ProfBuilder: A Package for Rapidly Building Java Execution Profilers

Brian F. Cooper, Han B. Lee, and Benjamin G. Zorn

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430 USA

Telephone: (303) 492-4398
FAX: (303) 492-2844
E-mail: {cooperb,hanlee,zorn}@cs.colorado.edu

CU-CS-853-98 April 1998

University of Colorado at Boulder

Technical Report CU-CS-853-98
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430 USA

Telephone: (303) 492-4398
FAX: (303) 492-2844
E-mail: {cooperb,hanlee,zorn}@cs.colorado.edu

ProfBuilder: A Package for Rapidly Building Java Execution Profilers*

Brian F. Cooper, Han B. Lee, and Benjamin G. Zorn

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430 USA

Telephone: (303) 492-4398
FAX: (303) 492-2844
E-mail: {cooperb,hanlee,zorn}@cs.colorado.edu

April 1998

Abstract

ProfBuilder is a collection of Java classes that provide an easy method for constructing Java execution profiling tools. By instrumenting Java bytecode, a programmer can measure dynamic properties of an application, such as bytecode count, number of memory allocations, cache misses and branches. ProfBuilder provides an easy way to create customized tools to measure these and other properties by writing a small amount of code. Although commercial profiling tools for Java available, there are currently few, if any, Java profiler generators. In this paper, we describe the Prof Builder meta-tool as well as two tools built using the package, a bytecode instruction profiler and a memory allocation profiler. We also describe our experience using these tools on programs, including the overhead of instrumentation and profiling. Finally, we discuss the results of using the instruction profiler to optimize the execution time of a Java program, increasing its performance 15% with minor changes to the code.

*This research is supported in part by NSF Grants CCR-9711398 and IRI-9521046.

1 Introduction

ProfBuilder is a software package for rapid construction of tools used to measure various aspects of Java program behavior. The package provides a meta-tool for constructing custom profilers, as well as a data structure for modeling program execution. A user can build a tool, instrument an application with a tool, and gather dynamic execution data by writing just a few lines of code. This paper describes the design and implementation of ProfBuilder and describes two specific examples of tools constructed using it.

The dynamic characteristics of a program determine how the program actually behaves during execution. Examples of dynamic behavior include cache misses, branches taken, and memory allocated. Examination of this behavior can provide insights into the efficiency of algorithms, allow identification of performance bottlenecks, and enhance understanding of resource allocation issues. By gathering dynamic execution data, a programmer can easily identify code and algorithms that can be optimized for speed, memory usage, and correctness.

Java is an ideal domain for gathering and using dynamic execution data. The ease of programming in Java coupled with the platform independent nature of the language has encouraged many developers to build Java applications. However, as an interpreted language, many Java implementations are slow to execute. It is therefore especially important that the code be as efficient as possible so that the performance is acceptable. These factors suggest that Java applications are a good target for optimizations based on dynamic execution data.

The concept of execution profiling has a long history and has been widely available in successful tools, such as gprof [3], which generates a hierarchical profile of an application. ProfBuilder is a meta-tool for building sophisticated tools in the style of gprof with a small amount of effort. ProfBuilder provides a fast and easy way to create a variety of profiling tools with a small amount of code. ProfBuilder uses the Bytecode Instrumenting Tool (BIT) invented by Lee and Zorn [8, 7] to instrument compiled Java programs. The central data structure of ProfBuilder is the Calling Context Tree (CCT) [1], which is used to model the execution of an application and to store arbitrary measurements of dynamic behavior, such as branch outcomes or memory allocations.

In this paper we describe the CCT, BIT, ProfBuilder, and two tools created using ProfBuilder: IProfTool (a bytecode instruction profiler) and MProfTool (a memory allocation profiler). We discuss performance issues related to using the tools to instrument programs and show data on various Java applications. We also discuss the results of using IProfTool to optimize JLex [2], a lexical analyzer generator. An examination of the bytecode instruction profile for JLex allowed us to identify a significant performance bottleneck, and upon reimplementation of a single procedure, the execution time of the application was cut by 15%.

This paper is organized as follows. Section 2 describes the CCT and BIT in more detail. Section 3 provides a demonstration of one CCT profiling tool, the IProfTool, and walks through the process of building tools,

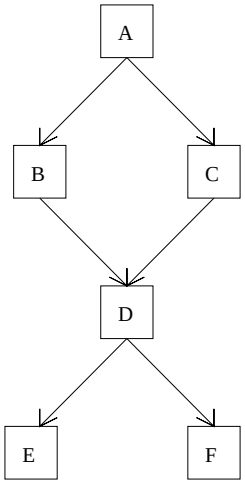

Figure 1: A sample call graph.

instrumenting programs and collecting data. Section 4 describes how profiling tools are built in more detail, and Section 5 describes the implementation of the CCT. Section 6 describes experimental experience using the CCT profiling tools to instrument various programs, and Section 7 describes related work. In Section 8, we present our conclusions.

2 Background

In order to fully understand ProfBuilder, it is first important to understand its two underlying technologies, the CCT and BIT. Section 2.1 describes the Calling Context Tree, and Section 2.2 describes BIT.

2.1 Calling Context Tree

Ammons, Ball and Larus [1] describe three data structures for recording the runtime behavior of programs, the dynamic call graph, the dynamic call tree, and the new structure they propose, the Calling Context Tree. In a dynamic call graph, each subroutine is represented as a vertex, and calls are represented as edges. The graph compactly represents the dynamic structure of the program. In fact, the compactness is one of the major advantages of the call graph; the number of nodes is equal to the number of distinct subroutines. Thus, deep recursion chains or repeated invocations of the same procedure do not require any more space. Unfortunately, this benefit comes at a price. A great deal of information is lost about any call chains longer than two routines. For example, Figure 1 shows one example call graph. In the graph, procedures `B` and `C` both call procedure `D`, which in turn calls procedures `E` and `F`. From this graph, it appears that the call chains `BDE`, `BDF`, `CDE` and `CDF` all occurred during program execution. However, if `D` only calls `E` upon invocation

from `B`, the call chain `BDF` never really occurs. This information is lost in the call graph, since it only records the ancestors and descendents of each subroutine, and not actual existent call chains.

