Computer Science
Technical Report

Compiling Java to SUIF: Incorporating Support for Object-Oriented Languages

Sumith Mathew, Eric Dahlman and Sandeep Gupta
Computer Science Department
Colorado State University, Fort Collins, CO 80523
{mathew, dahlman, gupta}@cs.colostate.edu

July 1997

Technical Report CS-97-114

Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873

Phone: (970) 491-5792 Fax: (970) 491-2466
WWW: http://www.cs.colostate.edu

Compiling Java to SUIF: Incorporating Support for Object-Oriented Languages *

Sumith Mathew, Eric Dahlman and Sandeep Gupta
Computer Science Department
Colorado State University, Fort Collins, CO 80523
{mathew, dahlman, gupta}@cs.colostate.edu

Abstract

A primary objective in the SUIF compiler design has been to develop an infrastructure for research in a variety of compiler topics including optimizations on object-oriented languages. However, the task of optimizing object-oriented languages requires that high-level object constructs be visible in SUIF. Java is a statically-typed, object-oriented and interpreted language that has the same requirements of high-performance as procedural languages. We have developed a Java front-end to SUIF, `j2s`, for the purpose of carrying out optimizations. This paper discusses the compilation issues in translating Java to SUIF and draws upon those experiences in proposing the solutions to incorporating object-oriented support in SUIF as well as the extensions that need to be made to support exceptions and threads.

1 Introduction

Object-oriented languages with their features of inheritance and abstraction equip programmers with the tools to write reusable programs quickly and easily. However, the use of these features results in code that is quite different in structure from procedural code. Object-oriented programs tend to have smaller method sizes, data-dependent control flow due to dynamic dispatch and a large number of potential aliases [ZCG94]. Heavy use of the feature of dynamic dispatch and the fact that methods tend to be small in size impose performance penalties when the programs are compiled using traditional intraprocedural techniques [ASU86].

Optimization techniques [Dea96, DDG+96, Ple96, Cha92] have been proposed which have a direct benefit of eliminating the overhead of dynamic dispatch by statically determining the receiver of a method call. Moreover greater indirect benefits are obtained by enabling inlining and hence further intraprocedural optimizations. However, in order to evaluate the relative costs and benefits of these optimization techniques, it is necessary to have a common framework on which these optimizations are performed. Also, the framework needs to be language-independent so that programs of different object-oriented languages can get comparable treatment. An intermediate representation can act as the interface between different language front-ends and the optimizing back-end.

SUIF [Gro94] is a compiler infrastructure that allows easy experimentation and incremental optimization through multiple passes. The feature of extensibility [Smi96] makes it possible for SUIF to support many user-defined constructs. Also, SUIF has built in support for interprocedural optimizations [Pan96] and parallelization. We propose to implement an optimizer for object-oriented languages on the SUIF representation of the program.

*This project is supported by Faculty Grant #168597 from Colorado State University

A necessary requirement of the intermediate representation is that it must allow the different front-ends to describe the structure of the source program in such a way that optimizations may be performed. Object-oriented languages use constructs like method calls, field accesses, object instantiations etc., which are not found in procedural languages. To perform static binding of dynamically-dispatched calls, it is necessary that these constructs be easily identifiable in the intermediate representation. However, the current version of SUIF (1.x) does not have any front-ends for object-oriented languages and does not support object-oriented constructs. Therefore, a major endeavor of this project has been to identify the extensions, if any, that need to be made to SUIF to support these features.

This paper deals with the issues in compiling Java, a statically-typed and object-oriented language, to SUIF. As Java is an interpreted and dynamic language we have had to make some changes to the compilation model. Java programs are compiled to bytecode before being translated to the SUIF representation. We draw upon our experiences in building the translator, `j2s`, to show how SUIF can be used to support object-oriented features. We also propose solutions for supporting exceptions and threads in SUIF.

The rest of this paper is organized as follows. Section 2 deals with the issues in translating bytecode to SUIF in detail. It describes the instruction format, data-types and object-oriented constructs that are supported by the Java Virtual Machine.

In translating Java to SUIF we have had to deal with the issue of supporting object-oriented features in SUIF. Section 3 outlines the features that can be supported in the existing version of SUIF as also the extensions that need to be made. The issues of exception handling and thread libraries are also discussed because they are present in most object-oriented languages.

