DYNAMIC INSTRUMENTATION FOR JAVA USING A VIRTUAL JVM

Kwok Yeung, Paul H J Kelly and Sarah Bennett
Department of Computing
Imperial College
180 Queen's Gate
London SW7 2BZ, UK
kcy@doc.imperial.ac.uk
p.kelly@imperial.ac.uk
s.bennett@imperial.ac.uk

Abstract Dynamic instrumentation, meaning modification of an application's instructions at run-time in order to monitor its behaviour, is a very powerful foundation for a wide range of program manipulation tools. This paper concerns the problem of implementing dynamic instrumentation for a managed run-time environment such as a Java Virtual Machine (JVM). We present a flexible new approach based on a "virtual" JVM, which runs above a standard JVM but intercepts application control flow in order to allow it to be modified at run-time. Our Veneer Virtual JVM works by fragmenting each method's bytecode at specified points (such as basic blocks). The fragmentation process can include static analysis passes which associate dependence and liveness metadata with each block in order to facilitate run-time optimisation. We conclude with some preliminary performance results, and discuss further applications of the tool.

Keywords: Java, dynamic instrumentation, virtual machine, reflection, dynamic introspection, performance analysis.

1. Introduction

Setting the Scene: optimizing Java RMI applications. The work we describe in this paper is part of a wider research programme at Imperial College, aimed at extending the technology of optimizing compilers to cross the boundary between systems on a network. Our main goal is to avoid unnecessary communications, and we have developed a prototype optimiser for Java applications that use Remote Method Invocation (RMI).

In looking for applications which might benefit from RMI optimization, we discovered the need for performance analysis tools. We also realised that the run-time optimisation framework which we had developed could also be used for performance instrumentation.

This paper presents the fruits of this insight.

The Veneer Virtual JVM. Static optimisation of Java is difficult due to dynamic binding, polymorphism, ubiquitous pointers, and dynamic class loading. Since communications (at least over a wide-area network) are expensive relative to computation, we can afford to invest in some run-time effort if it offers a reasonable prospect of reducing the number and/or size of messages required.

Although such optimisation could be done by extending a sophisticated Java Virtual Machine (JVM) such as the Jikes RVM [1], this would prevent users from using their chosen JVM, and would involve us in tracking JVM releases. Instead, we built Veneer, which operates on top of a standard JVM, running as a Java application. In effect, the framework is, itself, a JVM - which runs application class files in a controlled environment. Veneer is a "virtual" JVM, carefully designed to run reasonably fast by executing most of the application code directly — it jumps to the corresponding bytecode. It maintains control over execution by intercepting control flow; optional intercept points include method entry, basic blocks, and back edges.

This paper. The main contributions of this paper include:

- We give an overview of our Veneer Virtual JVM, an execution environment for Java bytecode applications which allows dynamic instrumentation, run-time optimization, and makes the results of static analyses available to inform run-time optimization.

- We briefly present JUDI, our Java Utility for Dynamic Instrumentation, a component-based environment which exploits Veneer's capability to modify an application on the fly.

- We briefly introduce the RMI optimisation tool which motivated this work.

We conclude with a discussion of directions for further work building on these ideas.

2. Related work

Dynamic instrumentation. The term "dynamic instrumentation" was coined by Hollingsworth and Miller to describe run-time patching of an application's binary code in order to monitor and measure the program's behaviour. Hollingsworth has published a portable library, DynInst [8], which supports this on a variety of processor types and operating systems.

The Paradyn Parallel Performance Tools [12] build on DynInst, to provide a tool for measuring and analyzing the performance of sequential, parallel, and distributed programs.

Dynamic instrumentation for Java. DynInst works by patching the application's instructions. Dynamic instrumentation for Java cannot be implemented this way, without exposing low-level implementation details of the JVM. There are a number of alternative approaches:

- Re-define the class using the Java Debug Interface (JDI) call `VirtualMachine.redefineClasses()`, introduced in Sun's JDK 1.4 [6]. This approach is used in ProbeMeister [13]. The overhead to do this is reported to be around 20 milliseconds for a small example, but increases with large classes since methods cannot be redefined individually, and JIT optimisation must be re-done. To reduce the overheads, Dmitriev [7] advocates refining the JDI with a call to redefine methods individually.