This information loss does not occur with the second data structure. The dynamic call tree preserves all information about the call structure of a program. A call tree has vertices that represent individual invocations of routines. Thus, the same routine invoked five times will be represented as five nodes in the tree. This structure preserves the most information, however, it has a significant disadvantage in that it requires a great deal of space for storage. Any programs with significant recursion or numerous subroutine calls within loops would certainly strain the memory of the computer that stores the tree.

The third data structure, the Calling Context Tree, captures the advantages of the call graph and the call tree while attempting to avoid their disadvantages. The CCT is similar to the dynamic call tree except that vertices represent individual contexts rather than individual procedure activations. A context is a procedure coupled with the call chain that resulted in the call to that procedure. Thus, repeated invocations of a procedure are stored in the same node as long as the call stack is the same each time. In addition, a recursive call is represented as a backedge, so that long recursion chains are compactly stored as a loop in the tree. Every unique call path is represented by a distinct node in the tree, thus providing more information than the call graph, while avoiding the memory requirements of the dynamic call tree.

The usefulness of the CCT lies in the ability of a programmer to associate arbitrary metrics with nodes in the tree. For example, in a specific context, a program can cause a number of cache misses, and this number can be attached to the CCT node representing that context. Alternately, the number of branches taken or memory allocations in the context can also be attached to a CCT node. Because any measurement of dynamic behavior can be connected to nodes representing contexts, it is easy to relate the dynamic context to the observed program behavior.

2.2 Bytecode Instrumenting Tool (BIT)

BIT is a library of Java classes that allow instrumentation of Java Virtual Machine (JVM) class files for the purpose of extracting measurements of their dynamic behavior. BIT was built based on the observation that being able to create customized tools to observe and measure the run-time behavior of programs is valuable for many tasks including program optimization and system design.

BIT is one of the binary editing or executable editing tools that offer a library of routines for modifying executable files. Other tools in this group include the OM system [12], EEL [5], and ATOM [11]. These tools operate on object codes for a variety of operating systems and architectures while BIT works on JVM class files.

BIT hierarchically decomposes the JVM bytecodes in a class file into different entities including methods, basic blocks and bytecode instructions, and provides facilities for navigating through these entities. Moreover,

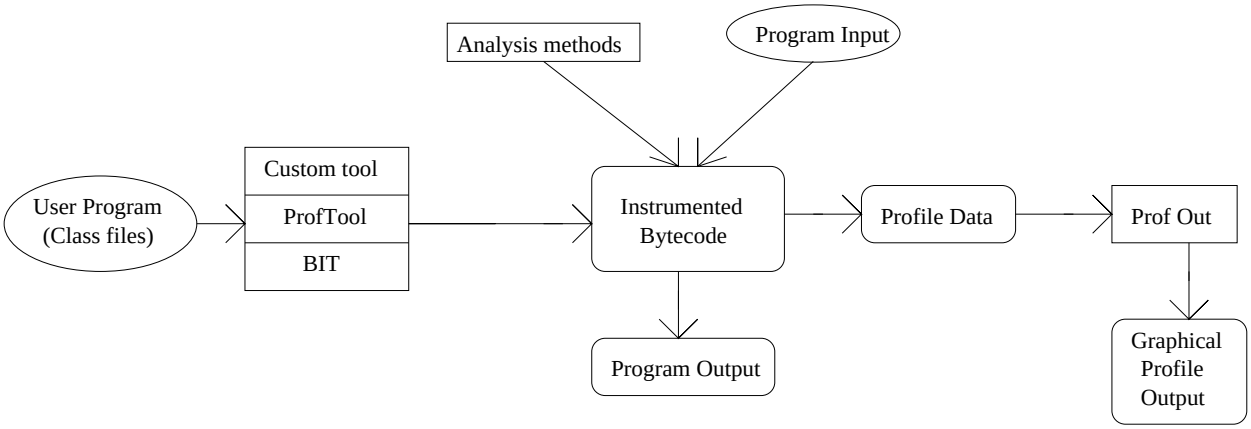

Figure 2: The instrumentation and data collection process. Rectangles represent components of the ProfBuilder package, ovals represent user supplied inputs, and rounded rectangles represent outputs.

BIT allows the user to insert calls to analysis routines before and after each one of these entities. Using BIT, users can create customized tools that can be used to extract specific dynamic information about the JVM class files by writing instrumentation code, which specifies how to insert calls (typically by iterating through certain entities and deciding whether to add before or after each of these). In addition to the instrumentation code, users write analysis code that is invoked as a result of instrumentation and specifies what computation or measurements are to be carried out when the program is run. For example, the operations performed might include collecting profile information during program execution and writing the profiles to a file when the program completes.

One important property of the instrumentation that is added using BIT is that the analysis routines do not have any semantic effect on the instrumented program. BIT is the first framework that allows users to create customized tools to analyze JVM bytecodes quickly and easily. Furthermore, because BIT works on bytecodes, instrumentation works on programs written in any language that can be compiled into JVM bytecodes and does not require that the program source be available.

3 Using ProfBuilder to Create Profiling Tools

ProfBuilder provides a set of classes that are used to create profiling tools. The complete system is illustrated in Figure 2. As the figure illustrates, the core of any tool is class `ProfTool`, which selects class files for instrumentation and inserts calls into the classes to accomplish the construction of the Calling Context Tree. After the class files of the user program have been instrumented, the program is run on the inputs of interest, and in addition to generating its original output, the instrumentation also generates profile data. This profile data is essentially an externalized version of the Calling Context Tree that can be visualized in a number of

different ways. Bundled with ProfBuilder, we provide `ProfOut`, which is a program that presents a text-based visualization of this data as illustrated in this paper. Other more sophisticated graphical visualizations are also possible.

