NetProf: Network-based High-level Profiling of Java Bytecode *

Srinivasan Parthasarathy, Michal Cierniak, Wei Li

{srini,cierniak,wei}@cs.rochester.edu

The University of Rochester
Computer Science Department
Rochester, New York   14627

Technical Report 622

May 1996

Abstract

In this paper we present a system for network based visualization of profile information generated by Java applets/bytecode. The system, called **NetProf**, is composed of several components each of which is interesting in their own right. The components are a bytecode to Java source code translator, a profiler that includes a static pass to insert profiler code, a dynamic runtime library that records relevant events and finally a visualization mechanism which highlights the profile information in an easy to use manner. All of this can be done over the Internet using a client-server approach and is independent of the underlying architecture/machine and human intervention.

Keywords: Java, Bytecode, Profiling, Visualization, Client-Server, World Wide Web, Internet.

---

1   Introduction

The past few years have seen a remarkable interest level in the Internet. This is mainly attributed to the prominence of the World Wide Web and using Web browsers to read news briefs, down-load relevant papers, order books and services etc. The reasons for the popularity of these browsers are that they are graphics based, and extremely easy to use. The number of applications within the Internet has grown steadily over the past few years. The main issues while programming such applications are: security, portability, allowing for a high level of inter-activeness while guaranteeing performance.

Java [9], is a language that seeks to address these problems while providing a relatively easy programming environment in which to build such applications. Java source code is compiled into bytecode (a representation of the Java Virtual Machine [11]), which is then interpreted. The bytecode is independent of the architecture and can be interpreted on any machine that has a Java interpreter. Complete descriptions are available in [10, 9, 3]. Java also provides mechanisms to embed programs (called applets) in Web pages. For example, Netscape Navigator 2.0 and beyond has a built-in Java interpreter that can execute these applets. This ability has transformed the Web from being a passive store of linked data to a store of interactive applications albeit interspersed with linked data. The advantage of running these programs through a browser is that they are interspersed with text, pointers to relevant information, and visual images that instruct in executing them, i.e. user friendly. This is opposed to down-loading the file and executing it in stand-alone mode. Down-loading time, access to a machine that supports an interpreter and availability of the bytecode are issues that have to be dealt with here.

Within the context of Java programs it is interesting both for developers and users to evaluate the performance of their code on different machines. Machines with varying processor speeds and on networks with different speed, bandwidth and contention may result in different performance bottlenecks. Both the user and developer need to find out where the performance is suffering and whether based on this information they can improve it. From the user's viewpoint it may not be possible to identify the performance bottlenecks if the original source code is not available as bytecode is not easy to read. Also even if the source code is available it may be more appropriate to base the profile on bytecode, to take into account any optimizations performed by the Java compiler. Program profiling is a well known technique for recording program behavior and measuring performance. A profiler for Java is useful for measuring instruction set utilization measuring program bottlenecks, and for possible compiler/runtime driven optimizations.

High-level program sources (to profile) in Java may not be readily available because of the presence of the intermediate bytecode. Inserting profile code in Java bytecode and generating program profiles is possible but it is not useful to the end user unless it can be visualized with high level constructs present. Wading through

1

a stream of bytes to get the performance related information is not a exactly walk in the park.

The final issue is that of doing all this over the Internet. It is quite possible for developers to restrict [1] their applets to run only through a browser and prevent them from being downloaded and run in stand-alone mode. Also if the user is just interested in the performance of a wide number of benchmarks without having to allocate storage for all of these benchmarks in the local disk then down-loading and executing is not the best option. Treating the Web as a secondary store may be a nice option. Obviously for this to happen the profiler must be run over the network.

These issues mean we need a system that can profile Java programs, provide high-level visualization connecting the bytecode to the performance profile and that can operate over the Internet.

The main components (the required functionality and properties) of NetProf are:

- A bytecode to Java (bc2j) source code reverse compiler. The bc2j reverse compiler recovers high-level control flow information with a high success percentage. This seeks to address the high-level aspect of our system.

- A profiler that consists of a static pass and a runtime library. The static pass instruments recovered code with necessary declarations and function calls pertaining to the runtime profiler library. The runtime library records the relevant events.

- A visualization mechanism to easily focus on particularly interesting profiles. Should be easy to view summary and/or complete information.