- Run the JVM in debugging mode, and set breakpoints to insert instrumentation. This is the approach taken by Popovici et al [14].

- Run the Java application in a virtual JVM. This is the approach used in our JUDI tool, and is presented in more detail in Section 1.4.1. We use the native JVM to execute application bytecode as much as possible, but have to intercept execution in order to retain control. The scheme suffers some overhead (see Section 1.5) on execution of all the application's code (apart from system libraries), but runs with JIT optimisation. Insertion and removal of instruments is very fast.

  Our vJVM was developed to provide a general framework for run-time optimisation, and is much more powerful than is needed for

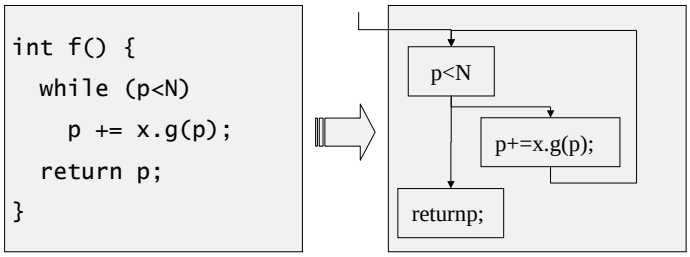

Figure 1. Fragmentation. When each class is loaded, each method may be fragmented according to a specified policy. For some purposes it is sufficient to intercept control flow only at method entry and exit. For our dynamic instrumentation tool we fragment at basic block boundaries - forks and joins in the method's control flow graph. Figure 4 shows the fragment graph for a real example.

dynamic instrumentation. In Section 1.5 we explore some of the potential advantages of the using Veneer compared to the alternatives above.

Runtime Introspection and Optimization. The idea of interposing a software layer to monitor, intercept or optimise an application is interestingly explored by the Dynamo/Rio projects [2, 4]. Our virtual JVM provides essentially the same capability, and we encounter a similar problem of intercepting control flow efficiently.

Performance analysis for Distributed Java applications. Various tools already exist for analysing Java performance; for distributed applications, for example, the JaViz tool offers a useful solution [10]. Our goal for JUDI is to build an infrastructure for more ambitious instrumentation. We review some of the possibilities in Section 1.7.

3. The Veneer Virtual Java Virtual Machine

Our instrumentation and optimisation tools are built on a virtual Java Virtual Machine (vJVM), which is a JVM written in Java, running on a Java JVM. It runs most of the application code directly, by jumping to the corresponding bytecode, but selectively maintains control over execution by intercepting control flow. We can choose the level at which control flow is intercepted, e.g. at method entry, basic blocks, back edges.

The control flow is intercepted by "fragmenting" each method at class-load time. There are several different fragmentation policies: at basic block boundaries, at method entry/exit, at method calls, and at potential RMI call sites (used for our work on RMI optimisation, described in Section 1.6). Figure 1 shows a simple example of basic block fragmenta-

