively unloading xdProf. Every *delay* milliseconds, xdProf will disable garbage collection and suspend all threads in the system; both these steps are necessary: garbage collection must be disabled so it does not start while call stacks are being accessed and running threads must be suspended so they do not change their call stacks while information is being collected. Information about the threads, methods, and classes is stored in the data format presented in Sect. 2.3. Suspended threads are resumed and the information is transmitted. Once xdProf has been notified that there is a virtual machine shutdown pending, the communication thread will close its socket to the server.

This "sampling" algorithm is based on the statistical CPU sampling algorithm used by the HPROF profiler,[1] except that xdProf does not attempt to calculate or assign costs to the contents of the call stack. Like HPROF, this algorithm is independent of the number of processors and should work equally well on single processor and multiprocessor machines.

2.3. xdProf Data Format

The xdProf client sends plain-text ASCII data to the machine specified via command-line arguments. The data format, shown in Figure 3, is stateless: any information necessary to determine what is running in the VM is sent with the data; no history is assumed. Thread, class, and method identifiers are acquired from the JVMPI events indicating their creation, and are transmitted as eight-digit hexadecimal numbers.

The thread name, group name, and parent name are the values returned by the JVMPI when the thread event was received; changes at runtime (via `java.lang.Thread.setName`) are not visible. The logical start specified for each thread is a monotonically increasing integer corresponding to the order in which threads were started: the first thread started is assigned 1, the second thread is assigned 2, etc. Threads are sent in no particular order; however, the logical start value provides a way to order them by starting time and to determine a possible parent relationship (the child's parent name is the same as the possible parent's name, and the parent's logical start value is less than the child's logical start value). A gap in the sequence of logical start values indicates that the missing thread has terminated. Thread status is an integer indicating if the thread is runnable, waiting on a monitor, or waiting on a condition variable; if a thread is interrupted or suspended in any of these three states, a flag bit will be set.

After sending information about a thread, xdProf sends the number of frames in the call stack, and then the content of the call stack as a list of stack frames. Each stack frame consists of a method identifier and a line number. Line numbers will reference a line in the class source file or indicate a compiled method, a native method, or an unknown line number. The top of the call stack, the method currently executing, is sent first; the thread entry point is sent last.

After the number of methods is sent, xdProf will send the class identifier, method identifier, method name, and a method descriptor* for each method, in no particular order. To reduce network traffic, xdProf sends method

*The method descriptor describes the data types of the parameters and return data type in a concise format: "`Object mymethod(int i, double d, Thread t)`" has the method descriptor "`(IDLjava/lang/Thread;)Ljava/lang/Object;`".[3]

```
<VM process id> <command-line description>
<virtual machine, runtime, and operating system information>
<N: number of threads>
<thread 1 identifier>
<thread 1 name>
<thread 1 group name>
<thread 1 parent name>
<thread 1 logical start>
<thread 1 status>
<F: number of frames for thread 1>
<frame F method identifier> <frame F line number>
<frame F - 1 method identifier> <frame F - 1 line number>
...
<frame 1 method identifier> <frame 1 line number>
...other thread blocks...
<M: number of methods>
<method 1 class identifier> <method 1 identifier> <method 1 name> <method 1 descriptor>
...
<method M class identifier> <method M identifier> <method M name> <method M descriptor>
<C: number of classes>
<class 1 identifier> <class 1 name> <class 1 source file>
...
<class C identifier> <class C name> <class C source file>
```

Figure 3. xdProf data format.

information only for those methods that currently appear in the call stack. Class information is sent last and, to reduce network traffic, only classes with one or more methods in the call stack are sent. Inner and anonymous classes are sent with names as they are internally represented: `package.name.Outer$Inner`, `SomeClass$1`, etc. The xdProf client does not use all information accessible for classes: for example, names and data types of static fields and instance fields are omitted because transmitting this information would require significantly more network traffic.

2.4. Performance

The xdProf client introduces two distinct kinds of overhead with respect to application execution time. First, xdProf must process certain events which happen mainly near the beginning of a program's execution: loading of classes (such as the Java library classes), the starting of threads, and the completion of VM initialization. The second source of overhead is that xdProf must stop every running thread every *delay* milliseconds, generate a call stack, look up applicable information, and send the data to the server.

