An Empirical Study of Java Dynamic Call Graph Extractors

Tao Xie David Notkin
Department of Computer Science & Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350 USA
+1 206 616 1844
{taoxie, notkin}@cs.washington.edu

ABSTRACT

A dynamic call graph is the invocation relation that represents a specific set of runtime executions of a program. Dynamic call graph extraction is a typical application of dynamic analysis to aid compiler optimization, performance analysis, program understanding, etc. In this paper, we empirically compare the results of nine Java dynamic call graph extractors quantitatively and qualitatively. We investigate those differences among the dynamic call graph extracted by different tools mainly caused by different underlying Java program instrumentation techniques and other design decisions. A comparison between static call graph and dynamic call graph shows software engineering tools for program understanding place a different requirement on dynamic call graph from compilers or profilers whose main purpose is optimization or performance tuning. Dynamic call graphs require some complementary static information and an effective representation to aid program understanding. Choosing an appropriate instrumentation technique, integrating static and dynamic information, and providing flexible user manipulation for dynamic call graphs can better facilitate program understanding task. In this paper, we discuss the study and sketch the design considerations for an effective dynamic call graph tool to support program understanding.

1. INTRODUCTION

A call graph is a binary relation over selected entities in a program, such as methods, classes, subsystem, modules, files, etc., which represents invocations between those entities. A static call graph is the relation describing those invocations that could be made from one entity to another in any possible execution of the program. Static call graphs are generally expected to be conservative: that is, they are not expected to omit any invocations that can take place in any execution. In practice, due to computational complexity, static call graphs are imprecise, including invocations that are never executed. A dynamic call graph is the relation including invocations over one or more actual executions of the program. Ideally a dynamic call graph is the subset of the static call graph for the same program. A dynamic call graph can be considered as one instance of the corresponding static call graph.

Dept. of Computer Science & Engineering
University of Washington
Technical Report 02-12-03
December 2002
UW CSE Technical Report 02-12-03

Compilers sometimes compute static call graphs to aid optimization. Many software engineering tools extract static call graphs to assist program understanding tasks, for example, subsystem classification, architecture recovery, architectural evolution tracking [15], software reflexion models [9], etc. Compared to static call graphs, dynamic call graphs tend to be simpler because they focus only on the invocation relation in certain executions of the program. In addition, a dynamic call graph reflects the connection between the dynamic behavior, which exhibits certain behavior of the program, and the program structure, which represents the implementation of that exhibited behavior. Dynamic call graphs can also be used to evaluate test coverage thoroughness and help debugging. Dynamic calls are popularly used in profiling tools to provide a framework for performance analysis or profile-driven compiler optimizations. They are especially useful in tuning the program parallelization and tracing the multiple-thread or distributed applications.

Compilers generally need to compute conservative static call graphs to ensure the correctness of optimizations over all possible executions. Software engineering tools for program understanding may impose a relaxed requirement on static call graphs [10]. For example, some false negatives (mistakenly omitted invocations) may be acceptable in static call graphs for some tasks. However, in some ways, software engineering tools may place a different, stricter in some sense, requirement on dynamic call graph than compilers or profilers. To better understand the connection between the program structure and dynamic behavior inferred by dynamic call graphs, the dynamic call graph should be supplemented with some static information. In addition, the requirement of the dynamic information scope and representation for program understanding task is also different from the one for compiler optimization or performance tuning.

Java, an object-oriented (OO) and multithreaded language, has been extensively adopted by the community of software developers [6]. It shares many language features common to most programming languages in use nowadays. To lay a better foundation for understanding dynamic call graphs, in this paper we empirically compare dynamic call graphs extracted from two Java micro-benchmarks by nine dynamic call graph extractors, comparing the static call graphs and dynamic call graphs extracted from the same program.

This paper makes four contributions:

- It presents, quantitatively and qualitatively, the differences of the extracted dynamic call graphs by nine different tools.

- Based on an analysis of the differences among extracted dynamic call graphs, it shows the capabilities and limitations of different underlying instrumentation techniques adopted by dynamic call graph extractors.
- Based on an analysis of the differences between static and dynamic call graphs, it shows that the integration of static and dynamic analysis can attack some problems faced by static call graph and dynamic call graph respectively in assisting program understanding.
- It discusses key design considerations for dynamic call graph extractors.

2. TOOL CATEGORIES

Dynamic analysis in Java instruments code by adding instructions to the Java source code, to the bytecode or to the Java Virtual Machine (JVM). While executing the Java program, these instrumented instructions export the required state information, such as executed methods, resources consumed on methods, etc. Instrumentation for dynamic analysis normally does not affect program semantics or change a program's functional behavior. However, it generally induces some performance overhead, perturbing the program to some degree. In this section, we take a broad look at the instrumentation techniques adopted by various Java dynamic analysis tools most of which can extract dynamic call graphs or call trees. In Table 1, the first three categories are to instrument the application program but the remaining three ones are to instrument the runtime environment where the application program is running.

Table 1. Instrumentation categories for tools

Instrumentation techniques	Examples
Source-level Instrumentation	Panorama for Java 1.0.5[13]
Static Bytecode Instrumentation	Bytecode Instrumenting Tool [3]
Dynamic Bytecode Instrumentation	Trace release 1 based on Byte Code Engineering Library [17]
Profiler Agent Using Java Virtual Machine Profiler Interface (JVMPI)	Intel VTune 5.0 [20] Rational Quantify 2001A [14] Optimizeit 4.02 [12] JProbe 3.0 [7] TrueTime 2.1 [18] IBM Jinsight 2.1 [16]
Statically Instrumented JVM	IBM Jinsight 2.0 [16]
Dynamically Instrumented JVM	ParaDyn-J [11]

Source-level instrumentation inserts instrumentation code at specified locations in a given piece of source code. Bytecode instrumentation inserts the instrumentation code by adapting the bytecode. Static bytecode instrumentation instruments the bytecode before it is loaded or executed in the runtime environment. Dynamic bytecode instrumentation instruments the bytecode after the class contained in the bytecode is loaded in the runtime environment. The Java Virtual Machine Profiler Interface (JVMPI) is a two-way method call interface between the Java virtual machine and an in-process profiler agent [19]. A profiling tool based on JVMPI can obtain a variety of information for a comprehensive performance analysis task. Statically instrumented JVM approaches instrument the JVM program in order to export some state information available while it executes the bytecode. The Dynamically Instrumented JVM approach generates and inserts instrumentation code into the JVM, or removes it from the JVM at runtime.