In order to build a new tool, a user simply has to create a subclass of `ProfTool` and override one function, `instrumentOneClass`. When the tool runs, it iterates through a set of class files in a directory, instrumenting each one to build the CCT and then calling `instrumentOneClass` to insert tool-specific instrumentation.

ProfBuilder also includes class `CallingContextTree`, a Java implementation of the CCT. This class allows a user to define how many different metrics can be stored at each node in the tree, and provides operations for building the tree based on the call stack. These operations are invoked during the execution of the application by the instrumented code. When the application exits, a record of the tree and the recorded metric values are saved in a data file.

Using this framework, it is possible to build a variety of tools. This section illustrates the process of building and using profiling tools by describing one such tool, a Java bytecode instruction profiler called `IProfTool`. `IProfTool` counts the number of bytecode instructions executed in each context; this information can be used to determine which contexts are incurring the most overhead in terms of execution time.

3.1 Building the instruction profiler

CCT profiling tools must do two things. First, they must instrument the program to build the CCT during program execution. Second, they must instrument the program to collect useful data about the execution. The first task is performed in the same way for all profiling tools using the CCT. For this reason, the functionality has been encapsulated in the `ProfTool` class, the base class from which all other profiling tools are descended. It provides the interface that allows the user to specify a set of class files to be instrumented, loads each class file into a BIT `ClassInfo` structure, and inserts the necessary calls to build the tree. Thus, the first step in building a tool is simply to declare a class that extends ProfTool.

This extension class performs the second task, which is gathering and collecting the specific information of interest at runtime. In our example, the bytecode instruction profiler, `IProfTool`, gathers information about the number of bytecodes executed in each context. In order to accomplish this, the tool consists of two parts. The first part, shown in Figure 3, is the instrumentation component of the profiler. It consists of two methods, `main` and `instrumentOneClass`. `main` is the entry point for the program. It instantiates a new tool, and then calls `instrumentClasses`, a method inherited from `ProfTool`. `instrumentClasses` takes as a parameter a pointer to the command line arguments, and parses those arguments to find the directory containing the class files to be instrumented as well as the directory to place the instrumented files in. This method loads each class file, and then calls `instrumentOneClass`, the second method in Figure 3. `instrumentOneClass` is called for every class being instrumented, and it is here that `IProfTool` performs the

```
public class IProfTool extends ProfTool {
   public static void main(String[] argv) {
        IProfTool ipt=new IProfTool();
        ipt.instrumentClasses(argv,2);
   }

   public void instrumentOneClass(BIT.highBIT.ClassInfo ci) {
        nameMetric(ci,1,"Instructions");
        for (Enumeration e=ci.getRoutines().elements(); e.hasMoreElements(); ) {
            BIT.highBIT.Routine routine= (BIT.highBIT.Routine) e.nextElement();
            for (Enumeration f=routine.getBasicBlocks().elements();
                 f.hasMoreElements(); ) {
                 BIT.highBIT.BasicBlock bb=(BIT.highBIT.BasicBlock)f.nextElement();
                 bb.addBefore("IProfTool", "trackInstructions", String.valueOf(bb.size()));
            } //end of basic blocks iteration
        } //end of routine iteration
    }
}
```

Figure 3: The instrumentation component of IProfTool.

```
public static void trackInstructions(String count) {
    CCT.AddToMetric(1,Integer.parseInt(count,10));
}
```

Figure 4: The analysis component of IProfTool.

tool-specific work. The purpose of `instrumentOneClass` is to navigate all the basic blocks in the program being instrumented and insert calls to the analysis routine, `trackInstructions`, before each basic block is executed. In the example, this navigation is accomplished using the BIT operations `getRoutines`, an iterator that returns successive routines in the class file; and `getBasicBlocks`, an iterator that returns successive basic blocks in each routine. The call to the `bb.addBefore` method instructs BIT to add a call to the routine `trackInstructions` before the program executes the current basic block. The parameter passed to `trackInstructions` is the size of the current basic block (returned by the call to `bb.size`) converted to a string.

The profiler consists of one analysis routine, `trackInstructions`, shown in Figure 4. The instrumentation phase inserts calls to this method into the bytecode of the instrumented classes, and these calls pass the number of bytecodes in the basic block to the analysis routine. The routine adds that number to the metric for the current calling context. This context is stored in CCT, which is the calling context tree member data constructed by `ProfTool`. In this way, by calling `AddToMetric`, the metric data is automatically associated with the current context.

```
import java.io.*;
class Test {
    public static void main(String[] argv) {
    System.out.println("main()");
    for (int i=0; i<10; i++)
        A();
    }

    public static void A() {
        System.out.println("A()");
        for (int i=0; i<10; I++) {
            B();
            C();
            B();
        }
    }

    public static void B() {
        System.out.println("B()");
        D();
    }

    public static void C() {
        System.out.println("C()");
        D();
    }

    public static void D() {
        System.out.println("D()");
    }
}
```

Figure 5: Example program.

The correct functioning of `IProfTool` is dependent on the correct interaction between the instrumentation and analysis components of the tool. The instrumentation places calls to the analysis routines in the appropriate places, and the analysis routine collects the data. The `ProfTool` functionality performs the other tasks related to instrumentation and analysis, including writing out the instrumented class file, building the CCT, and writing out the CCT when the program exits.

3.2 Instrumenting a program

Once the profiling tool is built, the next step is to instrument the program to be studied. Figure 5 lists a very simple program that will illustrate this process. Figure 6 shows the program's static call graph. This program calls its methods `A`, `B`, `C`, and `D` several times. Note that there are two different paths from `A` to `D`,

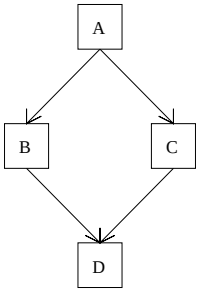

Figure 6: The call graph of the example program

and the path {`A`, `B`, `D`} is taken twice as often as {`A`, `C`, `D`}. This difference will appear in the output from the CCT and profiling tool.