2.4.1. SPECjvm98

We used the SPECjvm98 benchmark[4] to evaluate the effect of the xdProf client. The SPECjvm98 is a standard measure of the performance of a Java virtual machine and the underlying hardware. Several different tests are run and the elapsed time for each test is used to calculate a SPEC ratio with respect to a reference system. The SPECjvm98 and SPECjvm98_base metrics are the geometric means of the best and worst ratios for each test, weighted equally. Benchmarking is performed inside a web browser or Java applet viewer: the system under test will download the test classes from a remote web server and run them.

Table 2 contains the SPECjvm98 and SPECjvm98_base results for the Sun Java 2 Runtime Environment 1.3 with HotSpot Client VM; we used an Intel Pentium II 350 MHz computer with 512 MB of RAM running Windows 2000 Professional for our testing.[†] xdProf was either not used at all, used to send data to the local machine every 100 milliseconds, or used to send data to a remote machine (different from the web server) every 100 milliseconds. Since we were interested in the performance effect of the client, we used a native code program[5] to receive data from the client, but did not process or analyze the data.

[†]The test ratios used to determine these values and the complete system environment are listed in Appendix A.

Table 2. SPECjvm98 results for Sun Java 2 Runtime Environment 1.3 with HotSpot Client VM.

	SPECjvm98	**SPECjvm98_base**
Without xdProf	22.0	18.3
With xdProf, local machine, 100 millisecond updates	20.8	18.6
With xdProf, remote, 100 millisecond updates	21.4	17.4

The SPECjvm98 value without xdProf is between 2.8% and 5.8% higher than with xdProf; the SPECjvm98_base value is 1.6% higher if xdProf sends data locally instead of not sending data, and not sending data at all is 5.2% faster than sending data remotely. However, since the SPECjvm98 benchmark runs as an applet and measures the elapsed time for test execution, it does not factor the time for Java VM initialization and shutdown.

2.4.2. Total elapsed time

To measure the overall effect of xdProf, we measured the total elapsed time from JVM start to JVM shutdown of an application that ran each SPECjvm98 test exactly twice; this includes the "up-front" overhead such as loading library classes, etc. We tested against Sun's HotSpot Client and Classic (no just-in-time compilation; all byte code is interpreted) virtual machines and no changes were made to default garbage collection behavior.

Table 3 shows that the overhead of xdProf on the HotSpot VM is between 1.93% and 7.76%, and that, as the delay between messages increases, the overhead generally decreases. Table 4 is interesting because the overhead of xdProf sending information locally is significantly higher than the overhead to send data remotely. We believe that this is because more context switches are required for the Classic VM to send and receive data locally but more investigation is necessary. Preliminary performance measurements on the IBM 1.3 Classic VM are consistent with the Sun Classic VM performance results.

2.4.3. Network traffic

As shown in Appendix A, xdProf sends approximately 4300 bytes per stack trace with the Classic VM and 2800 bytes per stack trace with the HotSpot VM, excluding TCP/IP transmission overhead. Network traffic is highly-application dependent: Sun's Forte for Java, Community Edition version 1.0[6] sent approximately 7100 bytes per trace when the Classic VM was used, and around 4300 bytes per trace when the HotSpot VM was used. The difference is due to the fact that the HotSpot VM will only report call stacks for threads that are not blocked so fewer stack frames, classes, and methods are sent. Overall, the Classic VM generated 50% more network traffic per stack trace. However, since execution of the elapsed time benchmarks took longer under the Classic VM, more stack traces were sent, and the Classic VM generated a total of fourteen to sixteen times the network traffic of the HotSpot VM.

Table 3. Elapsed time on Sun 1.3 HotSpot Client VM (without xdProf = 383.136 seconds).

Refresh (milliseconds)	**Local time (seconds)**	**Local overhead**	**Remote time (seconds)**	**Remote overhead**
100	412.863	7.76%	395.258	3.16%
200	402.338	5.01%	390.531	1.93%
1000	398.202	3.93%	392.143	2.35%

Table 4. Elapsed time on Sun 1.3 Classic VM (without xdProf = 3600.477 seconds).