The main contribution of this paper is a prototype implementation of NetProf that demonstrates the above features and how they can be used in a network setting.

The rest of this paper is organized as follows. Section 2 deals with the overview of the design of such services and important network and security issues that play a role in the current implementation. Section 3 focus on the technical (high-level) details of each component in the system and describe what the system looks like to the end user. In Section 4 we describe some implementation (low-level) details of our system. Finally in Section 5 we summarize our conclusions.

---

[1] With the presence of a bytecode to source code translator it may not be possible to completely restrict this. However they can still make it very difficult for the user to run the program in standalone mode by inserting appropriate checks points in the code. Removing or accounting for these checks may be incentive enough to ensure that the program is not downloaded.

2 NetProf: System Overview

In this section we describe the overall picture of NetProf. We first describe what we would like to have in our eventual system. We then describe what we currently have and why.

2.1 System Design

Below we describe the system through the following example. A client loads the tool over the net, selects the source to profile (available anywhere on the net), and then types profile. The system should then be able to recover the source code, instrument profiler information and run the program. Then through the visualization tool the client should be able to view the profile information over the net. All of the four components (client, bytecode source file, decompiler/profiler and visualizer) in the system can reside on different nodes on the Internet. The computation, however, will be performed on the client node, after the user grabs the remote components through a browser.
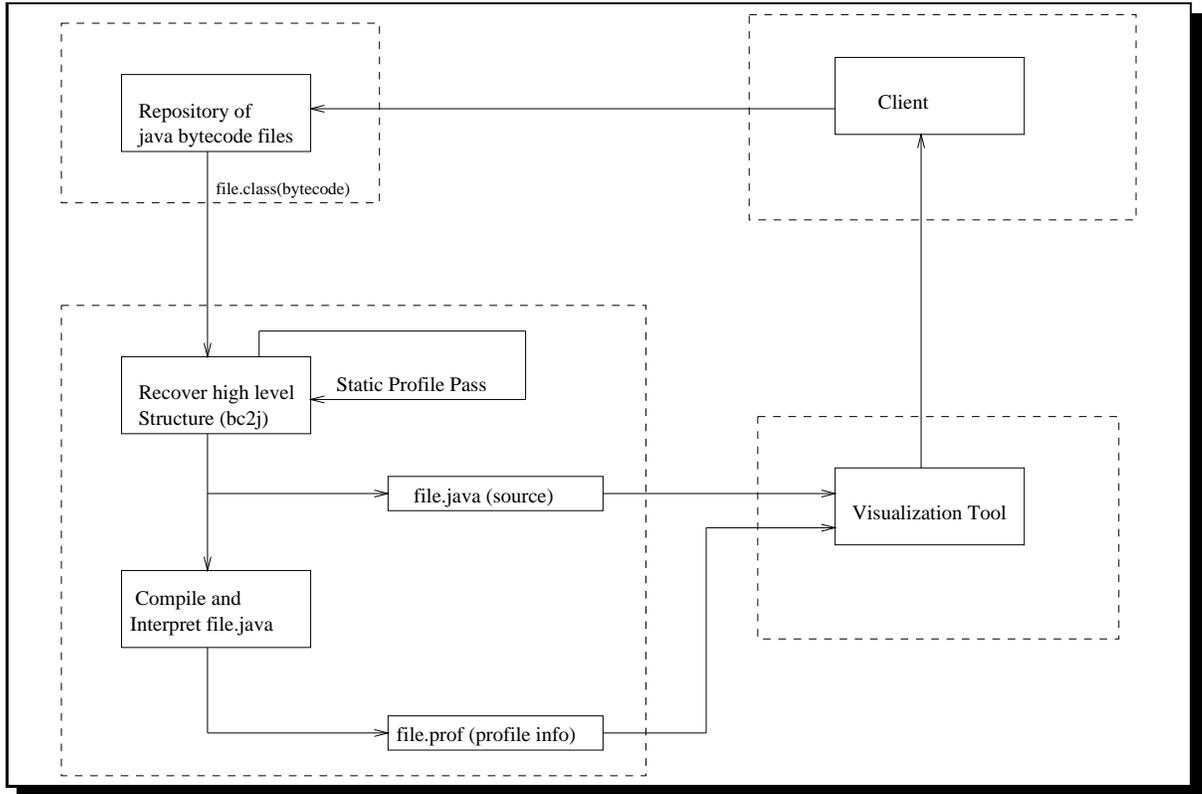
Figure 1: Logical System Components

3

Clearly this kind of setup requires support at the level of the Web browser, especially to connect/retrieve other source files, dynamically load and modify classes on the fly and write to files that can be accessed over the Internet, in a manner that is transparent to the user. The advantage of the setup is everything can be done at a level where most users need not have to know anything about the Internet at all. Since the whole setup is at a client-server level there are no human intervention delays.