In order to instrument this program, it is first compiled with a Java compiler, and the class file it produces is stored in a directory (in this example, called `source`). The bytecode instruction profiling tool is then run on the compiled bytecode, and the result stored in another directory (called `dest`) with the following command that invokes the Java interpreter:

```
java IProfTool source dest
```

The tool automatically instruments every routine of every class file in the source directory.

3.3 Gathering profile data

The instrumented code is now ready for data collection. Since the analysis methods of the `IProfTool` class will be called by the instrumented class files, it is important that the directory containing `IProfTool.class` be in the CLASSPATH of the environment. In order to collect data, the program is run normally,

```
java Test
```

and when the execution is finished, the file `Test.Prof` is created in the `dest` directory. This file is a serialized Calling Context Tree and can be loaded by another program using the appropriate `CallingContextTree` method.

3.4 Viewing the data

The data that has been gathered can be displayed in any appropriate format. We have written a tool that loads this data and displays it graphically by showing the structure of the tree as well as a histogram indicating the value of each metric. The program, `ProfOut`, is run with the following command line:

```
java ProfOut Test.PROF
```

and writes the output shown in Figure 7 to `stdout`. The section of the output entitled "Total Metrics" shows the sum for all of the metrics over the entire execution of the program. The "Metrics local to procedures" section displays values of metrics for each context in the program, while the "Metric totals for subtrees" section sums the metrics for each subtree and displays the sum at the node that roots that subtree.

The output captures the structure of the Calling Context Tree as well as the values of the metrics at each context. On the left side of the diagram is the numerical value of the metric for each context, with a horizontal bar that graphically displays that value. The length of the bar represents the value for each context as a fraction of the maximum value for any context. The right of the output indicates the context in which that metric data was gathered. Specifically, the indentation is used to indicate the caller/callee relationship; children of the same parent are aligned at the same indentation. For example:

```
parent
  child1
  child2
```

The output also displays any recursion. Because the CCT handles recursion by introducing backedges into the tree, the recursive child of a function is represented as a pointer to the previous node for that function in the tree. The display indicates the recursion with an R:, as in

```
parent
  R: parent
```

The display does not traverse the backedge, and merely indicates that it exists. Thus, if parent recursively calls itself to a depth of ten, this is indicated as ten calls to parent rather than a display with the word "parent" printed ten times.

The output in Figure 7 indicates that the number of calls to `D` in the context {`main`, `A`, `B`, `D`} is twice as large as the number of calls to `D` in the context {`main`, `A`, `C`, `D`}, as indicated by the larger bars on the left hand side for that context as well as a larger value for the metric (indicated in parentheses) on the left of the display. This conclusion is consistent with the prediction that we made earlier. The output also indicates that the largest numbers of bytecode instructions were executed in the `B` function in the context {`main`, `A`, `B`}.

4 Building Tools

ProfBuilder is designed to allow an easy interface to the CCT data structure and rapid construction of profiling tools. It provides an API that allows users to build and store information in the tree. By extending

```
Total metrics
Calls: 611
Instructions: 3560

================================================================

Metrics local to procedures

----------------------------------------------------------------
Calls
----------------------------------------------------------------
(       1)                                        | Test.main
(      10) **                                     |   Test.A
(     200) ****************************************|     Test.B
(     200) ****************************************|       Test.D
(     100) ********************                     |     Test.C
(     100) ********************                     |       Test.D
0                                                200
----------------------------------------------------------------
Instructions
----------------------------------------------------------------
(      60) **                                     | Test.main
(     800) ********************************       |   Test.A
(    1000) ****************************************|     Test.B
(     800) ********************************       |       Test.D
(     500) ********************                     |     Test.C
(     400) ****************                         |       Test.D
0                                               1000
----------------------------------------------------------------

================================================================

Metric totals for subtrees

----------------------------------------------------------------
Calls
(     611) ****************************************| Test.main
(     610) ***************************************  |   Test.A
(     400) ***************************              |     Test.B
(     200) *************                            |       Test.D
(     200) *************                            |     Test.C
(     100) ******                                   |       Test.D
0                                                611
----------------------------------------------------------------
Instructions
(    3560) ****************************************| Test.main
(    3500) ***************************************  |   Test.A
(    1800) ********************                     |     Test.B
(     800) ********                                 |       Test.D
(     900) **********                               |     Test.C
(     400) ****                                     |       Test.D
0                                               3560
----------------------------------------------------------------
```

Figure 7: IProfTool output for example program

class ProfTool, numerous custom tools can be constructed. This section describes in detail the issues related to building arbitrary tools using ProfBuilder.

4.1 ProfTool API

Class ProfTool provides a set of methods that are common to all profiling tools built with ProfBuilder. Figures! 8 lists the important parts of the ProfTool class API. The most important functions are the ones that profiling tools can override to create the individual tool. The instrumentOneClass method is called for every class that is instrumented by the tool. In the ProfTool class, this method does nothing, so an instance of class ProfTool could be used to build the CCT without any extra profiling functionality. Subclasses of ProfTool override the instrumentOneClass method to instrument each class, inserting calls to the analysis routines unique to that tool. A user can completely specify a tool by simply overriding this one function and a main function to instantiate the tool. The main function, after creating an instance of the tool, calls instrumentClasses to instrument all classes specified in the path in the command line, outputting modified class files to the path in the command line and generating a CCT with the number of metrics specified by the argument nummetrics. This number includes metric 0, which is used by class ProfTool to count the number of calls to each procedure.

ProfTool also introduces flexibility in the form of other methods that can be overridden. The StartUp method is called before any classes are instrumented, and a user can override this function to initialize the tool in any way necessary.