Refresh (milliseconds)	**Local time (seconds)**	**Local overhead**	**Remote time (seconds)**	**Remote overhead**
100	4352.799	20.90%	3680.101	2.21%
200	4263.951	18.43%	3639.373	1.08%
1000	4207.349	16.86%	3601.698	0.03%

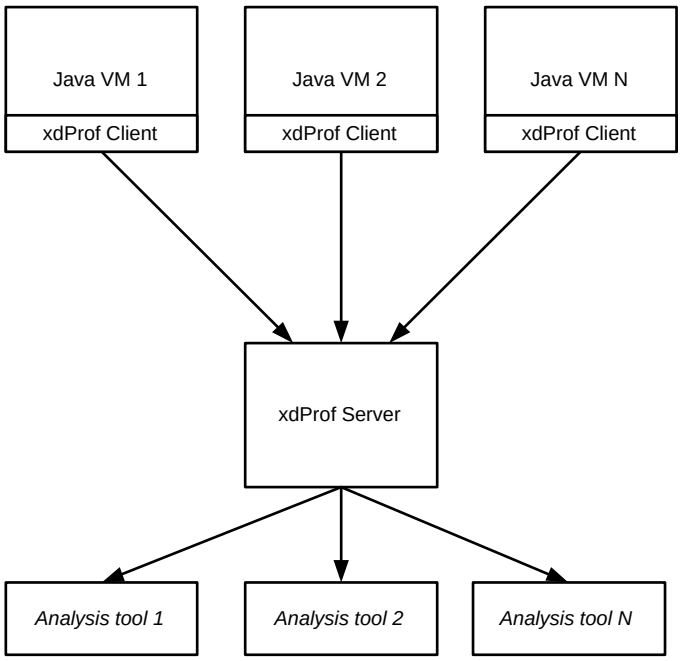

Figure 4. xdProf server architecture.

3. XDPROF SERVER

The xdProf server receives and analyzes information from multiple xdProf clients. It supports the use of multiple analysis tools, which can be added or removed at runtime. The xdProf server and analysis tools are written in Java; it would be possible to write equivalent server programs in other languages or to use Java Remote Method Invocation to off-load analysis to another, more powerful machine.

3.1. Interfaces

Each analysis tool in the xdProf server is notified when an event happens: an xdProf client connects to the xdProf Server, a trace is received for a connection, or a client disconnects from the server. There are three functions in the `ServerListener` interface, listed in Table 5. A `Connection` object stores information about the client; a `Trace` object contains information about the classes, methods, and threads loaded, as well as a time stamp. The same analysis object is used for all connections; this allows the tool to examine global patterns.

3.2. Analysis Tools

We will describe two analysis tools that currently exist; they are written in Java and can be run simultaneously or separately.

3.2.1. GUI tool

The GUI tool in Figure 5 displays the xdProf clients currently connected to the server, the threads running in a selected client, and the stack trace of a selected thread. There is a `Pause` button than will "lock" the GUI and allow the user to examine the stack traces of all the virtual machines at a specific point in time. (Since traces are received

Table 5. xdProf analysis tool interface

`public void addConnection(Connection c);`	Called once when a new Connection arrives.
`public void addTrace(Connection c, Trace t);`	Called when a trace is received for a Connection.
`public void removeConnection(Connection c);`	Called when a Connection disconnects.

xdProf Server

PID	Description	Address	VM	OS	Port
1489	none	motherboard/129.22.244.57	HotSpot(TM) Client VM 1.3.0_02 interpreted mode, Sun	SunOS 5.8 sparc	1337
1708	*forte*	*electronic/129.22.251.240*	*Classic VM 1.3.0-C native threads, nojit, Sun*	*Windows 2000 5.00 x86*	*1337*

Order	ID	Name	Parent	Group	#frames	Status
2	06A840B8	Signal dispatcher	none	system	0	Runnable
3	06A86AA8	Reference Handler	none	system	3	Condition Variable Wait
4	06A8A860	Finalizer	none	system	4	Condition Variable Wait
5	*002348F8*	*Main*	*system*	*main*	*24*	*Runnable*
6	06AF5108	DPROF Background Thread	none	system	0	Runnable
7	07433F48	AWT-EventQueue-0	system	main	6	Condition Variable Wait
8	07432D00	SunToolkit.PostEventQueue-0	system	main	3	Condition Variable Wait
9	07434410	AWT Windows	system	main	3	Runnable
11	0C12CFF0	OpenIDE Request Processor-0	null	system	2	Condition Variable Wait