Figure 1 describes the logical structure of the overall system. The highlighted boxes (formed with dashed lines) indicate the independent logical entities in our system, where these entities can physically reside on different nodes in the net. The client connects and determines a bytecode (file.class) file that it wants to profile. This file is fed through the bc2j and static profiler pass to generate the high level source combined with instrumented profile code (file.java). This file.java can be compiled and interpreted to generate the profile information (file.prof). Both the file.prof and file.java files are used as input to the visualization tool that then presents the profile information hierarchically along with the high level source (removing the instrumentation) to the client.

2.2   Current Restrictions

The main restriction we faced was that the current Web browsers with built-in Java interpreters have strong security restrictions that disallow some of the functionality we require. For instance we cannot instrument the code over the network without support of dynamic class modification and loading which are currently not supported.

In the current implementation since the Web browser does not have the above functionality, we could not separate the independent logical entities as shown in figure 1. We had to simulate the environment in the following manner. We have the bytecode file (to profile) and the profiler on the same host. On client request the source (bytecode) file is reverse compiled, the static profiler pass is inserted, and the source file is recovered along with calls to the runtime profiling system.

The connection to this server is established through a simple applet. The applet establishes a connection with the server at a known port number on the host machine (the host from which the applet is downloaded). Once the connection is established it passes in the name of the bytecode file to profile. The profile and high level source code are generated and are then written out to a location which is accessible through the World Wide Web. The visualizer then kicks in and displays the profile information in a hierarchical manner showing only profile information that is requested by the client and supplying summarycomplete information on request. Currently this step (visualization) can be used independent of the other stages, on a different server as long as the two input files have already been generated and are accessible to it.

3 Prototype System

In this section we describe the higher level details of how the prototype system is set up. The system organization can be naturally divided into three separate phases. These are:

- **bc2j:**

  A bytecode to Java source code, reverse compiler. This is a specific instance of an optimizing compiler being developed by us. The compiler reads in Java source or bytecode, converts it into a common high level intermediate representation **JavaIR**, and writes it back as Java source, bytecode, or other target language. JavaIR can be modified by multiple passes (optimization and/or profiling) before the source code is generated. For the purpose of this work, we read in bytecode for generality (we do not need access to the source that we want to profile) and write out Java source for readability.

- **Profiler:**

  It constitutes of two parts:

  1. A static pass to instrument recovered code with necessary declarations and function calls pertaining to the runtime profiler library.

  2. A runtime library that record relevant events without being too obtrusive. Currently methods, loops and conditionals are instrumented.

- **Visualization:** A mechanism to clearly pinpoint on sources of performance hits. It takes as input; output from bc2j and the profiled information from the Profiler. It displays the recovered source code (minus profiler declarations and calls) and facilitates clicking on different profiles to view summary and/or complete information.

How these phases are interconnected and put together is described below.

3.1 bc2j

We have been developing a compiler with a common intermediate representation and multiple front- and back-ends. For the work described in this paper we have built a bytecode front-end and a Java source back-end. We call this version of the compiler: **bc2j**. The instrumentation code of the profiler is integrated with bc2j, so that the complete profiler reads in a bytecode representation of a given class, converts it to an intermediate representation, instruments it, and writes to a file as Java source code. The Profiler modifies the JavaIR generated by bc2j and then generates the Java source code.

3.2 Profiler

Program profiling is a well known technique for recording program behavior and measuring performance. The basic idea is to insert code into a program and execute the modified program. Program profiling monitors the number of times that each basic block or control structure executes. It is used to measure instruction set utilization, bottlenecks in the program and possibly complier driven/runtime code optimizations [1, 4, 5, 8, 7, 6, 2].