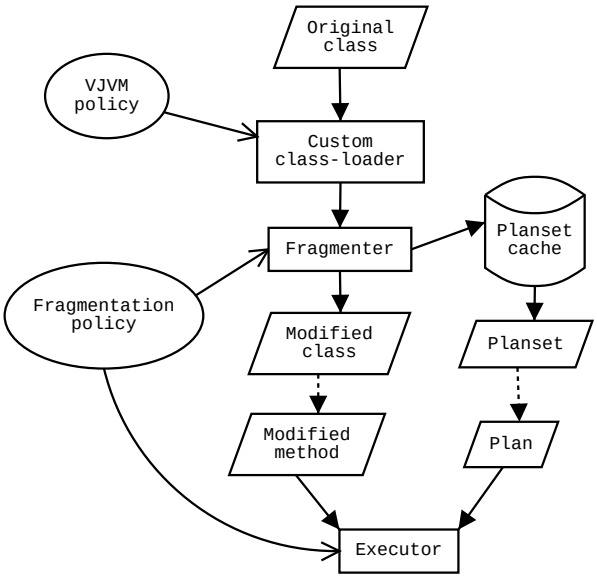

Figure 2. Architecture of the vJVM. We intercept the JVM's class loader, and replace each method of each class with a call to a plan executor. For each newly-encountered class we generate a planset, the fragmented code variants, which are traversed by the executor. Plansets are cached to avoid redundant work.

tion. The method body is split into blocks, an execution "plan", and the method entry is replaced by an "executor loop" that walks the control flow graph, invoking each block in turn. A method's control flow graph can be updated without synchronisation, as the application is running, allowing us to use this as a framework for dynamic instrumentation.

An architectural overview of the vJVM is given in Figure 2.

3.1 How it works

The Virtual Java Virtual Machine (vJVM) uses a custom class-loader to intercept classes as they are loaded. If the class belongs to the Java core library or to the vJVM, then its loading is delegated to the parent class-loader.

If modification is necessary, the methods of the class are processed one-by-one. The active policy is invited to generate *variants* for the method — the execution plans that can be executed in place of the original method body. Each method can have multiple variants, fragmented according to differing policies, although only one can be active at a time. The active variant may be changed at runtime, which effectively switches method implementations on-the-fly.

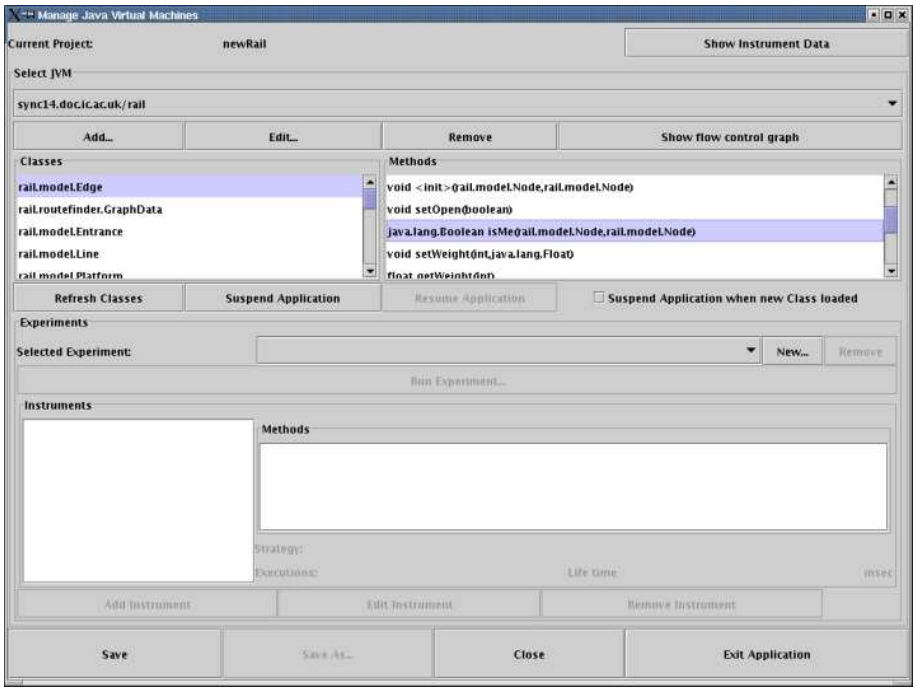

Figure 3. Java Utility for Dynamic Instrumentation. This prototype user interface connects to a specified, possibly remote, JVM. The user can then browse the classes and methods, and inspect their control-flow graphs. Instrumentation experiments are constructed, then deployed for a specified period, or until an instrument execution count is reached. Each instrument is packaged as an Instrument Strategy Component; an example is shown in Figure 4.

Finally, the modified class is loaded into the underlying JVM.

