j2s: A SUIF Java Compiler

Holger Kienle and Urs Hölzle
Department of Computer Science
University of California
Santa Barbara, CA 93106
http://www.cs.ucsb.edu/oocsb

Technical Report TRCS97–16

November 3, 1997

Abstract

We give a short overview of the Java bytecode-to-SUIF compiler that is currently being developed at UCSB. The goal of the compiler is to support the complete Java language specification, based on SUIF 2.0 and OSUIF. We expect to have a version of the compiler that is ready for release to the public at about the end of '97.

1 Introduction

This paper describes the design and preliminary implementation of a Java bytecode to SUIF 2.0 compiler. This compiler will be called `j2s` in the following. The `j2s` compiler compiles a single Java class file to SUIF 2.0 intermediate representation (*off-line compilation*).

The `j2s` compiler can take full advantage of the OSUIF/SUIF compiler framework, especially optimization passes and native code back ends. Thus, our compiler will hopefully serve as a useful tool for object-oriented compiler research.

The SUIF 1.0 compiler system [WFW+94] developed at Stanford University is a platform that supports experimental research on new compiler techniques. It consists of various tools and standard optimization passes that operate on a standardized and well documented intermediate format. The intermediate format of SUIF 1.0 only provides support for imperative languages. For example, only C and Fortran front-ends exist. SUIF is used in various research projects, but because of the limited intermediate format little research in the area of object-oriented programming languages has been done.

Currently the development of a "better" SUIF — SUIF 2.0 — is underway. The basic principles and functionality will stay the same, but the underlying object-oriented design and implementation is completely different. The SUIF 2.0 compiler infrastructure is part of the National Compiler Infrastructure project [NCI].

As part of this project, UCSB is building additional functionality for object-oriented languages on top of SUIF 2.0. This extended compiler system is called *Object SUIF*

(OSUIF) [DCI+97]. The idea of OSUIF is to provide a standardized library that allows researchers to express object-oriented behavior in SUIF, thus providing a framework that makes it possible to share optimizations that are specific to object-oriented languages.

The `j2s` compiler is the first larger project that uses SUIF 2.0 and the first front-end that will support an object-oriented language by making use of OSUIF. Thus, we hope that our compiler provides valuable input for the design, implementation and verification/testing of OSUIF and, to some extent, for SUIF 2.0. It is planned that `j2s` will eventually become part of the SUIF distribution.

The goal of the compiler is to support the complete Java language specification [GJS96]. This requires the implementation of a runtime environment that includes garbage collection, threads, exception handling, Java monitors, object creation/finalization, class initialization, etc.

The remaining sections are organized as follows. Section 2 discusses the `j2s` compiler in detail. Section 3 describes first experiences with SUIF 2.0. Section 4 gives the current status and future work of the project.

We assume that the reader is familiar with Java [GJS96] and the Java Virtual Machine (JVM) [LY96], as well as the SUIF 1.0 and SUIF 2.0 compiler systems.

2 The `j2s` Compiler

The `j2s` compiler expects a valid[1] Java class file (not Java source), analyzes the class file and produces (optimized) SUIF output.

Because SUIF (and hence OSUIF) are implemented in C++, the `j2s` compiler is implemented in the same language.

Figure 1 shows how the `j2s` compiler is integrated into the SUIF framework.

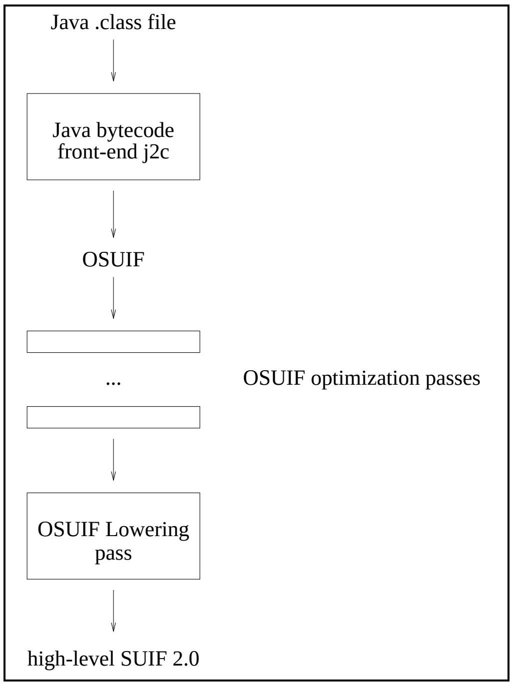

Figure 1: Java bytecode front-end

The following sections briefly explain the translation process of the compiler. Figure 2 gives an overview of the translation steps.