3. EMPIRICAL STUDY

In this empirical study, we gathered nine tools each of which extracts a dynamic call graph or method call information from a Java program and applied them to two micro-benchmarks. These tools represent an exhaustive list of the tools that were available to evaluate that met two criteria: the ability to produce or deduce a list of dynamic invocation relations, and the ability to run on the Intel Pentium platform running the Microsoft Windows 2000 operating systems. We found that among those tools that can extract dynamic call information, most of them are profiler tools intended for performance tuning. Furthermore, most commercial profiler tools are based on profiler agents that use JVMPI.

Bytecode Instrumenting Tool's ProfilerBuilder and ParaDyn-J are not maintained or available. Therefore there is no tool based on static bytecode instrumentation or dynamically instrumented JVM techniques in this empirical study. We studied these two techniques based on two related technical papers [3][11]. Among these nine tools, six are commercial, and we downloaded their trial versions to evaluate.

These nine tools we used in this empirical study were:

- Panorama for Java 1.0.5, a commercial tool family that performs both code static analysis and code dynamic analysis using source-level instrumentation [13].
- Trace release 1, an on-the-fly runtime method tracing tool using Byte Code Engineering Library 4.2.3 [17].
- Intel VTune Performance Analyzer 5.0, a commercial tool for performance tuning based on a profiling agent using the JVMPI [20].
- Rational Quantify 2001A, a commercial performance profiling tool based on a profiling agent using the JVMPI [14].
- Optimizeit Profiler 4.02, a commercial profiling tool based on a profiling agent using the JVMPI [12].
- JProbe Profiler 3.0, a commercial profiling tool based on a profiling agent using the JVMPI [7].
- DevPartner TrueTime profiler 2.1, a commercial performance profiler based on a profiling agent using the JVMPI [18].
- IBM Jinsight 2.0, a tool for visualizing and analyzing the execution of Java programs using instrumented Java VM based on IBM JDK 1.1.8 [16].
- IBM Jinsight 2.1, a tool for visualizing and analyzing the execution of Java programs using IBM Java 2 environments, and provides instrumentation via a profiling agent which uses the Java 2 JVMPI interface [16].

To provide the same inputs to these tools, we ran two micro-benchmarks using these tools on Sun's Java 2 runtime platform, except for IBM Jinsight 2.0 and 2.1, which are run on IBM Java 1.1.8 and 2.0 platforms.

3.1 Micro-benchmarks

In this empirical study, we applied the extractors to an OO micro-benchmark and a multi-threaded micro-benchmark. The OO micro-benchmark comprises some typical object-oriented features, like inheritance, virtual methods, etc. The Fibonacci micro-benchmark was chosen to investigate the result for target programs with multi-thread and parallelism [5]. The source code appears in appendix. To compare dynamic call graphs extracted by different tools, we lexically extracted the static call graph from the source code without using any further program analysis techniques (such as virtual method call resolution, etc). Therefore the extracted static call graph is mapped very closely to the source code.

We defined a set of notations for the representation of the static call graph. Figure 1 and 2 show the static call graph for the OO micro-benchmark and Fibonacci micro-benchmark. In the static call graph, a round corner rectangle represents the method whose name is marked inside the rectangle. The arrow represents the method invocation whose starting method is caller and ending method is callee. The vertical arrow beside the method rectangles with a number near the arrow represents the repeated method calls with times of that number, for instance, the notation for a method call A.func() inside a loop statement in Figure 1. The arrow pointing to the dotted rectangle enclosing the methods represents conditional method calls, which corresponds to the method calls inside "if… else…". More notations would be needed to express conditional method calls inside those multiple branching statements. However, in this paper we only introduce those notations needed to express the micro-benchmarks in this study.

3.2 Quantitative Results

To facilitate the comparison, we considered the common dynamic calls extracted by more than half of those nine tools as the "true" dynamic call graph, which was used as comparison baseline. Generally this baseline dynamic call graph consists of all user method calls and system method calls, which are sufficient to aid the optimization, performance tuning or program understanding.

The program parameters for these two micro-benchmarks are 1 and 2 respectively during execution. Figures 3 and 4 show the baseline dynamic call graph for the OO micro-benchmark and the Fibonacci micro-benchmark respectively. The notations for the dynamic call graphs are similar to those for static call graphs. The conditional method calls in static call graph have been resolved to actual method calls during execution. In the baseline dynamic call graph, the dotted round cornered rectangle is used to highlight a method callee that is not present in the static call graph but that appears in the baseline dynamic call graph. The line with a dot on one end connects the static callee name with the corresponding dynamic callee name, if they are different.