Currently a good profiler for Java does not exist to the best of our knowledge. The built-in profiler provided with the JDK profiles only methods, and this is not sufficient information for either the client-user or the developer. It is not necessary the case that optimal algorithms written for C or C++ are optimal for an interpreted language like Java (some features may be much more expensive, and if avoidable could be avoided). This further necessitates the need for a good profiler.

The profiler could not be implemented along the lines of gprof [4] or any interrupt based sampling profiler as that would require hooks into the Java interpreter which are currently not provided. The only option available to us was explicit recording of events (externally). It has the option to be more exact than any of the statistical (sample) based methodologies, in that it does not depend on the program to run for a long time (due to the number of samples). We say option because the user can trigger whether to maintain summary information or information for every call in every instance. The former is expensive in terms of memory utilization but is more exact and maybe useful in Java programs (and there are many of them out there) which do not have multiple object instantiation, and/or deep recursion. Summary information has lesser overhead but is still more exact than statistical methods.

The Profiler as mentioned earlier consists of the static pass that runs over the JavaIR and instruments the code where appropriate, and a runtime library that records the relevant events. In our current implementation we profile basic blocks. Basic blocks include methods, loop structures and if-then-else conditional statements. These are the basic blocks that are currently generated by bc2j.

Static Pass

The static pass is done after the abstract syntax tree is constructed by bc2j and before the final call to generate the recovered source file. We instrument the various blocks as follows

- **Loops**: A call to the runtime profiler before and after the end of each loop is inserted. Within the loop a call to keep track of iteration count is inserted.

- **Methods**: A call to the runtime profiler at the start of and at the end of each method (before all return statements) is inserted.

- **Method Calls**: Calls are inserted before and after every method call. If the method is not being profiled then this returns total time of execution for the method.

- **If-then-Else Statements:** Calls are inserted at start of then and else clauses to record the number of times each branch is taken.

- Additionally some declarations are inserted into each of the classes being profiled. These are described in detail in the next section.

Runtime Library

The runtime library through calls and declarations inserted by the static pass records the following events for the corresponding basic blocks.

- **While Loops:** We record the number of iterations and total time each execution of the loop.

- **If-then-Else Statements:** We record number of times the if branch is taken and the number of times the else branch is taken.

- **Methods:** We record the total time of execution of the method, and invocation and release times. Invocation time is time when method is called to time taken to reach the first statement of the method. This is a rough measure of the time to index into the method table (since Java supports polymorphic methods), save status, and method initiation. Having this information in case of polymorphic methods is especially useful to see if under interpretation the execution time was quite large. (Polymorphic) Code-reuse vs Monomorphic methods could have an interesting performance perspective especially when the code is being interpreted. Release times measure the direct opposite of invocation times.

More details on the exact implementation of this system are provided in the next section.

3.3   Visualization

The Visualization aspect lends to the overall appeal of being easy to use. The profile information that is generated can be huge and can be extremely time consuming to go through. The visual display unit can read this file and compute summary information and display that to the user. Alternately if the user is interested in a particular instance of an object and wants to see the profile information of all times a method was called in that instance, they can in an easy manner.

The basic components of the visual tool are:

- **Input Preprocessing**: This component is basically responsible for preprocessing both the profile information generated by the runtime system and the recovered source code from (bc2j + profiler) the static pass.

- **Initial Display**: The initial display consists of the recovered source arranged as a list of selectable rows and a profile cache. The source is appended with comments indicating which rows can be selected to generate useful profile information. This is shown on the right hand side of fig 2. The row marked

  ```
  "    // Click Here for While Profile-> 1"
  ```