3.2 The fragmentation framework

The fragmentation framework breaks the body of a method up into blocks. Blocks represent sections of code from the original method, plus some additional meta-data. The structure of the original method is retained by building up a control-flow graph known as a *plan*, with the blocks forming the nodes of the plan. All the plans for a class are gathered up into a construct known as a *planset*.

The method is executed by invoking a method in an *executor* class, which takes the plan as a parameter. The executor traverses the plan, executing the blocks as it visits them. The executor therefore takes on the role of an interpreter, and has full control over the way in which the blocks are executed.

Fragmentation policy. The way in which a method is fragmented and the executor that is used is determined by a user-defined policy. In general, we attempt to pack as much as possible within fragments, since these can be executed directly by the JVM and are therefore fast. Breaks between blocks are introduced where we need to regain control over the system, since at these points control is passed back to the executor. Parameterised blocks are used to identify certain types of instruction that may need to be treated specially by the executor.

Method entry. If a method is fragmented, its body is replaced by a sequence of instructions that initialises the locals array, retrieves the plan and executor for the current method, and then executes the plan using the executor.

Planset caching. The fragmentation process, which is implemented using the SOOT framework from Hendren's group at McGill University [18]) is rather involved, as data flow analysis is used to minimise fragmentation overheads (the metadata resulting from this analysis is used more aggressively in our RMI optimisation work). Plansets are cached in the local filesystem and reused if classname and SHA-1 signature match.

4. JUDI: Java Utility for Dynamic Instrumentation

JUDI is a prototype dynamic instrumentation tool for Java. It manages the deployment, removal, data collection and data visualisation for performance instrumentation experiments.

4.1 Instrumentation

Instrumentation is done by inserting instruments as additional blocks in the method plan. These instruments are then executed by the executor.

The instrument can access the method's parameters and locals, and call other classes, for example to trigger further instrumentation. For high-resolution timing, we used JNI to access a C/assembler routine which reads the Intel timestamp counter.

As illustrated in Figure 3, JUDI's client graphical user interface (GUI) connects to a set of remote vJVM's running fragmented code. The GUI allows the user to browse the remote systems' methods, and to upload instruments to the remote systems. The methods can be viewed by a visualiser (see Figure 4) which shows the plan as a graph (using Open-

JGraph [9]). JUDI's record of accessible classes grows as classes are dynamically loaded, but persists from run-to-run to allow instrumentation of methods in advance of their execution.

4.2 Instrumentation Strategy Components

The unit of instrumentation deployment is an "Instrumentation Strategy Component" (ISC). This consists of:

- A set of **Instruments** - subclasses of a generic Instrument plan block. Instruments typically start, stop and log timers, or generate a log entry recording control flow, or data values (in aspect-oriented programming terms, this is the "advice").
- An **instrumentation strategy**. This is usually just whether the instrument is to be executed before, after, or before-and-after the specified method, and whether it applies to the whole method, or every basic block in the method.
- **Instrumentation targets**: the set of program objects (methods, classes) to which the instrumentation strategy should be applied. If not the entire program, this is selected explicitly through the GUI.
- **Instrumentation data class**: instruments generate data, usually either a log or some kind of histogram.
- **Instrumentation analyser**: this is a GUI component for viewing the results from the experiment.

Figure 4 shows the results from an example ISC which traces control flow through selected methods. This ISC operates on methods fragmented at the basic-block level. The instrument is applied before each block, and simply logs the method and block id. The GUI allows us to select the particular method of interest, and view the results. The instrument analyser displays its control flow graph, the three distinct control flow paths which were taken through this method, the number of times each was executed, and the average execution time for each path.