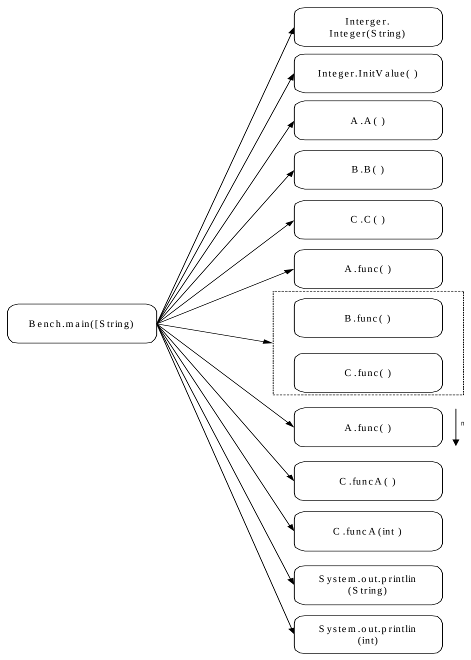

Figure 1. OO micro-benchmark static call graph

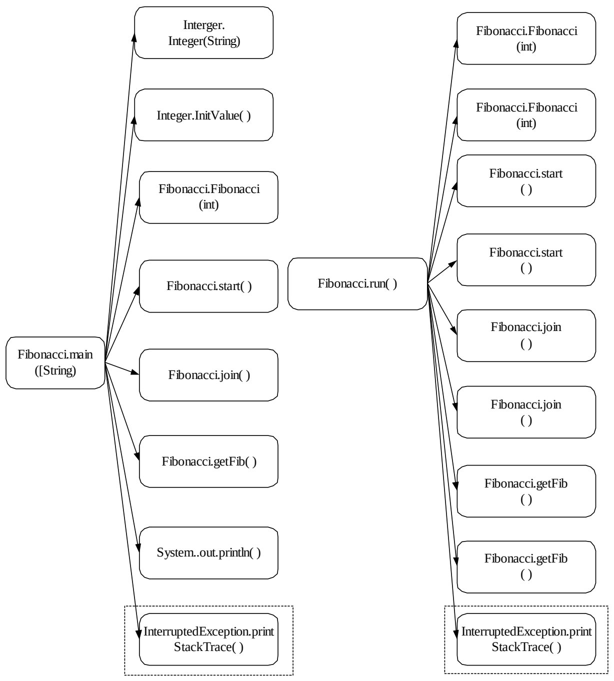

Figure 2. Fibonacci micro-benchmark static call graph

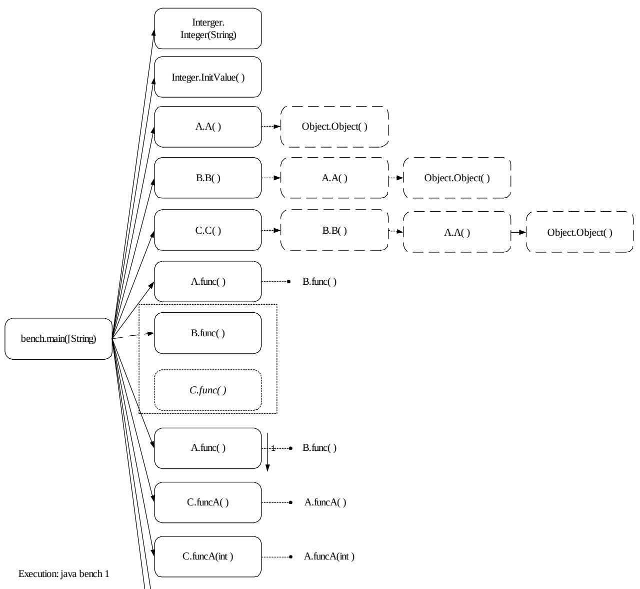

Figure 3. OO benchmark baseline dynamic call Graph

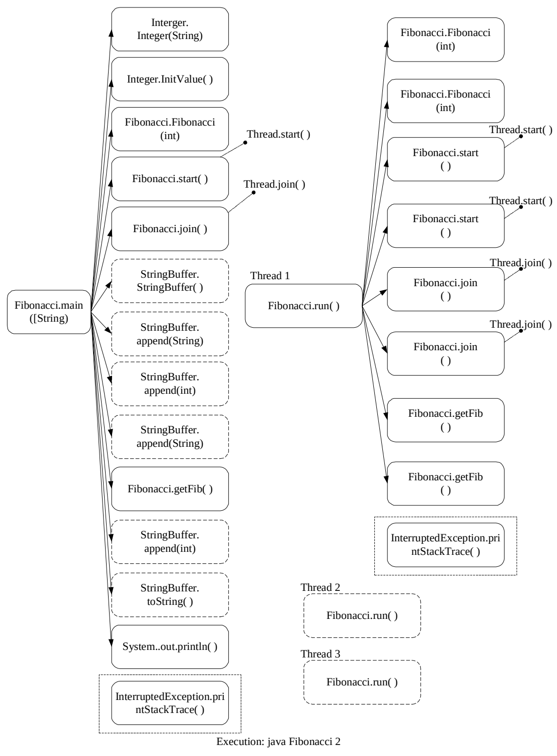

Figure 4. Fibonacci benchmark baseline dynamic call graph

In the baseline dynamic call graph, OO micro-benchmark's main method has 12 direct method calls, including eight user method calls and four system method calls. Fibonacci micro-benchmark's main method has 13 direct method calls, including two user method calls and 11 system method calls.

The comparison between static call graph and baseline dynamic call graph shows following results:

- Five method calls in the baseline dynamic call graph extracted from OO micro-benchmark do not appear in its static call graph. Neither do eight method calls extracted from the Fibonacci micro-benchmark.
- The names of classes that four callees belong to in the static call graph are different from those corresponding ones in the dynamic call graph for the OO micro-benchmark. So are six callees' class names in the call graph for the Fibonacci micro-benchmark.

The quantitative comparison between baseline dynamic call graph and extracted dynamic call graph by nine tools is showed in Figure 5 and Figure 6.

These figures point out that no two tools extracted the same set of dynamic method calls for either micro-benchmark. To see the detailed differences of those extracted method calls that are not present in baseline dynamic call graph, Figures 7 and 8 show the method calls by the main method that are not in the baseline dynamic call graph. Each bar in these two figures reports the number of the method calls whose caller is main method and whose callee is specified in the leftmost of the row, extracted by the tool whose name is specified in the bottom of the column. These two figures show the result that all tools produce different set of method calls that are not present in baseline dynamic call graph except for Panorama, Trace and Jinsight 2.1. These three tools do not extract method calls that are not present in baseline dynamic call graph.

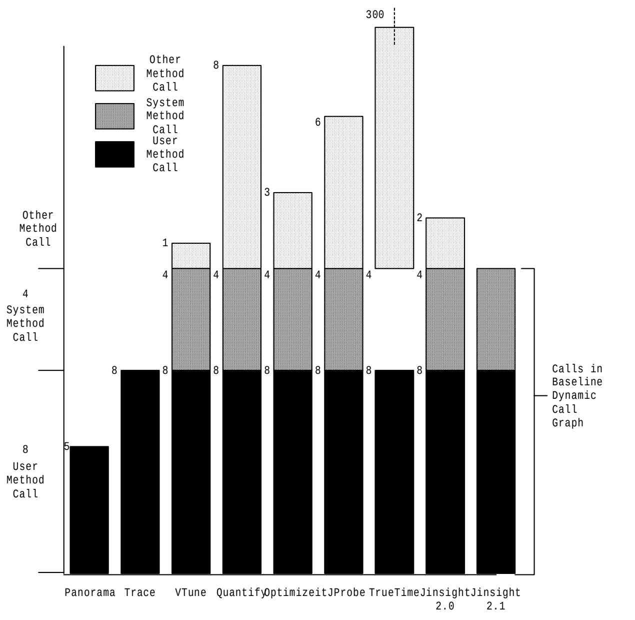

Figure 5. Quantitative comparison between baseline dynamic call graph and extracted dynamic call graphs by nine tools on OO micro-benchmark