  can be clicked on and the resulting dialog box shown in figure 3 pops up. Scroll bars are provided for scrolling through the recovered source. The left hand side constitutes a profile cache that contains user desired profile summary. This area can be written through the profile window and is described below. The two buttons provided at the bottom control panel are for modifying the contents of the profile cache.
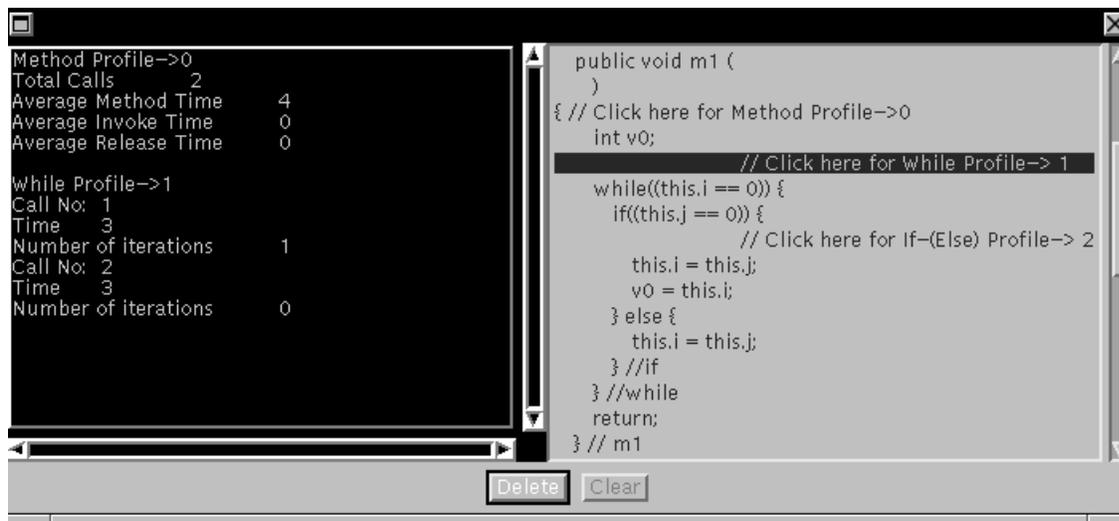

Figure 2: Initial Display

- **Profile Window**: On clicking one of the selectable rows a profile window pops out that has a list of instances and instance summaries one can choose from on the left hand side. After selecting the elements that are interesting (as determined by the user) on clicking the "Display Info" button displays the information selected by the user as shown in figure 3 on the right hand side. "Clear Info" clears the text area on the right, while "Close" closes the profile window. The "Cache It" button is used for caching user determined profile information in the profile cache in the initial display. The user can select the
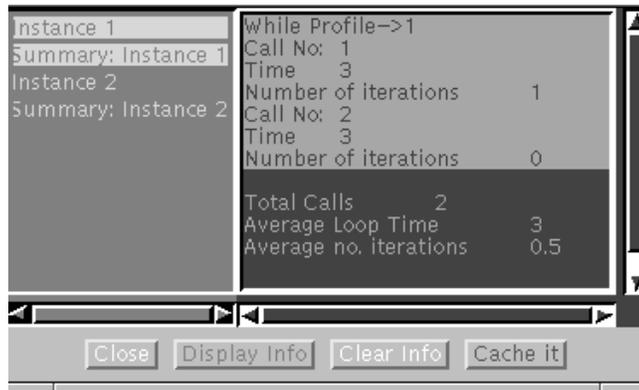
8

Figure 3: Profile Window

relevant profile information to store by highlighting the relevant text on the right hands side of Profile Window. Then on clicking the 'Cache It" button the text gets displayed in the initial display. Scroll bars are automatically generated for both the list elements and the text area where necessary.

4 Implementation Issues

In this section we briefly discuss some of the implementation level decisions we made with a justification as to why we made them.

4.1 bc2j: Implementation Choices

Traditionally instrumenting the code is done with access to the source code, so that annotations that correspond to the source program can be inserted. We decided to use bytecode as our input form, because we anticipate the many applications (applets) will be distributed without source. As argued before, it is good to profile such applications as well.

The profiled output on the other hand should be presented in a high-level form, so that the person using the visualization of the profiled information can reason about the analyzed applications in the terms we are used to. Therefore, we decided to use the compiler we are building which we plan to equip with multiple front-ends (we currently anticipate Java source and bytecode as input to the compiler, but possibly other languages could be used as well). The front-ends create a high-level intermediate representation which we call **JavaIR**. The back-ends can write JavaIR as any of the supported target languages.

We chose a modular structure of the compiler so that we can reuse the same optimization and instrumentation passes (since the intermediate representation is

9

shared by all combinations of front- and back-ends) no matter what source and target language combination we decide to use.