5. Experimental results

We have evaluated JUDI using two substantial applications:

1 SpecJVM98_209_db (Data Management) Benchmark [16]. 1028 lines of code, 3 classes, 24 public methods.

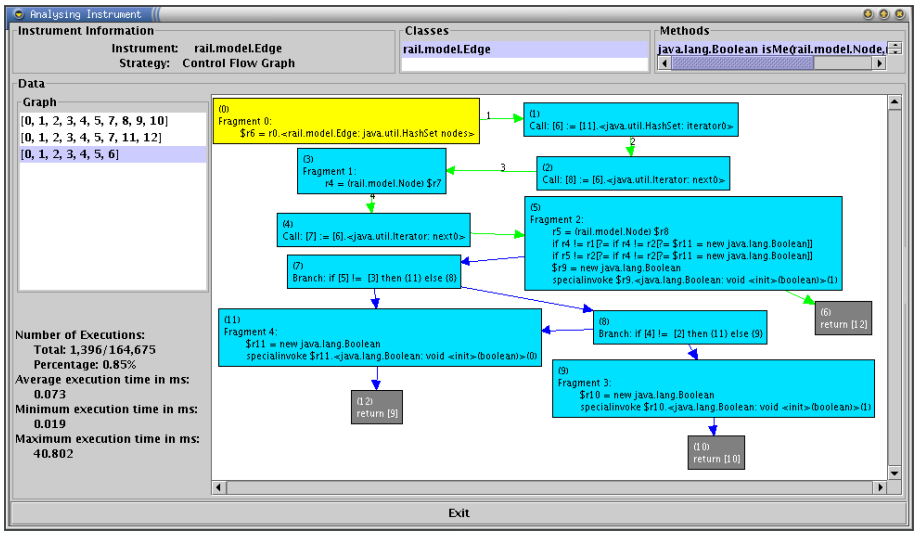

Figure 4. Fragmented execution plan for the example method `isMe`. In this example fragmentation has been applied at basic block boundaries, so that control flow within the method can be studied. The light-coloured block is the root of the plan which is always executed first. The small blocks are terminating blocks, where the method exits. The text in the boxes (unfortunately illegible here) shows disassembled code.

This display was generated by an Instrumentation Strategy Component (see Section 1.4.2) designed for control flow analysis. This ISC logs which control flow path was taken through the specified method, producing a histogram of path frequencies together with each path's mean, minimum and maximum execution time.

Benchmark	*Exec^n time (JDK1.4)*	*Exec^n time (vJVM over JDK1.4)*	*Slowdown factor*
SpecJVM98_209_db	22.02s	24.30	1.10x
RouteFinder	3.73s	28.13	7.54x

Table 1. Slowdown due to running benchmark applications under the Virtual JVM (without any instrumentation). The performance impact varies enormously. Note that system and standard library methods are not fragmented and run at full speed. Times are average of five runs.

2 RouteFinder, a railway route finder application written by a group of MSc students at Imperial College in spring 2002. 3192 lines of code, 17 classes, 145 public methods.

The experiments were run on a 1400MHz AMD Athlon processor with 512MB RAM, running Linux (Suse 7.2) using Sun JDK1.4.

Virtual JVM performance. Running an application under the virtual JVM leads to an inevitable increase in execution time due to the need to intercept control flow. The performance impact is given in Table 1. Our performance compares well with the reported slowdown of more than 700 for Sun's fully-interpretive JavaInJava virtual JVM [17], but is some way from matching the approaches discussed earlier based on the JDI `VirtualMachine.redefineClasses()` feature [13] and using breakpoints [14].

Veneer is at an early stage of its development, and we expect to improve its performance substantially. The implementation whose performance is reported here suffers a substantial overhead at each fragment boundary. The design allows for each method to have several different implementations, fragmented to different degrees. Thus, the overhead could be reduced to just a test or indirection on method entry (even this could be avoided in an implementation that can appropriate the method table). However, to ensure instrumented threads executed newly-inserted instruments promptly, the back-edges of loops generally also need to be intercepted.

We could also improve performance by inlining instrumentation code into the modified method (or method variant) as it is loaded. This is likely to be particularly useful when large amounts of an application's code is to be instrumented.

6. Optimising RMI applications

Figures 5 and 6 illustrate the two main optimisations we have implemented to reduce communications in Java RMI applications.