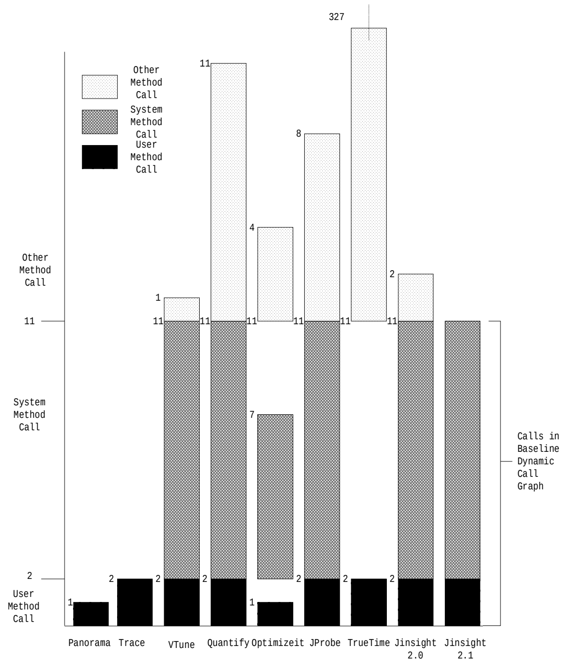

Figure 6. Quantitative comparison between baseline dynamic call graph and extracted dynamic call graphs by nine tools on Fibonacci micro-benchmark

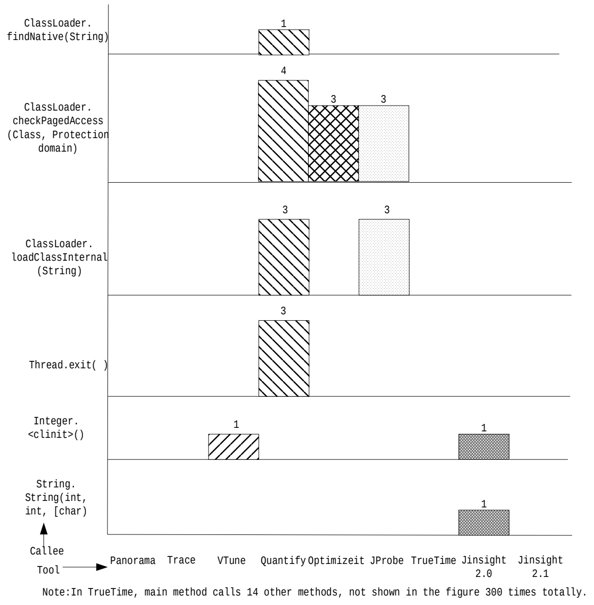

Figure 7. Quantitative comparison of callees produced by nine tools within main method that are not in the baseline dynamic call graph on OO micro-benchmark

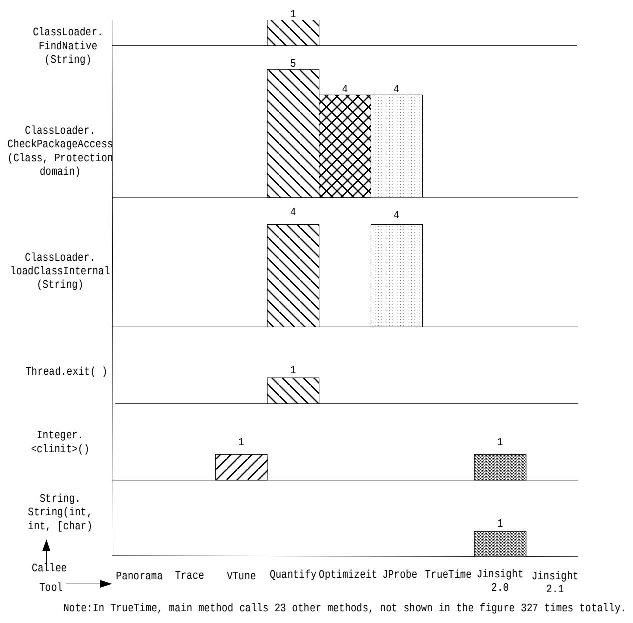

Figure 8. Quantitative comparison of callees produced by nine tools within main method that are not in the baseline dynamic call graph on Fibonacci micro-benchmark

3.3 Qualitative Results

3.3.1 Comparing Static and Dynamic Call Graph

Comparing the static call graph to baseline dynamic call graph qualitatively can provide some insights into the relationship between static and dynamic call graph.

For the static call graph, we call the method call extracted lexically from the source code the *formal method call* and the method call extracted dynamically from the program execution as the *actual method call*. There is an "*actual method call prediction*" problem for the static call graph, because in some cases the call graph extractor cannot decide exactly which callee will be called during runtime because of conditional branches or other language features; polymorphism is the most common cause of this problem in OO languages such as Java. Similarly, for dynamic call graphs there is a "*formal method call backtracking*" problem, where the dynamic call graph extractor cannot precisely recognize the caller in source-level terms; this is often due to instrumentation below the source-level. Together, these two problems complicate program analysis and understanding.

Some differences are shown between the static formal method calls and dynamic actual method calls in our study:

- The actual method callee can be different from the formal method call because of inheritance. For example, in the OO micro-benchmark call graph, the formal method callee C.funcA() becomes the actual method callee A.funcA() at runtime, because class C's funcA is inherited from class A. In the Fibonacci micro-benchmark, the formal method callee Fibonacci.Start() becomes the actual method callee Thread.Start() for a similar reason.

- The actual method callee cannot have a corresponding formal method callee because of inheritance. For example, in the OO micro-benchmark, the actual method callee A.A() by B.B() or C.C() does not have a formal method callee in the source code. In addition, implicit inheritance also produces some system actual method callees, like the actual method callee Object.Object() by A.A().

- The actual method callee can be different from the formal method callee because of virtual methods. For example, in the OO micro-benchmark, formal method callee A.func() becomes actual method callee B.func().

- Implicit system actual method callees can not have a corresponding formal method callee in the source code. For example, in the Fibonacci micro-benchmark, the StringBuffer.StringBuffer() does not have a explicit formal method callee in the source code.

- All tools only show the method name in the dynamic call graph with the format of CLASS.METHOD(), without providing the complementary format of OBJECT.METHOD (). However, in program understanding, sometimes the object name for the method call is also helpful. Sometimes distinguishing different object method calls is meaningful, rather than representing them as the same class method calls.