The conclusions and future work are described in section 4.

2 Translating Bytecode to SUIF

Translating bytecode to SUIF involves handling the different data types and bytecode instructions without any alteration in the semantics of the program. The bytecode is to be finally compiled to native code. Currently, the compilation model requires that all the bytecode files be available at compile time as it does not support dynamic linking.

The current version of the translator compiles each bytecode (`.class`) file into a SUIF file. Therefore, programs with multiple (`.class`) files will have a corresponding number of SUIF files. Each of the SUIF files have one `file_set_entry` per `fileset`. In a later pass the multiple `file_set_entries` are linked and placed in a single `fileset` for the purposes of interprocedural analysis.

2.1 Compilation Model

The `Java` to SUIF translator, `j2s`, is one step in the compilation framework for `Java` programs which is outlined in Figure 1 .

`Java` source code is first compiled to bytecode before it is translated into SUIF. The reasons for choosing bytecode as the source language are many. As bytecodes do not have to be parsed, unlike source code, a simple reader is sufficient to read in the information from a bytecode file. Further, bytecodes provide a sufficiently high level of representation of the constructs used in the language. This is important for the purposes of optimization where all the object-oriented constructs from the source program need to be retained.

We are currently considering three alternatives for the execution strategy. The approaches can be classified as those interfacing with the Java Virtual Machine Interface and those that take a stand alone approach. In either case all the component bytecode files are required to be available at compile time. Dynamic loading is not currently supported. There are two ways of interfacing compiled code with Sun's Java Virtual Machine - using the Native Method Interface or the JIT (Just-In-Time)

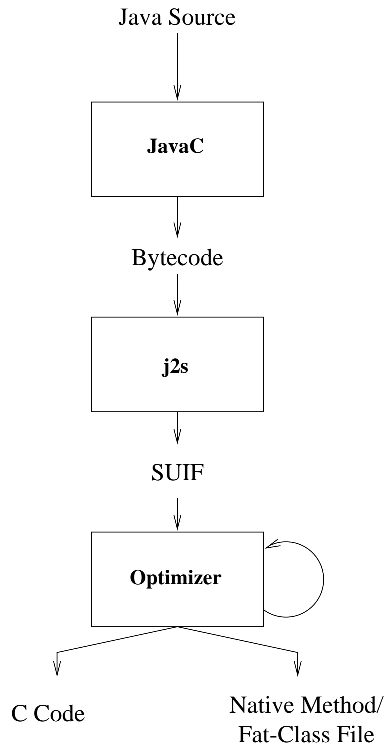

Figure 1: **Compilation Framework.**

API [Yel96]. The stand alone approach would require a run-time environment that duplicates many of the services that are provided by the Virtual Machine.

2.2 Converting Stack-Based Instructions to Three Address Code

A Java Virtual Machine instruction [LY97] consists of a one-byte opcode specifying the operation followed by zero or more operands supplying the arguments. The bytecode instructions are stack-based instructions. The translation procedure converts it to three-address code (SUIF expression tree). The inner loop of the translation procedure is given in Figure 2.

```
do {
    read the opcode
    read the arguments from stack and bytecode array
    create appropriate SUIF objects for arguments
    create appropriate SUIF instruction object
    if (no more instructions in expression tree)
      emit instruction
    else
      place address of instruction on SUIF stack
} while   (more instructions available)
```

Figure 2: **Converting Stack-Based Instructions to Tree Expressions**

Each expression tree is labeled with the number of the instruction in the Java bytecode using annotations to the `mrk` instructions. These are used in determining the targets of jump instructions.

The bytecode instructions for jumps represent the target as an offset within the given method. (We do not currently support jumps for exception handling using the `catch` and `finally` clauses). Patching the right SUIF instruction to jump to requires two passes. The first pass annotates the branch instructions with the number of the Java bytecode instruction. The second pass will patch in the destination operand of the SUIF jump instructions.