The compiler configured to use bytecode as input and Java source as output is called **bc2j** and can be thought of as a Java "de-compiler" with a possibility of inserting an arbitrary sequence of compiler passes which transform JavaIR. For this work only one such pass, namely the instrumentation is used.

4.2   Profiler: Implementation Choices

Ideally, profile data must be gathered in a way that causes little overhead beyond the time and space requirements for normal execution of the program being profiled. The most important consideration is that the execution time overhead of gathering profile data must be small enough to allow normal program operation. Coutant [2] define external instrumentation to be that which is done without modification to the program being instrumented such as sampling program counters with periodic interrupts. They define internal instrumentation to be the kind that is done with modification to the program on hand.

In gprof [4] the basic idea of profiling is to to record certain event counts (procedure calls) internally and externally sample and maintain program counter histograms to determine the total time spent in each basic block. The main problems associated with using exact times and doing everything internally are the fact that cost of record keeping could be prohibitive and that the clock resolution available maybe too small. However, in our approach the entire profiling is done internally partly by choice and mainly because we were forced to. By and large most of the Java applications available publicly on the web do not involve large iteration loops, multiple method invocations, highly recursive code etc. This observation led us to believe that explicit record keeping through internal profiling may not be a bad option especially for Java applications. Explicit record keeping includes maintaining all information in all calls across all object instances. Optionally at the users discretion the profiler can be made to maintain only summary information at either the object level or at the entire class level. Also without hooks in the Java interpreter we cannot really interrupt the interpreter and sample and maintain program counter histograms. Since the purpose is to have a widely available tool changing the interpreter is not a viable option. Furthermore since Java programs are being interpreted and much slower than corresponding C programs the time resolution problem may not be that much of a factor. This of course could vary from program to program.

Once we decided with this approach (optionally maintaining all profiles/summary of profiles) the next question was do we store profile information at an object instance by object instance fashion or do we have one common structure to manipulate all the profile information. We chose the former because of the simple data structure and better runtime performance during the execution of the program being profiled.

10

The memory requirements of both would be the same however since the objects are distributed printing out the relevant information could take slightly longer.

Next we discuss the static pass implementation issues.

Static Phase: Implementation Choices

```
class loops1 extends java.lang.Object
{
  public int i;
  public int j;
  public RProfile profile = new RProfile(7,"loops1");

  public void m1 () {
    int v0;
    profile.start_profile_method(0);
    profile.before_while_invoke_time(1);
    while((this.i == 0)) {
      profile.while_count(1);
      if((this.j == 0)) {
        profile.if_profile(2);
        this.i = this.j;
        profile.capture_invoke_time(3);
        global.m2();
        profile.capture_release_time(3);
        v0 = this.i;
      }else{
        profile.else_profile(2);
        this.i = this.j;
      } //if
    } //while
    profile.after_while_invoke_time(1);
    profile.end_profile_method(0);
    return;
  } // m1
}
```

Figure 4: Static Phase: Example

Figure 4 gives the resulting code after reverse compiling a bytecode file with the profiler pass. The commands added by the profiler pass can be identified by looking for the keyword "profile". As can be seen from the code each class instance has

11

associated with it an object of type profile. Every time a relevant event [2]occurs like the start of a while loop or the then part of the if branch is taken, there is a call to the relevant profile method. The parameter to these methods is the static reference to each block being profiled. Blocks are numbered as they appear in the code.

The instantiation of the profile object has as parameters the name of the class and the number of basic blocks being profiled in the class.

Runtime Library: Implementation Choices

The runtime library essentially consists of an implementation of the RProfile class (see figure 4). The RProfile class is instantiated with two parameters, the name of the class and the number of basic blocks being profiled. Both these are stored in private variables and a corresponding number of profile units are instantiated. When the first call to any profile unit (While, If-then-else, Methods) is made, the type of profile unit is initiated (this is encoded in the method call as can be seen from figure 4) and it starts storing events relating to that basic block in the code.

- In While loops we measure time from start to completion, the total number of iterations per call. From the figure it is easy to see that the profile statements before and after the while loop determine the total time, while the while count updates the counter. At the same time we keep a running summary of events (average execution time per iteration etc.).

- In If-then-else statements we simply update the if count if the then branch is taken and the else count if the else branch is taken.