Method	Class	Line
int indexOf(int, int)	java.lang.String	1195
void <init>(java.net.URL, java.lang.String, java.net.URLStream...	java.net.URL	485
void <init>(java.net.URL, java.lang.String)	java.net.URL	376
sun.misc.Resource getResource(java.lang.String, boolean)	sun.misc.URLClassPath$JarLoader	510
sun.misc.Resource getResource(java.lang.String, boolean)	sun.misc.URLClassPath	134
java.lang.Object run()	java.net.URLClassLoader$1	192
java.lang.Object doPrivileged(java.security.PrivilegedExceptio...	java.security.AccessController	Native
java.lang.Class findClass(java.lang.String)	java.net.URLClassLoader	188
java.lang.Class loadClass(java.lang.String, boolean)	java.lang.ClassLoader	297
java.lang.Class loadClass(java.lang.String, boolean)	sun.misc.Launcher$AppClassLoader	286
java.lang.Class loadClass(java.lang.String)	java.lang.ClassLoader	253
java.lang.Class loadClassInternal(java.lang.String)	java.lang.ClassLoader	313
void initialize()	org.openide.util.actions.CookieAction	73
void initialize()	org.netbeans.modules.text.ConvertToTextAction	34
java.util.Map getMap(org.openide.util.SharedClassObject)	org.openide.util.SharedClassObject$DataEntry	396
java.lang.Object getProperty(java.lang.Object)	org.openide.util.SharedClassObject	181
void addPropertyChangeListener(java.beans.PropertyChangeListen...	org.openide.util.SharedClassObject	205
void add(org.openide.modules.ManifestSection$ActionSection)	org.netbeans.core.ModuleActions	117
void processAction(org.openide.modules.ManifestSection$ActionS...	org.netbeans.core.ModuleItem$InstallIterator	528
void invokeIterator(org.openide.modules.ManifestSection$Iterator)	org.openide.modules.ManifestSection$ActionSection	235
void forEachSection(org.openide.modules.ManifestSection$Iterat...	org.openide.modules.ModuleDescription	324
void restoreSection()	org.netbeans.core.ModuleItem	333

Pause

Figure 5. GUI tool.

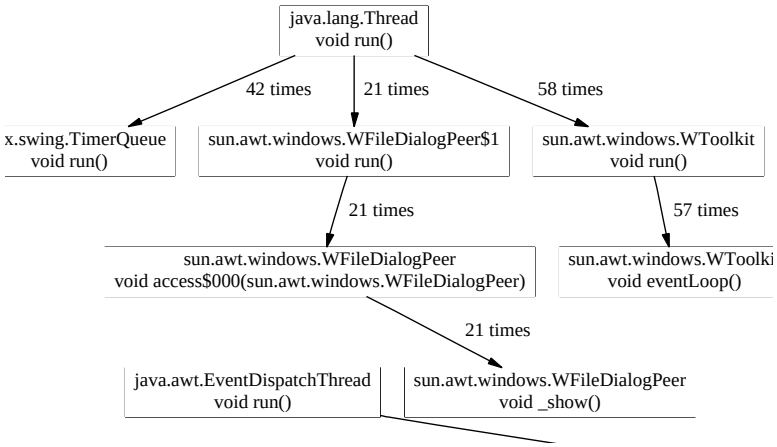

Figure 6. Call graph generator output (detail).

in any order and after different delays, this is not a global state.) After the user is done examining the system, clicking the `Pause` button again will update the display with the most recent information.

3.2.2. Call graph generator

We also wrote a call graph generator for xdProf, the output of which is shown in Figure 6. This tool generates files for the `dotty` graph visualization program[7]; it is highly configurable and has a GUI front-end.