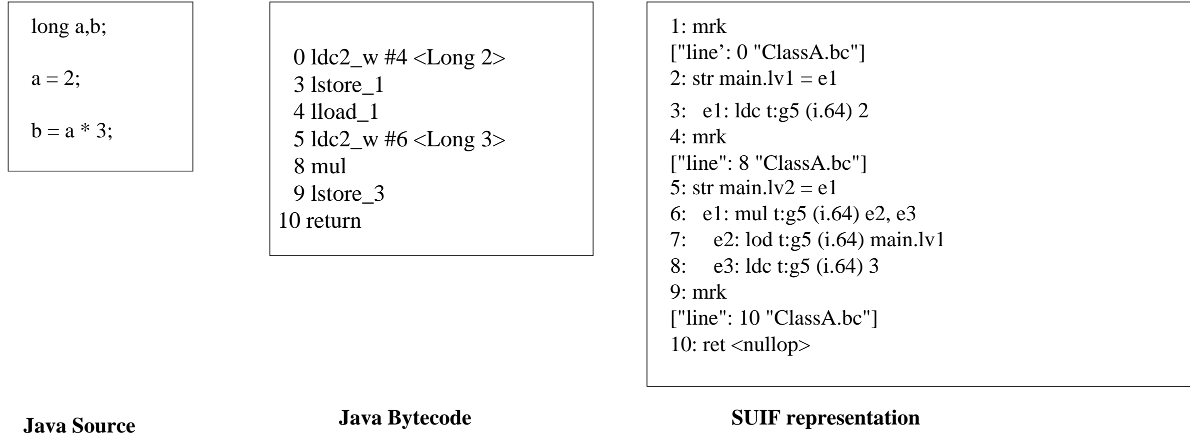

Figure 3: **Example of a Simple Expression Tree**

Figure 3 lists an example output from the translation procedure that converts stack-based code to an expression tree. The Java expression $b = a * 3$ is translated to bytecode. The `store` instructions in the bytecode generally denote the end of an expression tree. Instructions 4-9 in the bytecode are translated to instructions 5-8 in SUIF.

2.3 Annotations for Object-Oriented Features

Annotations are used to provide flexibility and extensibility to the intermediate representation [Gro94]. We use annotations to record information that is specific to object-oriented languages extensively.

Figure 4 shows the structure of the SUIF file and the annotations to the symbol tables as well as the `tree instructions`. The various elements of the Java program that are represented or enhanced by annotations are:

- **Class Hierarchy:** The superclass of each class is represented by an annotation to the `file symbol table`. When the `file_set_entries` are linked and merged under a single `fileset` , the program's class hierarchy is constructed from the superclass information and stored as an annotation in the `global symbol table`.
- **Meta-data:** The per-class data (section 3.5) is initially stored as annotations to the `file symbol table`. In a later SUIF pass the annotations are compiled to static data structures that are accessible to the run-time system.
- **Object Manipulation Instructions:** Method and field access instructions are implemented as procedure calls and are annotated as such. In object-oriented languages like Java this may not be necessary but other hybrid languages like C++ will require that regular procedure calls be different from method calls. Object creation instructions are implemented as procedure calls and need to be similarly annotated.
- **Line Numbers:** Each SUIF expression is annotated with the line number of the bytecode instruction. This is to help identify the target of branch instructions.