3.3.2 Comparing Extracted Method Calls

Comparing the call graph extracted by each tool to the baseline call graph quantitatively provides some insight into the nature of the extracted dynamic call graph, but it does not tell us much about the detailed similarities and differences and possible reasons of these differences. To provide insight into this information, we also performed the qualitative analysis on the results.

First we designed a framework to categorize the method calls extracted by each tool. Java program execution involves following five categories of method calls:

- *Explicit user method call* is a call whose callee is a user method and its implementation can be located in the user program. This type of method calls in the OO micro-benchmark include B.func(), A.funcA() and A.funcA(int). This type of method call is generally required in program understanding and other tasks.

- *Implicit user method call* is a call whose callee is a user method and its implementation cannot be located in the user program, for instance, the default constructor method in the user program. This type of method calls in the OO micro-benchmark includes A.A(), B.B() and C.C(). This type of method call is generally required in program understanding and other tasks.

- *System method call by user method explicitly or implicitly* is the call whose caller is user method and callee is system method. Explicit system method calls in the OO micro-benchmark include Integer.Integer(String), Integer.InitValue(), System.out.printlin(String) and System.out.printlin(int). Implicit ones include Integer.<clinit>(), String.String(int, int, [char]). When we want to analyze the performance impact of the system method calls, we may include this type of method calls in the dynamic call graph. In program understanding, sometimes some system method calls are required to understand the program behavior, like Thread.start() and Thread.join(), whose class Thread is inherited by a user class in the Fibonacci micro-benchmark. But sometimes including some unnecessary system method calls can complicate the call graph and make users lose focus on other important method calls; for example, including too many System.out.println() system method calls may make program understanding more difficult.

- *System method call by system method* is a call whose caller and callee are both system methods. We do not show this type of method calls in our study. Occasionally, this type of method calls is considered in performance analysis. Sometimes this type of method calls is useful in program understanding when users want to investigate the behavior of some system method calls; for example, some users may be interested in the method calls among classes in the underlying system framework.

- *Environment method call* is a call made in the runtime environment in order to execute the Java program. This type of method calls in the OO micro-benchmark include ClassLoader.findNative(String), ClassLoader.checkPagedAccess(Class, Protection domain), ClassLoader.loadClassInternal(String), etc. This type of method call is useful only when the users want to investigate the runtime environment behavior or measure the program's actual performance, including the runtime environment overhead.

Although omitting some method calls would compromise the accuracy and completeness of the result, it would be misleading to believe that extracting more calls is inherently better. The specific task of the software engineer will drive the demands on the call graph extractors in this (and other) dimensions.

To facilitate the qualitative analysis of the result, we consider the call graphs in terms of *false negatives* and *false positives*. False negatives are calls that are present in extracted dynamic call graph, but which are omitted from the baseline dynamic call graph. False positives are calls that are not present in extracted dynamic call graph, but which are included in the baseline dynamic call graph.

Our study showed that baseline dynamic call graph includes all explicit and implicit user method calls, most system method calls by user methods, but no environment method calls. System method calls by system methods are not included since we only investigated the direct callees of the user methods.

Some results related with false negatives and false positives in the study are:

- Only Jinsight 2.1 can extract the same dynamic call graph as the baseline one, yielding no false negatives nor false positives.

- Only the dynamic call graphs extracted by Panorama, Trace and Truetime produce false positives for both micro-benchmarks. Only Panorama produces user method call false positives, most of which are implicit user method calls, such as A.A(), B.B(), C.C() in OO micro-benchmark. All of Panorama, Trace and Truetime produce system method call false positives.

- Only the Panorama, Trace and Jinsight 2.1 do not produce false negatives. The other tools produce false negatives, most of which are environment method calls.
- Although it is reasonable to consider environment method calls in some cases of performance analysis, no two tools extract the same set of environment method calls; that is, no two tools produce the same set of environment method call false negatives.
- During performance analysis, some false negatives are shown to consume a nontrivial portion of the time spent by the caller. For example, Thread.exit() consumes 26% of the time spent on the main method of the OO micro-benchmark produced by Quantify but all other tools do not produce this false negative.

There are two results that are abnormal, representing potential bugs in the tools:

- In the Fibonacci micro-benchmark, Panorama does not even produce Fibonacci.Fibonacci(), which is an explicit user method call.
- OptimizeIt produces false negatives for the Fibonacci micro-benchmark. It does not report four method calls related with implicit String operations: StringBuffer.StringBuffer( ), StringBuffer.toString( ), and two calls of StringBuffer.append(int). Moreover, it does not report a user method call, Fibonacci.getFib( ), which is called to generate the parameter value of another method.

In addition, all tools except for Panorama support thread analysis, grouping the method calls of Fibonacci micro-benchmark by thread.

3.3.3 Comparing Method Call Representations

In this paper, we have used the term *call graph* to represent the invocation relation extracted from a program. Ammons etc. [1] described three representations for displaying the dynamic invocation relation: *dynamic call graphs*, *dynamic call trees* and *calling context trees*. In a dynamic call graph, each method is represented by only one vertex and each directed edge represents one or more method invocations. Its compactness is gained by sacrificing some context information. For example, a call chain whose length is beyond two is difficult to be extracted from the call graph when one method of that chain, which is neither the head nor the tail of that chain, is called by another method besides the call made by the caller in that chain. In a dynamic call tree, each vertex except for the root vertex, which is usually the main method, represents the callee for the corresponding method call. If a method is called *n* times, there are *n* vertices in the dynamic call tree. Its accuracy is gained at the cost of additional storage space especially when there are too many repeated method calls, for example, method calls inside a loop. There is a modified call tree that compacts the same callee with multiple invocations by the same caller as one vertex. It trades the ability to distinguish different invocations of the same callee for space storage and compactness. In the calling context tree, each vertex represents an individual context, which is a method together with the call chain that resulted in the invocation to that method. Repeated method invocations are represented by the same vertex if their call stacks are same. In addition, a back edge represents a recursive call. Each vertex encodes a unique call path. It is more compact than call tree, but still cannot distinguish different invocations of the same callee inside a method.

The different tools we studied use variants of several of these representations:

- Panorama represents the method calls as summary report text. Trace represents the method calls as trace line text.
- Vtune represents the method calls as a local call graph, which only displays the selected method, the method's parents (which are that method's callers), and descendants (which are that method's callees).
- Quantify and JProb represent the method calls as a dynamic call graph.
- OptimizeIt, Jinsight 2.0 and 2.1 represent the method calls as a call tree.
- TrueTime represents the selected method, the method's parents and the method's descendants in the form of table. Quantify and JProb also provide this table as a complementary method call representation.
- Vtune, Quantify, OptimizeIt and JProb provide some mechanisms to expand or collapse descendants, focus or hide the subtree. Jinsight 2.0 and 2.1 provides some mechanisms to expand a call tree to a specified depth or to focus the subtree.