2.1 Class Loader

The *class loader* reads a Java class file and stores the various components of the class file into an object hierarchy. Every component of the *class file format*, like fields, methods, attributes, and various different entries of the *constant pool* are represented by a C++ class. The class loader is the only part of the compiler that is directly concerned with file han-

[1] A large subset of the bytecode verifier and the static and structural constraints that the JVM poses on bytecodes have been implemented (mainly for debugging purposes).

dling and I/O.

At this stage the original structure of the class file remains unchanged. This means that, for example, the constant pool and indexes in the constant pool are kept (i.e. the constant pool is not *resolved*). Furthermore, field and method *descriptors* are kept in the original representation as UTF-8 strings.

The class loader reads in a single class file at a time. Because it does not do any resolution, the class loader does not need to access class files of other classes for a complete representation.

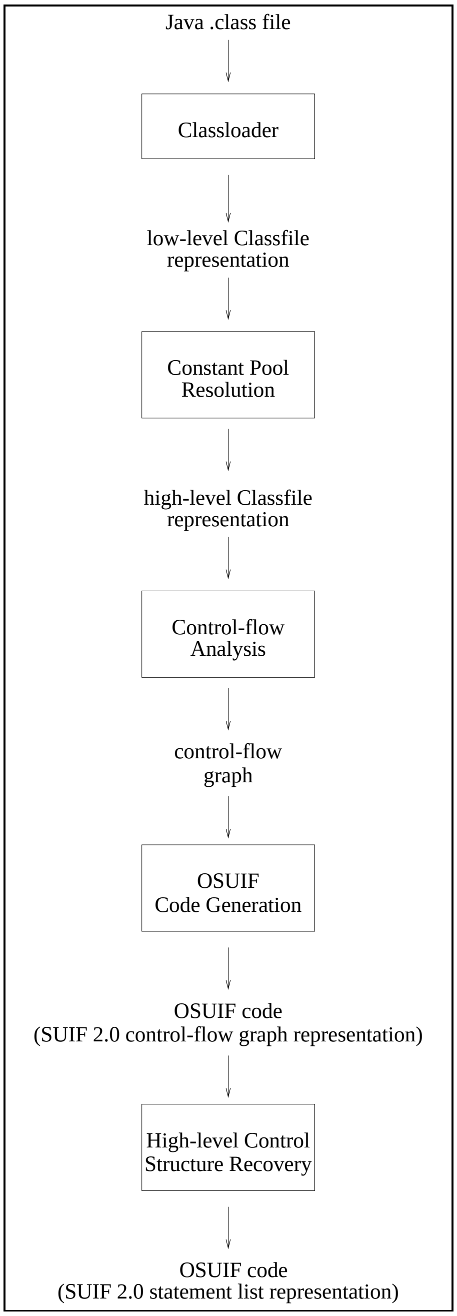

Figure 2: `j2s` translation passes

2.2 Constant Pool Resolution

The *resolver* uses the representation of the class loader to construct a high-level view of the Java class file. The constant pool is completely eliminated by resolving all references to it. Whenever a reference to another Java class file is encountered during resolution, this class file is loaded (and in turn resolved).

The resolver keeps a dictionary of all (transitively) loaded classes. Because the dictionary guarantees that every class is only loaded once, the equality check of two classes can be simply done by comparing their references.

The resolver provides a convenient, high-level representation of the class file information. For example, references to a Java class are no longer indexes to UTF-8 strings that contain the fully qualified name of the class. Instead, classes are represented by a (unique) reference to an instance of the C++ class `jhl_Unit` that represents a class, interface, or array class.