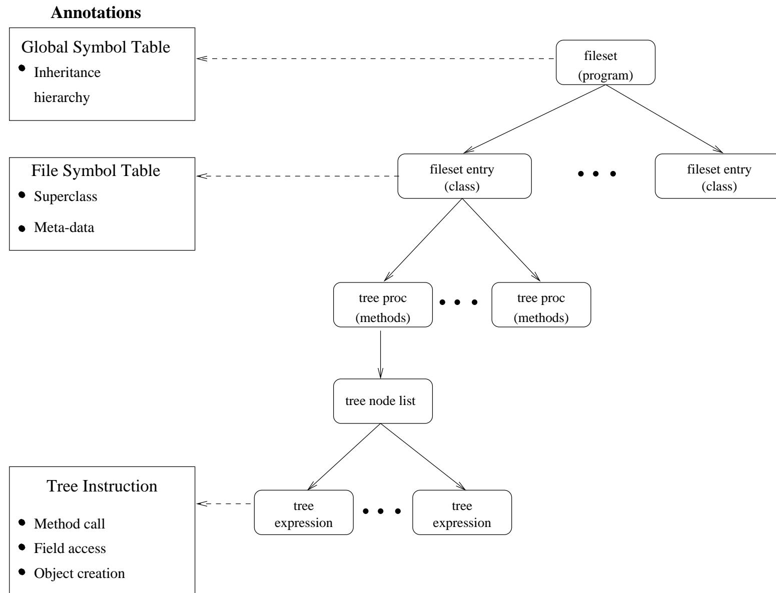

Figure 4: **Annotations for object-oriented features**

2.4 Handling Data Types

The `Java` language operates on two kinds of types: *primitive types* and *reference types*. All type checking is done at compile time and not by the Java Virtual Machine therefore data need not be tagged. Instead, the type of the data can be identified from the instruction that acts on it. The Java Virtual Machine has instructions intended to operate on values of specific types. All data values in the `constant pool` are in *big-endian* format.

When translating to SUIF it is necessary to find a corresponding SUIF data type. The remainder of this section discusses these issues.

2.4.1 Primitive Types

The primitive data types in `Java` are the numeric types, which include the `char` type, and the floating point type. These types are mapped to C data types in a straight forward manner.

2.4.2 Reference Types

All reference types are addresses (32-bit values). `Java` has three different kinds of reference data types:

- **Objects:** Object manipulation instructions are translated into operations that are supported by SUIF. Object references are pointers to a handle. However, the object layout is an implementation issue. This translator uses the C++ object model [WM95]. All field and method accesses to objects are handled via a process of `constant pool` resolution which is detailed later in this section.
- **Arrays:** Unlike in C, `Java` arrays are objects. All array operations are performed through method dispatch. Though direct referencing of array elements is not allowed, `Java` does allow fake array referencing instructions in the manner of `C++`. These source instructions translate to the `aload` and `astore` series of bytecode instructions. They are further translated to generic method calls in SUIF.

- **Strings:** Strings in Java are instances of class String and are not arrays of characters as in C++. Strings in Java are non-mutable (have a constant value) and string variables that are created with the same contents point to the same object. The process of loading a string literal from the constant pool requires the creation of a new String object.

2.5 Handling Object-Oriented Constructs

In the interpreted version, Java classes and interfaces are dynamically loaded, linked and initialized [LY97]. During the process of loading the class, the meta-data or Class object (see section 3.6) for the class is created from the binary file. The loaded binary class files are then linked to the Virtual Machine. Initialization of a class consists of executing the static initializers and initializers for static fields.

References to object-oriented constructs in the constant pool are initially symbolic. At run-time the symbolic reference is used to determine the actual reference of the entity. This process is known as constant pool resolution and it may involve loading, linking and initializing several types.

In a statically compiled environment all the classes in the program are loaded and linked prior to runtime. Therefore, the load and link components of constant pool resolution are absent. However, for the purposes of optimization the references to the constant pool are resolved and the names of references are made available in the SUIF code.

The instructions below are initially implemented as generic function calls. If the JIT API is used for execution the function will call the appropriate function in the Java Virtual Machine. In other cases, the implementation of this function is to be worked out.

The instructions for manipulating objects include:

- **Object creation :** The instructions for object creation include new, newarray, anewarray and multianewarray. They are implemented as function calls to the function new_java_object which takes the meta-class object (section 3.6) as the argument.
- **Method invocation :** The instructions invokeinterface, invokespecial, invokestatic and invokevirtual are used to invoke methods. As the index of the method may not be known statically the method invocation is maintained as a call to the generic function call_java_method_x which takes a pointer to the meta-class (section 3.6), method name and method parameters as arguments (void pointers). The x is a number between 1 and 64 and it stands for the number of arguments. This eliminates the overhead involved in using variable number of arguments. The exact type of the arguments is constructed from the method signature. There is a separate class of generic function calls for static methods.
- **Field Access :** The field access instructions include getfield and putfield. They are implemented as calls to the function put_java_field_x or get_java_field_x. The x stands for the type of the field. The arguments to the function are the meta-class of the object (section 3.6), the name of the field, the object pointer and the value (in case of pufield).
- **Other Object Manipulation:** The instructions include instanceof and checkcast. These are implemented by function calls that lookup the information in the meta-class object.