Filtering is a good technique to remove false negatives or false positives when trying to customize the default extracted dynamic call graph to users' need. Filtering of the method calls can be performed before, during or after the execution of program. *Filtering before execution* imposes the filtering on the method call collection process. *Filtering during execution* can dynamically turn on or off the data collection on the fly when the program is running. This filtering is especially useful when users want to extract method calls during certain periods of the program execution. *Filtering after execution* imposes the filtering on the display process. The filtering can be *inclusion* or *exclusion filtering*. Some results of filtering are showed as below:

- Panorama and Vtune provide no filtering functionality.
- Trace, Quantify, OptimizeIt, and TrueTime provide filtering after the program execution, including exclusion and inclusion. Trace's filtering is at the granularity of classes. Quantify and OptimizeIt's filtering is at the class and method granularity. TrueTime's filtering is at the package and class granularity.
- Jinsight 2.0 and 2.1 provide filtering during program execution, including exclusion or inclusion of all traces in different time periods. It also provides exclusion filtering after execution before loading the trace information at a package and class granularity, or by limiting the call stack depth.

- JProb provides filtering after program execution, including exclusion and inclusion, at the class and method granularity.

Some other results of method call representation features are shown below:

- No tools present the time order of the callees inside certain methods, except for Trace tool. For example, in the OO micro-benchmark, the tools generally do not report which one of A.funcA() or A.funcA(int) is executed first.
- VTune, Quantify, JProb and TrueTime can highlight the associated source code of a selected user method if its source code is available.

4. DESIGN CONSIDERATIONS

4.1 Instrumentation

Source-level instrumentation provides high flexibility for users to specify *what program point to instrument* and *what to instrument at those points*. Besides a dynamic call graph, it is capable of supporting some other complex dynamic analysis tasks, such as Daikon's likely invariant detection [4]. It maps the dynamic information directly to the high-level source code. It generally needs a static analysis front-end to assist instrumentation. However, such instrumentation cannot be done without access to source code. Non-user callees including system library callees can not be extracted because system library methods can not be instrumented in the absence of their source code. Even implicit user method calls cannot be extracted, since the callees' implementation may not be located in source code.

Similar to source-level instrumentation, bytecode instrumentation also provides good flexibility for users. At the same time, it does not require that the source code be available. Bytecode contains more symbolic data than the executable image, keeping object-oriented information about the class, such as the names and type signatures of all the classes, methods, fields, and constant values. However, it loses access to some other useful information, such as local variable names, parameter names and precise source-level statement constructs, like loop constructs, which are available in the source code. Therefore the dynamic information cannot be represented in terms of the above lost entity names. If the complete set of classes can be determined statically and their bytecodes are available offline, a static bytecode instrumentor can instrument the transitive closure of all classes offline and produce a new set of instrumented bytecodes, including system method callees. It does not need to modify the JVM. But sometimes the complete set of classes cannot be determined statically and it can not extract the implicit user method calls. Dynamic bytecode instrumentation does not require a priori knowledge of the set of classes loaded and saves some of the space needed to store statically instrumented. This approach can solve the problems encountered by static bytecode instrumentation. But online instrumentation needs to modify the JVM and imposes extra runtime overhead during loading phase.

The Java Virtual Machine Profiler Interface (JVMPI) provides hooks into the JVM that can be used without modifying the user program or the JVM itself. A profiler agent instructs the virtual machine to send it the relevant JVMPI events, such as method enter and exit, and processes the event data into profiling information. Most call graph based Java profilers adopt this approach. However, the tools using this approach are limited by the events provided by JVMPI. It cannot perform some complex dynamic analyses that can be performed by using source-level instrumentation or bytecode instrumentation.

JVMPI supports two kinds of profiling: *statistical CPU sampling* and *code instrumentation* [19]. In statistical CPU sampling, the executed application has to be interrupted periodically to record which methods are currently being executed. The accuracy of the result is largely affected by the sampling frequency. Moreover it cannot record the number of method calls. For example, when using sampling mode with one millisecond frequency, which is the minimum value in Optimizeit, no method calls are extracted inside bench.main() of OO micro-benchmark and only Thread.join(), StringBuffer.append(int) and StringBuffer.append(string) are extracted inside Fibonacci.main( ) in the Fibonacci micro-benchmark. In code instrumentation, JVMPI allows the profiler agent to instrument every class file before it is loaded by the virtual machine, which is similar to dynamic bytecode instrumentation. This mode may cause larger overhead and distort the performance result, but it usually does not miss the method calls.

An instrumented JVM can provide more flexibility to users but the development effort of this approach is much higher than using JVMPI. In addition, the evolution of a supported JVM, or supporting more JVMs, can induce a high maintenance cost. Among those tools that adopt this approach, Javiz [8] intends to use JVMPI to remove its dependence on a modified JVM. But its tracing of client/server activity still needs to be done by modifying the RMI library implementation, since JVMPI does not provide this functionality. Jinsight 2.0's subsequent version 2.1 supplies a profiling agent using the JVMPI for Java 2 instead of using an instrumented JVM.

Dynamic instrumentation inserts and deletes instrumentation in Java method code and JVM code at any point during execution. It can dynamically change the instrumentation with high flexibility when necessary.

4.2 Instrumentation Design Considerations

Instrumentation involves two issues: what program point to instrument and what to instrument at those points. Tool designers must consider how these issues can be addressed by a candidate instrumentation technique before choosing it.

4.2.1 Instrumentation Place

Tool designers should make sure the selected instrumentation approach provides users enough flexibility to instrument the required program points. In source-level instrumentation, instrumentation points are limited in the program points between source-level statements. Bytecode instrumentation allows the instrumented program points to be between the bytecode instructions, which can be inside a source-level statement. In runtime environment instrumentation, tool designers should make sure whether the required dynamic information can be extracted at a given point of the JVM. Source-level instrumentation is not appropriate for extracting dynamic call graphs since the system method or implicit user method's entry and exit points are not available in the source code level. Neither is static bytecode instrumentation, since it cannot extract the implicit user method calls. Dynamic bytecode instrumentation and runtime environment instrumentation are good candidate techniques for dynamic call graph extraction.