The JVM stores field and method designators in UTF-8 string constants. When the resolver encounters a designator string, it parses the designator string and builds a *type tree*[2]. Parsing of the designator strings is very easy

[2] Actually it is just a "type list."

because they can be described with a *Simple LL(1)* grammar.

The only component that is kept in its original representation is the *bytecode array*, but the implementation hides this from the programmer by providing a bytecode *iterator* and various bytecode *visitors* [GHJV94]. The bytecode iterator traverses the bytecode array sequentially and calls for every bytecode the corresponding (virtual) method of the visitor. For example, the resolver makes use of this mechanism by subclassing a print visitor from the abstract visitor class. This class prints out a textual representation of the class file that is similar to that of `javap` or `guavad`. This design decision preserves the benefits of having a compact representation of the bytecode array while providing a convenient interface for accessing it.

2.3 Control Flow Analysis

The control flow analysis uses the representation provided by the resolver to construct a control flow graph of a Java bytecode array.

In order to construct the control flow graph two steps must be performed:

1. Partitioning of the bytecode array into *basic blocks*. The basic blocks are the nodes of the control flow graph.

2. Computation of the edges of the control flow graph. The edges are determined by the last bytecode of a basic block. For example, a branch instruction with one target creates two edges in the graph. One edge goes to the following basic block, and the other edge goes to the basic block whose first bytecode is the branch target. The only exception is the `ret` bytecode. The possible targets of this instruction are not known at this point of the analysis (similar to an *indirect jump*). Therefore these targets are computed later during dataflow analysis (see below).

Part of the JVM structure is a so-called *frame*. A new frame is created when a Java method is invoked, and is destroyed after completion of the method. Each frame has its own *local variable array* and its own *operand stack*. A Java frame is very similar to an *activation record* of an imperative language.

In a class file all type information of local variables and stack entries is lost. However, it can be recovered with dataflow analysis[3]. This is possible because every bytecode implicitly specifies the types of the stack values and local variable slots that it manipulates.

For instance, the bytecode `iadd` expects two integer operands at the top of the stack — other types are illegal and would cause the analysis to fail. Both operands are popped and the integer sum of both operands is pushed on the stack.

In order to generate code for a single bytecode, it is sufficient to know the size of the local variable array and operand stack and the type of every local variable slot and operand stack entry. We call this information the *configuration* of the local variable array and operand stack, respectively.

The dataflow analyzer symbolically executes all execution paths of a program and records the configuration of the local variable array and operand stack at the beginning of every basic block. Every bytecode manipulates the local variable array and/or the operand stack in a well-defined way. The dataflow analysis simulates the behavior of the bytecodes by adjusting the configuration of the local variable array and the operand stack accordingly.

If control flow *merges* (i.e., more than one edge reaches a basic block) then the local variable array and operand stack must also be

[3] A similar dataflow analysis is implemented as part of the Java *bytecode verifier*.

merged. The merging procedure is briefly described in [LY96].

The bytecode array can contain *dead code* inside of a basic block. This is detected and the dead code is eliminated. If control flow analysis and merging reveals that the whole basic block is unreachable, the basic block is eliminated.

2.4 Code Generation

As stated before, code generation needs to know the configuration of the local variable array and operand stack for every bytecode. In the following this will be called the *configuration of the bytecode.* The control flow analysis pass (described in the previous section) computes this information.

Generation of SUIF code can be done for every basic block separately, because every block knows the configuration for its first bytecode. After code has been generated for the first bytecode, the configuration for the second bytecode is known and so on.

Code is generated by *symbolic execution* of the operand stack, a well-known approach that maps the stack values in the stack-oriented bytecodes to a set of (temporary) variables or machine registers. For every push and pop of a stack value a corresponding assignment instruction with a new temporary variable is generated.

This approach tends to create lots of temporary variables and does not guarantee that all created variables are used by successive code ("dead statement"). The philosophy of the code generation is to keep things as simple as possible and to rely on the SUIF optimization passes to improve the code. We verified that redundant temporary variables can be easily detected and eliminated with the following SUIF 1.0 optimization passes: (1) copy propagation (2) constant propagation, and (3) set-use analysis with successive dead code elimination.