A tool can also have command line switches. By default, the ProfTool class provides the following standard switches:

- -c to activate instrumentation to distinguish between distinct call sites (described in Section 4.3).
- -x to search the bytecode for System.exit calls and instrument them to save profile data before exiting the program. The default is to save the data only when the main function exits.
- -v to activate verbose mode instrumentation.

These switches are parsed in the instrumentClasses method, which should be passed the entire command line array argv. For each switch in the command line, ProfTool attempts to parse it as -c, -x or -v. If it cannot, it calls the method parseOneArgument. The subclass of ProfTool can override this method to attempt to parse argv[index], returning the index of the next unparsed argument if successful or index if not. This allows the subclass to add any additional switches as necessary.

If the tool cannot parse the command line switches, or if there are two few arguments on the command line (the minimum is a source path and destination path) then the tool calls outputUsageMessage. ProfTool

```
public class ProfTool {
    protected static CallingContextTree CCT=null;
    //The data member holding the Calling Context Tree

    protected boolean InstrumentCallSites=false;
    //Distinguish between call sites on true

    protected boolean Verbose=false;
    //Output messages during instrumentation

    protected boolean InstrumentExit=false;
    //Search bytecode for System.exit and instrument it to
    //save the profile data before exiting

    public static void main(String argv[]);
    //The entry point of the tool program

    public void instrumentClasses(String argv[], int nummetrics);
    //Search through the path specified in the argv command line, selecting
    //class files, and instrumenting each file, outputting to the path
    //specified in argv. nummetrics specifies the number of metrics
    //recorded by the tool

    public boolean parseCommandLine(String argv[]);
    //Parse the tool command line for switches

    public void nameMetric(BIT.highBIT.ClassInfo ci, int which, String name);
    //Associate a metric number with a name in a particular class file

    public void addBeforeMain(BIT.highBIT.ClassInfo ci, String classname,
                              String methodname, Object arg);
    //Add instrumentation code before the Main function to be executed upon
    //instrumented application startup

    ////////Methods that specific tools override ////////////////////

    int parseOneArgument(String argv[], int index);
    //Tools override this function to parse a single argument
    //at argv[index]. Returns a value > index which is the next
    //argument to be parsed, or a value == index to indicate
    //no value was parsed

    public void instrumentOneClass(BIT.highBIT.ClassInfo ci);
    //Tools override this function to instrument a single class

    void outputUsageMessage();
    //Message to output to the user who has invoked the tool
    //with an incorrect command line argument

    void StartUp();
    //Called when any ProfTool descendent starts up, before any
    //instrumentation occurs
}
```

Figure 8: The API for the ProfTool Base Class

provides a standard usage message, but subclasses should override this function to provide the tool user with any necessary additional information.

4.2 CallingContextTree API

The `CallingContextTree` class provides three types of methods for users: constructors, tree building, and metrics methods. Figure 9 lists the complete API. Users can construct the tree as an empty structure using the `CallingContextTree` constructor. The parameter determines the number of metrics in the tree. The `inputTree` method reads in the tree from a serialized `ObjectInputStream`. The tree can be written out to an `ObjectOutputStream` using the `Output` method. This can be done at arbitrary times, and the tree can be saved in different files by specifying different `ObjectOutputStreams`. In this way, a user could save a snapshot of the CCT at a particular point during execution. By default, `ProfTool` writes out the tree when the instrumented program exits.

The tree is built using the `Enter` and `Exit` methods. When a routine is called, it should be instrumented to invoke `Enter` with the name of the routine as the parameter. The `CallingContextTree` will generate a new connection in the tree from the parent routine to the child routine if such a connection does not already exist. The routine is then instrumented to call `Exit` upon exiting. This is necessary to preserve the internal state of the tree. Since these operations are performed by `ProfTool`, all tools that extend `ProfTool` will automatically encapsulate this functionality.

The `CallingContextTree` class also provides a set of methods for dealing with metrics. The user can create as many metrics as the memory of the system can support, and decides upon the assignment of each metric to a meaningful quantity (such as 0: Calls, 1: Instructions, etc.). Each metric can be named for later reference by the `NameMetric` method, which associates a string name with a numeric metric. In turn, the name of a metric can be found using the `GetMetricName` method.

The values of metrics are changed using `AddToMetric`, which adds a positive or negative value to a metric, or `SetMetric` which sets the value of a metric. These functions define the interface that allow the profile tool to manipulate the metrics associated with the currently active calling context. In addition, users can manipulate detail metrics, which allow the information for a metric to be further subdivided into categories. As an example of how to use detail metrics, consider that in `IProfTool` metric 1 is used to record the number of bytecodes executed. Detail metrics could be used in this case to categories the instruction count into memory instructions, arithmetic instructions, etc. `AddToDetailMetric` is used to increment the detail metric associated with the current calling context. `GetDetailMetric`, `GetTotalMetric` and `GetTreeMetric` return the current value of a detail metric, a total metric for the whole tree, and a metric tied to the current tree node, respectively.

```
public class CallingContextTree implements Serializable
{
    CallingContextTree(int _NumberOfMetrics);
    //Constructor to create an empty tree. The number of metrics
    //it can store at each node is specified by _NumberOfMetrics

    public static CallingContextTree inputTree(ObjectInputStream instream);
    //Loads a tree from an ObjectInputStream

    public void Output(ObjectOutputStream outstream);
    //Serialize the data structure on an ObjectOutputStream

    public void Enter(String name);
    //Enter a new procedure, creating a new node in the tree
    //if necessary

    public boolean Exit();
    //Exit a procedure. Return true if it is
    //the root of the tree that exited.

    /////////////////METRICS////////////////////////////////

    public int GetNumberOfMetrics();
    //Return the number of metrics that can be stored at each node

    public VariableList GetTotalDetails(int metricnumber);
    //Get the total value of a metric summed over all contexts

    public int GetTreeMetric(int metricnumber);
    //Get the current value of a tree metric (a metric attached to a
    //particular node)

    public void AddToMetric(int metricnumber, int newvalue);
    public void SetMetric(int metricnumber, int newvalue);
    //Change the value of a metric

    public int GetDetailMetric(int metricnumber, String metricname);
    //Get the current value of a detail metric

    public void AddToDetailMetric(int metricnumber, String metricname, int newvalue);
    public void SetDetailMetric(int metricnumber, String metricname, int newvalue);
    //Change the value of a detail metric

    public void NameMetric(int metricnumber, String name);
    //Associate a name with a metric

    public String GetMetricName(int metricnumber);
    //Get the name of a metric
}
```

Figure 9: The API for class CallingContextTree.