4. ISSUES

4.1. Impediments to Data Collection

Three things can reduce the amount of information accessible to the xdProf client. First, when Java applications are compiled without debugging information (`-g:none`), line numbers and names of source files are inaccessible to xdProf. Second, Java byte code optimizers and obfuscators, such as DashOPro,[8] may shorten or rename classes and methods to less-descriptive identifiers, as well as removing or modifying the line number information.

Finally, the xdProf client is affected at run time by the amount of data returned by the JVMPI implementation. The presence of a just-in-time compiler (JIT)[9] or adaptive optimizer (such as Sun's HotSpot™ [10]) in a Java Virtual Machine can improve performance of a Java application drastically by performing various optimizations (method inlining, native code compilation) on the Java byte code at runtime. However, by converting some Java methods to native code, and inlining other methods, the call stack may no longer contain useful method references; maintaining one call stack for native code and another for JVMPI use may be costly, and so the implementation may return less information.[‡] Regardless of the virtual machine, xdProf will attempt to send as much information as possible.

4.2. Average Delay

The user specifies a *delay* in milliseconds for xdProf to use at VM start-up. However, there are several factors that make this delay a lower-bound value. First, there is an inherent scheduling error introduced by the operating system thread scheduler and the VM thread scheduler. Second, xdProf must collect information about the threads, classes, and methods and then send this information after the threads have been resumed. Therefore, the total delay between stack trace collection is $delay + scheduling\_error + time(collection) + time(transmission)$, and not the *delay* specified by the user. However, data collected during benchmarking, given in Appendix A, shows that the average time between stack traces (total time divided by number of traces) is no more than 7% greater than the original delay.

4.3. Control Flow Issues

The xdProf server receives stack traces containing the current call stack for each thread in the virtual machine. Reconstructing control flow by inspecting the transitions between call stacks is difficult to impossible. Since the xdProf client updates every *delay* milliseconds; the call stack can change significantly in this period of time. Also, the line number is not sufficient to determine if a method has been called again or if a method is using a loop or branch structure.[§] The fact that line numbers may be missing or unavailable is a complicating factor.

However, some aspects of control flow can still be determined with xdProf. Intuitively, if $f() \rightarrow g()$ occurs in several stack traces, and $h() \rightarrow i()$ occurs in fewer stack traces, it is clear that $f()$ is a long-running function and that code in $i()$ is executed less often than code in $g()$. Although we are examining whether data mining or clustering techniques could provide additional insight into control flow, the most promising approach seems to be integrating a static analysis tool with the xdProf server.

4.4. Selecting a Delay

Although we have presented benchmarks for delays of 100, 200, and 1000 milliseconds, we have not considered how to select the delay. Fundamentally, the delay depends on the network, application, and the xdProf server analysis tool used. A delay of x milliseconds will generate k times as much network traffic as a delay of kx milliseconds, so smaller delays may be suboptimal for low-bandwidth or high-latency networks. Smaller delays may work best for analyzing computationally-intensive applications while long delays may be acceptable for long-running interactive applications. Since the xdProf client can be easily configured and analysis tools can perform many different functions, the "best" delay is one that fulfills the requirements of the user without overloading the network or analysis tool.

[‡]For example, IBM's Java Virtual Machine will not output any call stacks when a JIT is enabled; Sun's Java Virtual Machine with HotSpot enabled will only return call stacks for threads that are not blocked.

[§]If a thread has the call stack transition $f() : line\ 25 \rightarrow g() : line\ 30$ and the next trace includes $f() : line\ 25 \rightarrow g() : line\ 15$ multiple transitions could have occurred: g() returns and f() calls g() again, g() loops from line 31 to line 10, g() and f() return and are called with different parameters, etc.

5. STATUS AND FUTURE WORK

The xdProf client consists of approximately 2000 lines of C/C++ source code; since the only platform-dependent code is network-related, porting to other virtual machines and platforms will continue to be fairly easy. The xdProf server framework is approximately 1700 lines of Java source code. The GUI tool and call graph generator tools are approximately 400 lines of Java source code each. xdProf is currently available at `http://xdProf.SourceForge.net` for Intel/Win32, Intel/Linux, and Ultra/Solaris platforms as open source software under the BSD license.[11]

Future work includes improving the network code in the xdProf client by adding data compression and increasing reliability and performance. Plans for future analysis tools include capture and analysis of large data sets, real-time program visualization, and reliability estimation.