To optimise RMI, we configure Veneer to fragment only methods which contain potential RMI call sites (interface invocations with `java.rmi.RemoteException` on the throw list). The fragmentation is used so that the run-time system is invoked before each potential RMI call. If the target object is in fact remote, and the following fragment has no dependence on it, the RMI call is delayed. This way, RMI calls and local code are dynamically re-ordered. Eventually, when a dependence or externally-visible effect forces execution of the delayed RMI calls, the run-time system constructs an optimised execution plan which

```
void m(RemoteObject r, int a)
{
  int x = r.f(a);
  int y = r.g(a,x);
  int z = r.h(a,y);
  System.out.Println(z);
}
```

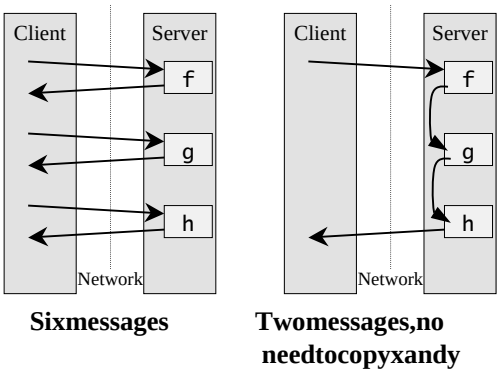

Figure 5. Aggregating adjacent, or near-adjacent RMI calls to the same server. Aggregation always reduces the number of messages. It may also reduce the amount of data transferred, since parameters used in multiple calls (such as `a` above) need be sent only once, and a result from one call passed as a parameter to another need not be routed via the client. Results which are not used by the client, such as `x` and `y`, need not be returned.

```
void m(RemoteObject r1,
       RemoteObject r2)
{
  Object a = r1.f(b);
  r2.g(a,b);
}
```

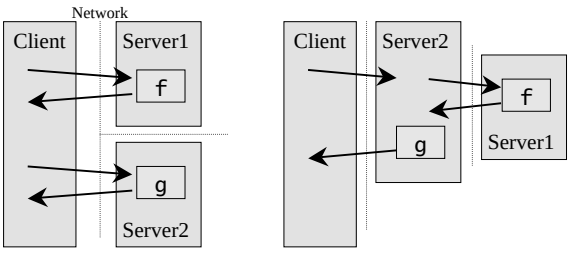

Figure 6. Aggregating adjacent, or near-adjacent RMI calls to different servers. If a result from one call (such as `a` above) is passed as a parameter to another, we can route the data server-to-server instead of server-to-client then client-to-server. This reduces marshalling costs and allows selection of a better server-to-server network path, if available. With a slow client-server connection, it may even be worthwhile purely to exploit common parameters such as `b`.

implements the aggregation and forwarding optimisations illustrated in Figures 5 and 6.

Performance results for RMI optimisation are currently being evaluated [19].

7. Conclusions and directions for further work

We have presented the Veneer virtual JVM. Veneer allows a Java application's behaviour to be selectively monitored and modified at runtime. Its design was motivated by our work on optimisation of RMI, where we combine static analysis with dynamic control flow. Veneer can also be used for performance instrumentation, and we presented JUDI, a prototype performance analysis tool. The tool is based on Instrumentation Strategy Components (ISCs), which combine instrumentation, an instrument deployment strategy, and instrument data analysis.

Given that much of this work is at an early stage, most of the interest lies in the directions for developing and applying it:

- **Veneer.** Veneer is at an early stage of development and we plan to improve its performance, and evaluate it more thoroughly. The tool has a number of interesting potential applications, particularly in security, which we are currently exploring.

- **JUDI.** The power of dynamic instrumentation lies in the ability to deploy instrumentation algorithmically. The idea is to formulate a hypothesis about a performance bottleneck, deploy an experiment to test it, then refine the hypothesis on the basis of the results. The approach has been explored in the Paradyn Performance Consultant [5, 15]. We have a preliminary implementation (as a JUDI ISC) which we are currently developing [3].