4.3 Recording call sites

In constructing a profile, it may be useful to distinguish calls to the same procedure from different call sites. For example, if method `A` calls method `B` twice, each time from a different point in method `A`, then `B` has been invoked from different call sites. For some purposes, these different activations should be regarded as different calling contexts. The `ProfTool` class provides functionality for this distinction.

The member variable boolean `InstrumentCallSites` controls this process. When this variable is set to true, `ProfTool` instruments procedures to record call site information. It does this by searching for invoke bytecodes, and inserting a call to an analysis routine before each invoke. The analysis routine saves the call site number for use by the CCT. The call site number is dependent only on the number of invoke bytecodes in the routine; the first invoke is call site 1, the second is call site 2, and so on, even if they invoke different routines. Thus, if `A` calls `B` from its third and fifth invoke bytecode, `B` will be stored in the CCT as `B.3` and `B.5`.

This functionality can be activated by setting the `InstrumentCallSites` variable to true inside a class that descends from `ProfTool` by overriding the virtual `StartUp` method, which is called when the tool first begins to execute. Alternately, the `-c` command line switch, which is parsed by `ProfTool`, can activate the call site recording. This switch allows users of profiling tools to determine when they want call site information.

4.4 Extending IProfTool to MProfTool

`ProfTool` contains the basic functionality that all tools will need to have, such as building the tree. `IProfTool` contains extra functionality to count bytecodes. This extra functionality represents task-specific code that makes each tool perform its specialized purpose. Instead of counting bytecodes, for example, the tool can be designed to count only `new` bytecodes, creating a tool that counts object allocations and thus profiles the memory allocation of the instrumented program. We built `MProfTool`, a memory allocation profiler, that counts `new` bytecodes as well as `newarray`, `anewarray`, and `multianewarray` bytecodes.

The instrumentation part of `MProfTool` iterates through the class file, examining every bytecode. When a `new` bytecode is found, a call to `trackInstructions` is inserted before the instruction, and the type of bytecode (new, newarray, etc) is passed to the `trackInstructions` routine. This analysis routine increments the appropriate metric (1: new, 2: newarray, 3: anewarray and 4: multianewarray).

`MProfTool` also records the types of objects allocated, which is done by static analysis of the bytecode. The BIT package provides an `Instruction.getOperandValue` method that returns the operand of the instruction. Since the operand of a `new` bytecode is an index into the class's constant pool, it can be used to

```
------------------------------------------------------------------
newarray
------------------------------------------------------------------
  Total details:
   [char]: 113
   [byte]: 1
   [long]: 6557
   [int]: 81
------------------------------------------------------------------
(       0)                      | JLex/Main.main
(      29)                      |   JLex/CLexGen.<init>
                                |      =>[char]: 29
(       2)                      |     JLex/CInput.<init>
                                |        =>[byte]: 1
                                |        =>[char]: 1
(       4)                      |     JLex/CSpec.<init>
                                |        =>[char]: 4
(       0)                      |     JLex/CNfa2Dfa.<init>
...
(       0)                      |               JLex/JavaLexBitSet.get
(    4994) ********************|             JLex/JavaLexBitSet.<init>
                                |                =>[long]: 4994
...(remaining output deleted)
```

Figure 10: Sample MProfTool output for JLex.

determine the name of the object allocated, and that name can be passed to `IncDetailMetric`, for example as `IncDetailMetric(1, "String")` to record that a string was allocated by new (metric 1).

`MProfTool` is useful for understanding the memory allocation behavior of a program. Figure 10 shows a portion of the output that resulted from using `MProfTool` to instrument the JLex application (described further in Section 6). The figure indicates the output for the newarray instruction, which creates arrays of primitive Java types. For each context (listed on the right) there is an associated number of calls to `newarray` (listed on the left) along with a breakdown of the types of arrays created (under the name of the procedure in which the allocation occurred). Thus, `JLex/CInput.<init>` called `newarray` twice, once to allocate an array of byte and once to allocate an array of char. At the top of the diagram, the total number of arrays of each type is listed. During this run, 113 char arrays were allocated, 6,557 long arrays were allocated, and so on.

Other tools, such as a cache miss profiler or a conditional branch counter, can be built in similar ways. The `ProfTool` class is used as a framework, and the appropriate places in the bytecode are instrumented to call analysis routines that increment the appropriate metric.

5 Calling Context Tree Implementation

Although the Calling Context Tree data structure is designed to work with BIT in instrumenting programs, it can also be used independently of BIT. That is, a program can use a `CallingContextTree` directly, without also using BIT. This functionality can be useful for loading and manipulating a CCT that has been created by an instrumented application. Here we describe the implementation of class `CallingContextTree` in Java.

A `CallingContextTree` is a collection of `CallRecords`. A `CallRecord` represents an invocation of a routine in a particular calling context. It holds a representation of the name of the procedure, and references to its parent in the tree as well as its children.

The `CallRecord` also stores metrics that are specific to a particular context. It has two kinds of metrics: regular metrics and detail metrics. All call records have the same number of regular metrics, and these are used to store basic information about the program. In the example in Section 3, the number of bytecodes was stored in metric number 1. `ProfTool` reserves metric 0 to count the number of calls to each procedure. There are also detail metrics, which record more specific details on a `CallRecord` by `CallRecord` basis. For example, in `MProfTool`, which counts objects allocated by a `new` bytecode, detail metrics are used to store the types of objects allocated. A particular routine may have called 50 `new` instructions, and in doing so, allocated 30 strings, 10 integers and 10 vectors.