3 SUIF Support for Object-Oriented Features

Current versions of SUIF do not support object-oriented constructs. The intermediate representation is a mixed-level representation [Gro94] for procedural languages. The only high-level constructs supported are loops, conditional statements and array accesses. This section deals with the various object-oriented constructs and other features that are present in Java that need to be supported for the purposes of optimization. Many of the constructs can be emulated by using the existing SUIF instructions while others may require extensions to SUIF.

3.1 Object Layout

The Java Virtual Machine Specification maintains a separation between the public design and private implementation of the Virtual Machine. A virtual machine implementation should be able to read `.class` files and to exactly implement the semantics of the code therein. The implementor is free to optimize the implementation within the constraints of the implementation. Therefore, the object layout can be tailored to the requirements of the particular implementation.

3.2 Creating New Objects

Type propagation of objects in a statically-typed language requires that the declared type of each object be available. This is the starting point for most optimizations based on static type analysis and propagation [DMM96]. Therefore object creation instructions need to be represented as a high-level construct in the intermediate code.

3.3 Method Invocation

Method invocation constructs are necessary as they are the primary targets of optimization in object-oriented languages. In case of execution using the JIT API they are implemented as calls to the virtual machine. In the stand alone case they are implemented as a method-table lookup. After optimization, these calls may be optimized to a single statically-bound procedure call if the type of the receiver object can be narrowed down to one choice. In the case where receiver class cannot be predicted it may be compiled as a case statement (if the choices are few).

3.4 Field Access

Field accesses are implemented as structure accesses at the low-level. Due to polymorphism the class of the object may not be known at compile time. Therefore in the case of JIT implementation these instructions will be translated to virtual machine calls. In the stand alone implementation these instructions are translated to a type check followed by a structure access. They can be optimized to direct field accesses if the object's type and layout is known.

3.5 Class Meta-Data

The dynamic nature of the language and the fact that object introspection is allowed in `Java` requires that the run-time environment maintain a detailed description of the classes defined in the program. This is also needed to ensure that appropriate data-structures can be created when the class is instantiated. In the normal interpretive version of `Java` the VM knows where the class meta-data is stored (`constant pool`). In compiled code it is necessary to create a representation of the meta-data where it is accessible to the VM (in case of JIT implementation). In a stand alone system the compiler has to make it possible for the run-time system to correctly initialize the data-structures before being used.

An object reference is implemented as a pointer to some data in the heap. The data structure could contain a pointer to a static data structure containing the meta-data for the class of that object. Some of the meta-data that are necessary are :

- Class Initialization Status
- Method Table
- Field Table
- Interfaces Table

- Access flags for class, methods and fields.

All the information for the class meta-data is available in the `constant pool`. This is implemented in the form of annotations to the file symbol table in the SUIF representation. The SUIF compiler pass would have to collect this information and statically allocate and initialize these internal data structures [Bot97].

3.6 Static/Class Methods and Fields

One of way of maintaining static or per-class information is by creating static data structures and having an appropriate naming scheme to identify the different structures.

Static or class methods and fields can also be supported by the presence of a meta-class. The meta-class is a per-class object. All instances of a class have a pointer to the meta-class information. This makes for a elegant design to store static fields and pointers to static methods. Meta-data can also be stored in the meta-class.

3.7 Support for Concurrency and Exceptions

SUIF does not support exceptions or threads which are a part of the standard `Java` library. This section presents proposed solutions to these issues.

3.7.1 Exception Handling

Exceptions are not required for object oriented programming but they are commonly used in several object oriented languages like `Java`, Ada and C++. In the implementation of these languages it is possible to create "zero-cost" exception handling systems, that is systems which have no runtime cost until an exception is raised. However, it is not possible to implement this type of exception handling without compiler support since it is necessary to know the addresses in the machine code spanned by the various exception blocks. This support is not present in the SUIF compiler, furthermore, it is not possible to support zero-cost exception handling and generate portable `C` code as an intermediate form.