- **Aspects.** How should a JUDI user specify which program points a given instrument should be attached to? Aspect-oriented programming (AOP) languages such as AspectJ offer an answer to this [11]. AspectJ is based on a reflective model of a Java program, and uses this to define a language for specifying "joinpoints" (i.e. points at which code should be added or interposed). AspectJ supports wildcards on method type signatures, names and package pathnames. Wildcards on methods can be combined with some dynamic properties, such whether an object's run-time type matches a given pattern, or whether one method is called via another.

  The PROSE tool of Popovici at al [14], which, as was mentioned earlier, implements dynamic instrumentation for Java by setting breakpoints via the debugging interface, is designed to support

run-time deployment of aspects. They dynamically construct "cross-cut" objects to represent how a specified code fragment is inserted into a running application.

We plan to redesign JUDI using an AOP-style language for characterising program points, and for classifying the measurements produced from instrumentation.

Acknowledgments

This work was supported by the EPSRC, through grant no. GR/R15566 (DESORMI). We would also like to thank Doug Brear, Thomas Petrou, Thibaut Weise and Tim Wiffen, who all contributed to the software development.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), February 2000.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[3] Douglas J. Brear. JBolt: The Java bottleneck locator toolkit. Master's thesis, Department of Computing, Imperial College, London, UK, 2002.

[4] Derek Bruening, Evelyn Duesterwald, , and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.

[5] H. Cain, B. Wylie, and B. P. Miller. A callgraph based search strategy for automated performance diagnosis. In Arndt Bode, Thomas Ludwig II, Wolfgang Karl, and Roland Wismüller, editors, *Euro-Par*. Springer Verlag, LNCS 1900, 2000.

[6] M. Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference, Tampa Bay, Florida, USA*, October 2001.

[7] M. Dmitriev. Application of the HotSwap technology to advanced profiling. In *First International Workshop on Unanticipated Software Evolution (USE2002)*, Malaga, Spain, June 2002. http://www.joint.org/use2002/sub/dmitriev-hotswapprof.pdf.

[8] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. In *Proceedings of SHPCC'94*, May 1994.

[9] Jesus M. Salvo Jr. Openjgraph - Java graph and graph drawing project, October 2002. http://openjgraph.sourceforge.net/.

[10] Iffat H. Kazi, Davis P. Jose, Badis Ben-Hamida, Christian J. Hescott, Chris Kwok, Joseph A. Konstan, David J. Lilja, and Pen-Chung Yew. JaViz: A client/server java profiling tool. *IBM Systems Journal*, 39(1):96–, 2000.

[11] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP*, volume 2072, pages 327–355. Springer Verlag, LNCS 2072, 2001.

[12] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[13] Paul Pazandak and David Wells. ProbeMeister: Distributed runtime software instrumentation. In *First International Workshop on Unanticipated Software Evolution (USE2002)*, Malaga, Spain, June 2002. http://www.joint.org/use2002/sub/pazandak-ProbeMeister.pdf.

[14] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect oriented programming. In *1st International Conference on Aspect-Oriented Software Development (AOSD 2002), April 22-26*, Enschede, The Netherlands, 2002. http://ikplab11.inf.ethz.ch:9000/prose/webthings/aosd02.ps.

[15] P..C. Roth and B. P. Miller. DeepStart: A hybrid strategy for automated performance problem searches. In *EuroPar 2002*. Springer Verlag, 2002.

[16] Standard performance evaluation corporation (spec) jvm98 Suite, 1998. Available from http://www.spec.org.

[17] Antero Taivalsaari. Implementing a Java Virtual Machine in the Java programming language. Technical Report TR-98-64, Sun Labs, 1998. http://research.sun.com/kanban/JavaInJava.html.

[18] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of CASCON '99*, 1999.

[19] Kwok Cheung Yeung. *Automated Optimisation of Distributed Java Programs across Network Boundaries*. PhD thesis, Department of Computing, Imperial College, London, UK, 2002. In preparation.