The `CallRecord` has several static members. A dictionary stores all of the names of the procedures and maps them to integers. Individual call records store the name of the associated procedure as an integer that indexes into the dictionary. There is also a stack that is used as an internal copy of the call stack, with `CallRecords` pushed and popped on the stack. This is necessary to determine the parent of the currently called routine and to restore the `CallRecord` of that parent as the current call record when the routine exits.

The `CallingContextTree` provides an interface to the `CallRecords` that are linked in tree fashion. Thus, it provides methods for tree construction, dealing with metrics, and loading and saving the tree to a file. It also stores its own set of regular and detail metrics, which represent totals for the whole tree. Whenever a metric is incremented in a particular context, it is also incremented in the total metrics.

6 Performance and Experience

This section describes the results of using `IProfTool` on a set of Java applications. Section 6.1 discusses the overhead that results from the CCT profiling. Section 6.2 describes our experience using `IProfTool` to optimize JLex, a lexical analyzer generator.

Program	Description	Uninstrumented		Number of Class Files
		Execution (secs)	Class File Size (Kb)	
JLex	Lexical analyzer generator	6.9	76.7	20
aster	Asteroids game	14.7	18.6	7
espresso	Java compiler	5.5	295.3	105
java_cup	Parser generator	1.6	117.6	41

Table 1: Overview of Applications Measured

Program	Instrumented with ProfTool			Instrumented with IProfTool		
	Time to Instr. (sec)	Execution Time (sec) / % Increase	Class File Size (Kb) / % Increase	Time to Instr. (sec)	Execution Time (sec) / % Increase	Class File Size (Kb) / % Increase
JLex	6.0	35.8 / 421%	85.1 / 11.0 %	35.0	16.3 / 2290 %	108.8 / 41.9%
aster	1.2	15.4 / 5.0%	21.5 / 15.5 %	3.9	21.2 / 43.5%	26.3 / 40.8 %
espresso	25.9	141.5 / 2454%	340.5 / 15.3 %	106.4	289.3 / 5118%	422.5 / 43.1%
java_cup	8.7	18.8 / 1058 %	141.8 / 20.6%	40.9	18.8 / 1059 %	176.9 / 50.4 %

Table 2: Overhead for running ProfTool and IProfTool on various programs. The values for `ProfTool` represent the base overhead from constructing the Calling Context Tree. Additional instrumentation, such as counting bytecodes (as `IProfTool` does) adds overhead.

6.1 Profiling overhead

Because `IProfTool` inserts calls to analysis routines at every basic block in a compiled bytecode, it can add a significant amount of overhead to the running program. `IProfTool` was used to instrument several programs in order to measure the overhead. These programs are JLex, a lexical analyzer generator [2], aster, an Asteroids space game (also used in [10]), espresso, a Java compiler [9], and java_cup, a "Constructor for Useful Parsers" [4]. Table 1 summarizes each application, including execution time and compiled bytecode size.

The applications were compiled and run using Sun's JDK version 1.1.5 on a machine with a 233 Mhz Pentium II with 64 MB of RAM. The JDK and applications were run under Windows NT version 4.0. All data presented is the average over 10 runs of each application. The variation in execution time between runs was insignificant.

Table 2 summarizes the overhead for instrumenting and running `ProfTool` and `IProfTool` on these applications. The table lists the time to instrument each application, the execution time and class file size of the instrumented program, and the corresponding increase for execution time and file size over the

uninstrumented version. `ProfTool` builds the CCT only; therefore, the `ProfTool` data represents the profiling overhead independent of any specific measurements, such as bytecode counting in `IProfTool`.

The most significant disadvantage of using ProfBuilder is the increase in program execution time. As Table 2 indicates, the overhead for building the CCT is very high (over 2000%) for large programs such as espresso, and the additional instrumentation in `IProfTool` causes even more overhead. Optimizing the `CallingContextTree` methods, which perform most of the runtime profiling computation, can reduce this overhead. This would reduce the time spent inside `CallingContextTree` methods, but little can be done to reduce the number of calls to these methods, since a call is generated for every procedure call in the instrumented program and, in the case of `IProfTool`, for every basic block.

Several implementation improvements would significantly decrease the overhead of ProfBuilder. First, the current version of BIT only allows a single argument to be passed to the analysis routines. Future releases will eliminate this constraint. Second, as in our examples, often the analysis routine, such as trackInstructions, is a small procedure, and could be inlined with better compilation technology. Finally, as all the code in ProfBuilder is itself written in Java, it will benefit substantially from better Java compilers.

6.2 JLex optimization

In order to test how useful tools, such as `IProfTool`, built using ProfBuilder could be in helping to optimize programs, we attempted to optimize the performance of a program we had not seen prior to instrumenting it. For the program, we chose JLex, a lexical analyzer generator for Java written by Elliot Berk and similar to lex [2]. This program is an example of a moderately sized, publicly distributed Java application written by an outside party. We were able to instrument the program and collect data without any prior understanding of the source code or underlying algorithms.

The original application required an average of 6.86 seconds of execution time for a sample grammar that is included in the JLex distribution. The program was instrumented with `IProfTool`, and data was collected. A portion of the output is displayed in Figure 11. The output clearly indicates that a large number of bytecodes (13,472,313) were executed in one calling context, the context for sortStates. Since the total number of instructions executed was 24,526,004, this one context accounted for over half of the instructions executed in the program.