- In Methods we measure time from start to completion, the method invocation time, and the release time. The start_profile_method and end_profile_method record the start and end times of each invocation. capture_invoke_time coupled with the start_profile_method helps record the invocation time and the end_profile_method coupled with capture_release_time records the release times. To handle calls across different objects or different instances of the same object, we have a set of global registers that can store start times and end times, and relevant callee and caller information so that the times can be stored in the appropriate profile units.

Profiler Performance

At time of writing we are currently evaluating the overhead of our profiling system. Preliminary results indicate that the overhead of profiling is not significant. Of course

---

[2]There are many optimizations that can be applied to minimize the number of these calls [5, 7]. For example if we know that the total number of while loop iterations is 50 and the else branch count is 20, then the if branch count must be 30. Such optimizations are currently not implemented.

this varies from program to program and depends on what segments [3] we choose to profile. We do not make any claims on the efficiency of our implementation as the goal of this paper was mainly to show that a tool like NetProf is viable and how it can be used.

4.3   Visualization Tool: Implementation Choices

It generates a GUI, which the user can click on relevant parts of the code and can see particularly interesting profile information. It preprocesses file.java (output from the static phase) and file.prof (output from the runtime phase). The preprocessing of file.java basically involves transforming it to a list of strings, parsing out profiler related code, and inserting flags to indicate whether the given list element is selectable or not. The preprocessing the file.prof involves storing the information in a data structure similar to ones used for storing sparse matrices. All the calls are stored in this data structure and there are instance indexes and basic block indexes (indexing which correspondence to the basic block in the source code) that let you index directly into a particular $< class_name >, < instance >, < basicblock >, and < callnumber >$.

On the user selecting a relevant profile (as described earlier) listing, it generates a dialog box through a call to Profile_Window, where the user can flip through several options. Once the user has made the choices on clicking the display button, all the selected information appears on the non-edit able text area.

5   Conclusions

We have presented a system called NetProf for visualizing high-level Java profile information over the Internet. We have described several parts that make up the whole system, each of which can be (modified and) used separately. bc2j is useful in that it recovers high-level constructs in the code without which certain profile information (loop level, control-level) would not be available. It also is easier to read than annotated bytecode produced by current Java compilers. The profiler is useful in stand-alone mode, it pinpoints high cost areas and determines what the programmer must concentrate on to improve performance. To facilitate the reading of profiler output we present a hierarchical visualization mechanism that makes reading the profiler data much easier.

---

[3]Eventually we want to be able to generate profiles for user specified segments and embed method/class instantiation parameters. After seeing the initial high cost areas the user can choose to restrict the profiling to only these areas andor subject to certain parameter values. The advantage is that the overhead of profiling would be greatly reduced and it could focus on the high cost areas. This feature is currently being implemented.

References

[1] T. Ball and J. R. Larus. Optimally Profiling and Tracing Programs. In *ACM TOPLAS*, July 1994.

[2] Coutant and others. Measuring the Performance and Behaviour of Icon Programs. *IEEE Transactions on Software Engineering.*, SE-9(1), January 1993.

[3] A. Freeman and D. Ince. *Active Java: Object-Oriented Programming for the World Wide Web.* Addison-Wesley Computer Science, 1996.

[4] S. Graham and others. An execution profiler for modular programs. In *Software: Practice and Experience, 13.*, 1983.

[5] J. R. Larus. Efficient Program Tracing. *Computer*, 26(5), May 1993.

[6] K. Pettis and R. C. Hanson. Profile Guided Code Positioning. In *SIGPLAN Notices ACM (25), PLDI*, June 1990.

[7] A. D. Samples. Profile Driven Compilation. In *PhD Thesis, UC-Berkeley, UCB-CSD-91-627.*, 1991.

[8] V. Sarkar. Determining Average Program Execution times and their Variance. In *SIGPLAN Notices ACM (24), PLDI*, June 1989.

[9] Sun Microsystems. The Java Language Specification. Available at http://www.javasoft.com/java.sun.com/doc/programmer.html.

[10] Sun Microsystems. The Java Language Tutorial. Available at http://www.javasoft.com/java.sun.com/doc/programmer.html.

[11] Sun Microsystems. The Java Virtual Machine Specification. Available at http://www.javasoft.com/java.sun.com/doc/programmer.html.