4.2.2 *Instrumentation Content*

Generally the instrumentation instructions export the required dynamic information to analysis tools indirectly or directly. The dynamic information is generally reported in terms of source-level program entity names, which is generally understandable to users. In source-level instrumentation, there are no problems in mapping the dynamic information to source-level program entity names. In bytecode instrumentation and runtime environment instrumentation, there are no problems in mapping the dynamic information to some program entities, such as class, method and field, etc., but it may be difficult to map it to some other program entities, like the name of local variable or parameters, source-level data constructs or control constructs, if there is no debugging information produced by compiler available to help compute these mappings. For dynamic call graph extraction, generally all of the above instrumentation techniques fulfill the requirement of exporting dynamic information in terms of class and method name.

4.2.3 *Online vs. Offline Analysis*

When the raw dynamic information is extracted from the instrumented points, the tool can process it online to save the I/O overhead to output the raw data. The online analysis results can be used by later executed instrumented codes or even add or remove the later instrumentation points online by using dynamic instrumentation technique if this approach is adopted. Moreover, when only the real time data is required and historical raw data do not need to be stored, online analysis can also save the space to be used to store those raw or intermediate data. In offline analysis, the raw data are exported without being analyzed at runtime, saving the process time but introducing the I/O overhead. The tradeoff between process time and I/O exporting time need to be considered in practice.

4.2.4 *Performance Issues*

Any type of instrumentation will perturb the application program to some degree. If the task is performance-critical, such as performance tuning or parallelism tracing, performance issues need to be considered when choosing the instrumentation technique. Many factors may affect the amount of overhead so an empirical comparison study needs to be done to provide some decision support information.

4.3 Scalability Considerations

Scalability problems appears to be more serious for dynamic analysis tools than for static analysis tools in some sense. Even in our limited study, the size of the trace file generated by running small micro-benchmarks is still relatively large. Theoretically its size is O(METHOD NUMBER * EXECUTION TIMES), because during each execution, the methods may be called multiple times, even infinite times if the methods are inside an infinite loop. In static call graph extraction, or other static analyses, the time complexity is generally the major factor to make it unscalable (or highly imprecise). However, for dynamic call graph extraction or other dynamic analyses, the space complexity is generally more of a concern. There are some design decisions to be made to attack this problem.

The first one is trace information filtering. The filtering can be imposed in method, class, file, package or directory granularity. But the more fine-grained the instrumentation granularity is, the more burden imposed on users. Filtering before or during program execution can reduce the runtime instrumentation overhead and the size of trace file. If filtering is changed, the program will have to be run again, imposing the new filter on the next execution. However, filtering after execution can allow user to change the filtering requirement without rerunning the program. Because we get the complete trace information during one execution, we only need to specify and impose a new filtering before loading or representing the dynamic information.

The second approach is to represent the trace information by loading on demand and caching frequently used traces. Persistently caching frequently used representations that are small but relatively expensive to compute can be effective [2].

4.4 Usability Considerations

There are many issues related to usability. One of them is to provide users with a way to easily manipulate information in the call graph. The key principle is to provide both coarse-grained and fine-grained manipulation of the information. The coarse-grained one is for the novice users, without imposing much burden. The fine-grained one is for experienced users, who want to gain more control of the information and can tolerate relatively complicated manipulation of the information. During tool design, the designer might not make an arbitrary decision between these two approaches. Providing both granularities of manipulation to users will attract both types of users, leaving the choice to future users; this is not especially easy, however. In dynamic call graph extraction, the tool shall provide the basic information to users, for example, without time order or object method call, clustering all invocations of the same callee inside the caller to one node, etc. It is reasonable to do so in order to control the complexity of the dynamic call graph to be manageable. But if users want to, the tool shall let them choose to show the detailed information to assist the program understanding, like time order, object method call associated source code, etc. In addition, it is good to provide expanding or collapsing, subtree focusing or hiding, for users to control the complexity of the focused information.

Incremental manipulation is a good feature for users. The tool shall provide certain functionality to let user save their manipulation results or the browsing history on the information. Then they can load them later and continue to work on them. Jinsight's slicing and workspace techniques are good in this dimension [16]. In addition, a static call graph shall be shown together with the dynamic call graph to attack the "actual method call prediction" and "formal method call backtracking" problems. For example, it is good to provide users to view the formal method calls that have no corresponding actual method calls, etc.

5. DISCUSSION

A lot of research work has been done to use the static analysis techniques to address the "actual method call prediction" problem; to some extent, for example, pointer alias analysis, virtual method call resolution, etc. However, the "formal method call backtracking" problem has not been paid much direct attention by researchers, tool designers and practitioners. Static information is generally ignored in dynamic analysis tasks, which makes program understanding difficult. The integration of static analysis and dynamic analysis can attack these problems.

A dynamic call graph extractor can resolve the "actual method call prediction" problem that frustrates the static call graph extractor, such as method pointers, virtual methods, etc. A dynamic call graph extractor also narrows down the scope complexity of user concerns by only collecting the information related with certain execution.

When we map the dynamic call graphs on the static call graph, the invocation relations in dynamic call graphs should be a subset of the invocation relations in the static call graph. With the integration of the static and dynamic call graph, we can analyze code coverage at the method level. Program profiles only show the times the callee is called and the accumulative time spent on it. However, by integrating static and dynamic call graphs, we can show those potential callees in the static call graph that have not been called during the past executions but can possibly be called in future executions.

Generally dynamic analysis captures the dynamic behaviors of the run time entities, which are named in the low level binary code or machine language space. Sometimes this low level information is hard for users to understand. For example, an object created in runtime is identified as the object ID in running environment, losing the high level object name's label. In addition, because of some languages features, like inherited methods, virtual methods, etc, the dynamic call information only shows the original class this callee method belongs to, losing the formal name information of the class that this callee is attached to in the high level source code. The integration of static analysis and dynamic analysis can address these issues. For example, the static information produced by the compiler can be used to find out the entity mapping between the low-level code and high-level code, attacking the first problem. Other static information can also assist in attacking the second problem.

6. CONCLUSIONS

Our empirical study shows that the extracted Java dynamic call graphs by nine tools are quite different because of the underlying instrumentation techniques and other design decisions made by tool designers.

Our study also shows the differences between static call graphs and dynamic call graphs. To analyze these differences, we propose the concepts of static formal method calls and dynamic actual method calls. The static call graph has the "actual method call predication" problem caused by many factors, including the branching statements and dynamic binding OO features. The dynamic call graph has the "formal method call backtracking" problem mainly caused by the differences between high-level program space and low level byte code space. The integration of static analysis and dynamic analysis can attack these two problems.