Examination of the JLex source code revealed that the `sortStates` method used a straight selection sort, an algorithm with an average running time of $O(n^2)$. In order to optimize this code, a merge sort algorithm (running time: $O(n \lg n)$) was substituted. The merge sort version had an average running time of 5.81 seconds for the same sample grammar. This represents a 15.4% reduction in execution time from the reimplementation of a single routine in the program. Clearly, the information provided by the `IProfTool` was valuable in the optimization of JLex.

```
Total metrics
Instructions: 24526011

=============================================================

Metrics local to procedures

-------------------------------------------------------------
Instructions
-------------------------------------------------------------
(       7)                     | TimeRun.main
(      14)                     |   JLex/Main.main
... (detail omitted)
(     798)                     |             JLex/JavaLexBitSet.set
(      48)                     |               JLex/JavaLexBitSet.resize
(      12)                     |                 JLex/JavaLexBitSet.nbits2size
( 3499882) *****               |             JLex/CNfa2Dfa.e_closure
( 1457610) **                  |               JLex/JavaLexBitSet.set
(13472313) ********************|               JLex/CNfa2Dfa.sortStates
(    2888)                     |             JLex/CNfa2Dfa.add_to_dstates
(     988)                     |               JLex/CAlloc.newCDfa
...(remaining output omitted)
```

Figure 11: A portion of the IProfTool output from JLex.

7 Related Work

The data structure and algorithms for the Calling Context Tree were described by Ammons, Ball and Larus [1]. The data structure was written as described in their paper, with a few modifications for the Java environment, specifically using arrays of integers instead of hardware counters to record metrics.

Digital Equipment Corporation developed ATOM [11], or Analysis Tools using OM, for instrumenting binaries on DEC machines. Lee and Zorn developed BIT [7, 8], a Java tool similar to ATOM. The Calling Context Tree tools use BIT to instrument the bytecode. Both ATOM and BIT allow the user to construct tools that monitor dynamic program execution by inserting calls to analysis routine directly into the compiled code.

Profiling tools originated with gprof [3], a dynamic execution profiler. gprof describes path profiles by describing the caller/callee connection between two routines. From these connections, the structure of the call graph is inferred. Descendents based on gprof include HiProf [13], a commercial hierarchical profiling tool for the x86 architecture that inspired our `IProfTool`, mprof [14], a memory allocation profiler that is the basis for the `MProfTool` described here, and cprof [6], a cache profiler. Each of these tools can be used to profile programs written in languages like C, but are not implemented for Java bytecode. Sun's JDK version 1.1.5 provides a profiling mechanism for Java programs by specifying the -prof option to the interpreter. However, this profile only provides information about Java library functions and not the application being run. Moreover, there are many profiling tools available, but few, if any, profiler generators. ProfBuilder is the only mechanism we are aware of that provides CCT profiling and the ability to rapidly construct new tools.

The CCT-based tools `IProfTool` and `MProfTool` differ from gprof and related tools in that they record the entire call path to each routine, and distinguish between different paths to the same routine. Thus, gprof can generate a dynamic call graph, but in a call graph, different paths merge at points where they share the same routine. Even if two paths share the same routine in a calling context tree, these paths are not merged, providing more detailed information about the dynamic behavior of the program.

8 Summary

We have implemented ProfBuilder, a software package for rapidly generating Java profiling tools. This package includes an implementation of a Calling Context Tree data structure in Java. Using the BIT library, we have constructed tools that instrument Java bytecode to construct a CCT and store metric data in it. Using CCT-based profiling tools, including the bytecode instruction profiler and memory allocation profiler, provides a more complete picture of the dynamic characteristics of a program. These characteristics include the efficiency of algorithms, excessive memory usage, and possible optimizations of the program, as

our experience with JLex demonstrates. ProfBuilder can be used to build a variety of tools to study branch prediction, caching algorithms, the differences in code produced by different Java compilers, and other details of dynamic program execution.

This implementation of the CCT and profiling tools not only demonstrates the effectiveness of using the CCT to profile programs, but the feasibility of using it to profile Java programs given the facilities of BIT. As the number of programs implemented in Java grows and the need for efficiency in execution continues to be relevant, the CCT profiling tools will provide a useful mechanism for understanding and optimizing these applications.

ProfBuilder is available for distribution by contacting the authors.

Acknowledgements

This research is supported in part by NSF Grants CCR-9711398 and IRI-9521046.

References

[1] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 85–96, New York, June 1997. ACM Press.

[2] Elliot Berk. JLex: A lexical analyzer generator for Java. Available at http://www.cs.princeton.edu/~appel/modern/java/JLex.

[3] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. An execution profiler for modular programs. *Software—Practice and Experience*, 13:671–685, 1983.

[4] Scott Hudson. Java based constructor of useful parsers (CUP). Available at http://www.cc.gatech.edu/gvu/people/Faculty/hudson/java_cup/home.html.

[5] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, La Jolla, CA, June 1995.

[6] Alvin R. Lebeck and David A. Wood. Cache profiling and the SPEC benchmarks: A case study. *Computer*, 27(10):15–26, October 1994.

[7] Han Bok Lee. BIT: Bytecode instrumenting tool. Master's thesis, University of Colorado, Boulder, Department of Computer Science, University of Colorado, Boulder, CO, June 1997.

[8] Han Bok Lee and Benjamin G. Zorn. BIT: A tool for instrumenting Java bytcodes. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems (USITS97)*, pages 73–82, Monterey, CA, December 1997. USENIX Association.

[9] Martin Odersky, Michael Philippsen, and Christian Kemper. EspressoGrinder. Available at http://wwwipd.ira.uka.de/~espresso.

[10] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The structure and performance of interpreters. In *Proceedings of the 7th International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS VII)*, pages 150–159, Cambridge, MA, October 1996.

[11] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[12] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimizations at link-time. *Journal of Programming Languages*, March 1993.

[13] Inc. TracePoint. Hierarchical profiling white paper. Available at http://www.tracepoint.com/frames.html, 1997.

[14] Benjamin Zorn and Paul Hilfinger. A memory allocation profiler for C and Lisp programs. In *Proceedings of the Summer 1988 USENIX Conference*, pages 223–237, San Francisco, CA, June 1988.