APPENDIX A. PERFORMANCE TESTING DETAILS

We used an Intel Pentium II 350 MHz with 512 MB of RAM running Windows 2000 Professional for our testing. The web pages were served off of a machine running FreeBSD and the Apache web server. The low geometric mean in Table 6 is the SPECjvm98_base score; the high geometric mean is the SPECjvm98 score. We used the `timethis` utility[12] against a special build of xdProf for Table 7 and 8.

Table 6. SPEC ratios for Sun 1.3 HotSpot Client VM.

	without xdProf		**with xdProf local machine 100 milliseconds**		**with xdProf remote machine 100 milliseconds**	
Benchmark	*Low*	*High*	*Low*	*High*	*Low*	*High*
_227_mtrt	22.10	25.30	21.80	24.20	21.10	25.10
_202_jess	22.40	31.30	23.50	29.10	17.90	30.90
_201_compress	11.80	13.40	12.70	12.80	9.36	12.90
_209_db	12.90	13.80	12.60	13.10	13.20	13.20
_222_mpegaudio	32.80	35.40	32.00	33.70	32.70	34.50
_228_jack	30.40	38.40	30.50	36.60	30.60	36.60
_213_javac	9.15	12.40	9.62	11.60	10.10	12.30
Geometric Mean	18.30	22.00	18.60	20.80	17.40	21.40

Table 7. Network traffic details for Sun 1.3 HotSpot Client VM (without xdProf = 383.136 seconds).

Server	Refresh (ms)	Time (seconds)	Overhead	Traces	Total Bytes	Total Bytes / Traces	Time / Traces
local	100	412.863	7.76%	3860	11133599	2884.352	106.959
local	200	402.338	5.01%	1927	5507828	2858.240	208.790
local	1000	398.202	3.93%	391	1124780	2876.675	1018.419
remote	100	395.258	3.16%	3731	10608305	2843.287	105.939
remote	200	390.531	1.93%	1874	5348439	2854.023	208.394
remote	1000	392.143	2.35%	386	1102655	2856.619	1015.915

Table 8. Network traffic details for Sun 1.3 Classic VM (without xdProf = 3600.477 seconds).

Server	Refresh (ms)	Time (seconds)	Overhead	Traces	Total Bytes	Total Bytes / Traces	Time / Traces
local	100	4352.799	20.90%	40974	178144602	4347.747	106.233
local	200	4263.951	18.43%	20536	89289586	4347.954	207.633
local	1000	4207.349	16.86%	4174	18183230	4356.308	1007.990
remote	100	3680.101	2.21%	35230	152744536	4335.638	104.459
remote	200	3639.373	1.08%	17646	76579076	4339.741	206.244
remote	1000	3601.698	0.03%	3578	15630092	4368.387	1006.623

ACKNOWLEDGMENTS

We would like to thank John Steven, Ken Alverson, and the Advanced-Java mailing list for their assistance with the implementation of xdProf and the anonymous reviewers for their comments.

REFERENCES

1. D. Viswanathan and S. Liang, "Java Virtual Machine Profiler Interface," *IBM Systems Journal* **39**(1), pp. 82–95, 2000.
2. Sun Microsystems, "Java Virtual Machine Profiler Interface Documentation."
3. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, Reading, MA, 1997.
4. Standard Performance Evaluation Corporation, "SPECjvm98," 1998.
5. @stake Research Labs, "`netcat` 1.1 for Win 95/98/NT/2000."
6. Sun Microsystems, "Forte for Java (`http://www.sun.com/forte/ffj`)."
7. S. C. North and E. Koutsofios, "Application of graph visualization," in *Proceedings of Graphics Interface '94*, pp. 235–245, (Banff, Alberta, Canada), 1994.
8. PreEmptive Solutions, "DashOPro."
9. T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczk, "Compiling Java just in time," *IEEE Micro* **17**, pp. 36–43, May – June 1997.
10. D. Griswold, "The Java HotSpot Virtual Machine Architecture," 1998.
11. Open Source Initiative, "The BSD License."
12. Microsoft Corporation, *Microsoft Windows 2000 Professional Resource Kit*, Microsoft Press, 2000.