By analyzing the differences between the dynamic call graphs extracted by nine tools quantitatively and qualitatively, we found that few tools under study can extract the satisfactory set of method calls and related information to aid program understanding. And these tools can be improved in different aspects to support program understanding. Finally we enumerated some design considerations for dynamic call graph extractors, including instrumentation, scalability and usability considerations. Although this paper only considers the Java dynamic call graph extraction, many of the observations on the design decisions still apply to other Java dynamic analysis tools.

7. ACKNOWLEDGMENTS

We wish to thank the authors and companies of all the call graph extractor tools used in this study for making their tools available to us.

8. REFERENCES

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In Proceeding of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97), volume 32, 5 of ACM SIGPLAN Notices, page 85-96, New York, June 1997. ACM Press.

[2] D. C. Atkinson and W. G. Griswold, The Design of Whole-Program Analysis Tools, Proceedings of the 18th International Conference on Software Engineering, March, 1996.

[3] B. F. Cooper, H. B. Lee, and B. G. Zorn. ProfBuilder: A Package for Rapidly Building Java Execution Profilers. Technical report, University of Colorado, April 1998. Bytecode Instrumenting Tool is available at http://www.cs.colorado.edu/~hanlee/BIT/

[4] M. D. Ernst, Dynamically Detecting Likely Program Invariants, PhD Dissertation, University of Washington, Department of Computer Science and Engineering, August 2000. Daikon is available at http://sdg.lcs.mit.edu/~mernst//daikon/

[5] Fibonacci benchmark is available at http://www.eb.uah.edu/~crash/javanauts/benchmarks/

[6] J. Gosling, B. Joy, and G. Steele, Java Language Specification. Addison-Wesley, August 1996

[7] JProbe 3.0, Sitraka Software Inc (2001), Ontario, Canada, available at http://www.sitraka.com/software/jprobe/

[8] I. H. Kazi, D. P. Jose, B. Ben-Hamida, C. J. Hescott, C. Kwok, J. Konstan, D. J. Lilja, and P. Yew, JaViz: A Client/Server Java Profiling Tool, IBM Systems Journal, Volume 39, Number 1, 2000, pp. 96-117. Javiz 0.3 is available at http://www.cs.umn.edu/Research/JaViz/

[9] G. C. Murphy, D. Notkin, and K. Sullivan, Software Reflexion Models: Bridging the Gap Between Source and High-Level Models, In the Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, October 1995, ACM, New York, NY, p. 18-28.

[10] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S.-C. Lan. An empirical study of static call graph extractors. ACM Transactions on Software Engineering and Methodology, 7(2):158--191, April 1998.

[11] T. Newhall and B. P. Miller. Performance measurement of dynamically compiled java executions. In Proceedings of the ACM 1999 conference on Java Grande, pages 42--50, New York, 1999. ACM Press.

[12] Optimizeit 4.02, VMGEAR Inc. (2001), available at http://www.intuisys.com/

[13] Panorama for Java 1.0.5, International Software Automation, Inc., San Mateo, CA (2001), available at http://www.softwareautomation.com/java/

[14] Rational Quantify for Windows v2001A, Rational Software Corporation (2001), available at http://www.rational.com/products/quantify_nt/

[15] T. Richner, Using Recovered Views to Track Architectural Evolution, ECOOP '99 Workshop on Object-Oriented Architectural Evolution, June 1999

[16] G. Sevitsky, W. De Pauw, R. Konuru. An Information Exploration Tool for Performance Analysis of Java Programs, TOOLS Europe 2001, Zurich, Switzerland, March 2001. Jinsight 2.0 and 2.1 are available at http://www.research.ibm.com/jinsight/

[17] Method tracing tool release 1 is available at http://www.geocities.com/mcphailmj/Trace/. Byte Code Engineering Library is available at http://bcel.sourceforge.net/

[18] NuMega DevPartner TrueTime Java Edition 2.1, Compuware Corp. (2001), Farmington Hills, MI, available at http://www.numega.com/products/java.shtml

[19] D. Viswanathan, S. Liang, Java Virtual Machine Profiler Interface, IBM System Journal, Vol 39, No 1, 2000, pp. 82-95.

[20] Intel VTune Performance Analyzer 5.0, Intel Co.(2001), available at http://developer.intel.com/software/products/vtune/

Appendix A: OO Micro-benchmark:

```
class A {
  int  count = 0;
  public void func()  {
     count = count +1;
  }

  public void funcA() {
    count = count + 2;
   }

  public void funcA(int i)  {
   count = count + i;
  }
}

class B extends A {
  public void func()  {
     count = count + 10;
   }
}

class C extends A {
  public void func()  {
    count = count + 100;
 }
}
```

```
public class bench
{
  public static void main (String arg[])
  {
          A          a;
          B          b;
          C          c;

          int n = new Integer(arg[0]).intValue();
          a = new A();
          b = new B();
          c = new C();
          if (n != 0)
                a = b;
          a.func();
          if (n != 0)
                b.func();
          else
                c.func();
          for (int i =0; i < n; i++)
                a.func();
          c.funcA();
          c.funcA(n);
          System.out.println("a.count =");
          System.out.println(a.count);
  }
}
```

Appendix B: Fibonacci Micro-benchmark:

```
public class Fibonacci extends Thread
{
   int fib;
   Fibonacci(int n) {
        fib = n;
   }
    /*  Called by start() */
   public void run()    {
        if (fib == 0 || fib == 1)
              fib = 1;
        else {
           Fibonacci thread1 = new Fibonacci(fib-1);
           Fibonacci thread2 = new Fibonacci(fib-2);
           thread1.start();
           thread2.start();
           try  {
              thread1.join();
              thread2.join();
              fib = thread1.getFib() + thread2.getFib();
           }
           catch( InterruptedException e) {
              e.printStackTrace();
            }
        }
   }

  public final int getFib()  {
        return fib;
  }
```

```
public static void main(String arg[])  {
        Fibonacci fib;
        int n = new Integer(arg[0]).intValue();

        fib = new Fibonacci(n);
        fib.start();
        try  {
            fib.join();
            System.out.println("The Fibonacci for " +
                         n + " is: "+ fib.getFib());
        }
        catch( InterruptedException e) {
               e.printStackTrace();
        }
   }
}
```