Exception handling can be performed in other ways which are amenable to compilation to `C`, but these techniques incur a runtime cost. The simplest technique is to maintain an exception handler stack, when the computation enters a new exception block a continuation for the exception handler is pushed onto this stack using `C`'s `setjump` function. When the computation reaches the end of the exception block this stack is popped. This is the method which we have elected to use in the next implementation of `j2s` since it is the simplest to implement in SUIF.

Another issue which has complicated the addition of exception handling into the SUIF compilation process is the need to preserve the integrity of exception blocks. Many optimizations which are routinely applied in a compiler need to be handled differently in the presence of exceptions. For instance, a statement which could raise an exception can not be moved out of an exception block which has a handler for that type of exception since such a transformation would alter the "meaning" of the program. This presents a problem since the current optimizations performed by SUIF do not recognize exception blocks. The expedient solution that we have adopted is to simply not to use any of the built in optimizations. This does not currently pose a problem but it will be necessary to investigate the interactions between the new optimizations and the standard optimizations performed by SUIF in the future.

3.7.2 Multithreading

Each Java Virtual Machine can support many threads of execution at the same time. The threads independently execute code that operates on objects and other values residing in a shared memory.

The operations on threads are implemented as a library, therefore the only compiler support required is in synchronization of these threads.

The virtual machine supports the `synchronization` construct which helps to synchronize the concurrent activity of threads. Associated with each object is a lock. The lock is manipulated by the `monitorenter` and `monitorexit` bytecode instructions. These instructions can be handled by translating them in system calls or function calls into thread libraries that are specific to the platform.

4 Conclusions

We have described the process of translating a limited version of Java to SUIF. In the present version, `j2s` acts as a Java front-end for SUIF by translating bytecode to SUIF representation. We are able to add compilation support for object-oriented features without having to implement any extensions to SUIF. However, all object-oriented constructs are translated into procedure calls. On optimization, these procedure calls can be inlined. We describe the issues involved in supporting object-oriented languages in SUIF based our experience with the `j2s` translation tool. We propose a solution to supporting exceptions by using a run-time exception stack to maintain the `try` blocks. We also outline the issues in supporting threads in Java and propose a solution to handling the lock instructions.

The development of the `j2s` tool is part of a larger project to implement a new optimization scheme for object-oriented languages. There are other efforts at extending SUIF to support object-oriented languages. A notable project is the OSUIF project being undertaken at the University of California, Santa Barbara [OSU]. They are developing a scheme to add support for class representation, dynamic dispatch and exceptions to SUIF.

The following bytecode instructions are currently not supported:

- Exception Handling :`athrow, ret, jsr, jsr_w`
- Synchronization :`monitorenter, monitorexit`

We are currently in the final phase of implementing the first version of the j2s translator. In the next version we plan to implement extensions to SUIF to handle exceptions and threads.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[Bot97] Per Bothner. A gcc-based java implementation. In *IEEE Compcon '97*, 1997.

[Cha92] Craig Chambers. *The design and implementation of the SELF compiler. an optimizing compiler for object-oriented languages.* PhD thesis, Stanford University, March 1992.

[DDG+96] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *OOPSLA '96*, October 1996.

[Dea96] Jeffery Dean. *Whole-program optimization of object-oriented languages.* PhD thesis, University of Washington, Seattle, 1996.

[DMM96] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *OOPSLA '96*, 1996.

[Gro94] Stanford Compiler Group. *The SUIF Library.* Stanford University, 1994.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[OSU] OSUIF. http://www.cs.ucsb.edu/~osuif. OSUIF Home Page at UCSB.

[Pan96] Hemant Pande. *Compile time analysis of C and C++ programs*. PhD thesis, Rutgers, The State University of New Jersey, May 1996.

[Ple96] J. B. Plevyak. *Optimization of object-oriented and concurrent programs*. PhD thesis, University of Illinois, Urbana-Champaign, 1996.

[Smi96] Michael D. Smith. Extending SUIF for machine-dependant optimizations. In *First SUIF Compiler Workshop*, January 1996.

[WM95] Reinhard Willhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.

[Yel96] Frank Yellin. *The JIT compiler API.* Sun Microsystems Inc., October 1996. White paper.

[ZCG94] B. Zorn, B. Calder, and D. Grunwald. Quantifying differences between C and C++ programs. Technical Report CU-CS-698-94, University of Colorado, Boulder, January 1994.