A local variable slot can be expressed directly with a corresponding SUIF variable. The JVM specification allows (re-)using of a variable slot with different types at different program points[4]. Because SUIF variable symbols are typed, a separate SUIF variable with the corresponding type must be created for every type that the local variable slot can hold.

The code generation pass creates code in a straightforward manner for every bytecode without taking the surrounding code into account. Every Java bytecode is translated to a single SUIF instruction or instruction AST. (The AST representation allows us to use instructions as operands for another instruction.) The current implementation generates code in the SUIF instruction list representation. The final implementation of the code-generation will create SUIF code that uses the control flow graph representation. This means that the constructed control flow graph from the resolver is transformed to a structurally equivalent SUIF control flow graph during code generation.

2.5 High-Level Control Structure Recovery

The JVM supports relative conditional and unconditional *branches* that allow to continue execution with an instruction other the one following the branch instruction. Because these branch instructions do not adhere to any nesting rules, the resulting control flow graph can be *unstructured.*

The goal of the high-level control structure recovery is to recover high-level control flow constructs (i.e. loops, conditional statements, and short-circuit operators) from the (possibly unstructured) SUIF control flow graph.

[4] The Java compiler of the JDK 1.1.3. does not make use of this feature.

The optimization pass for short-circuit operators looks for certain patterns in the control flow graph that correspond to short-circuit evaluations of expressions. Our technique is similar to the one described in [Cif96].

The goto elimination pass checks first whether the control flow graph is *reducible*. If it is, we recover high-level constructs from the flow-graph. This step involves a transformation from the control flow graph to the statement list representation.

The high-level recovery will be implemented as a SUIF pass that operates upon the control flow representation of SUIF.

3 First Experiences with SUIF 2.0

The first prototype of the `j2s` code generation pass used the SUIF 1.0 system. It basically supported the imperative ("C-like") subset of the Java language. An expression tree was constructed for every Java bytecode and a sequence of bytecodes was kept in the "flat lists of instructions" representation.

After the code generation for SUIF 1.0 reached a stable point, we migrated to SUIF 2.0. The rewriting of the structural framework was accomplished in three days, the rewriting of the generated instructions for the bytecodes in about two weeks. (No documentation was available at that time and the interface was not completely stable.) In our opinion, the migration from SUIF 1.0 to SUIF 2.0 was painless. SUIF 2.0 has a cleaner and more consistent interface, while still preserving familiar concepts from SUIF 1.0.

This migration provided a first validation of the design of the new system and allowed early debugging of its implementation.

The `j2s` compiler will use a fairly large subset of the available SUIF 2.0 functionality. So far only the instruction list representation is used, but the final compiler will make use of all three representations.

4 Status and Future Work

The "C-like" subset of the Java bytecodes has been implemented and we generate all the data types and variable definitions for class instances and meta data information.

The missing parts can be divided into runtime functionality and features that are supported by OSUIF. OSUIF code is generated for bytecodes that have object-oriented features (e.g. `invokevirtual` and `invokeinterface`). Because OSUIF is still in its design stage and not fully implemented, this functionality is not available yet. We expect that `j2s` will provide valuable feedback in the OSUIF design process.

Still missing are essential parts of the runtime support for the JVM, i.e. exception handling, garbage collection, Java monitors (`monitorenter` and `monitorexit`) and threads. We plan to focus on exception handling first because it is tightly coupled with the control flow graph representation and code generation. The control flow graph has to be augmented with special "exception-edges" that correspond to exception handling dataflow. The runtime system must be able to unwind Java frames and to activate the proper exception handler.

The runtime system will implement the *Java Native Interface* (JNI) [JNI97]. This allows interoperability with applications and libraries written in other languages, such as native method calls in the Java JDK.

No work has been done for the actual SUIF passes that will perform the high-level control structure recovery. As a first step, we implemented an algorithm that decides if the control flow graph is reducible. This algorithm does not depend on SUIF 2.0 (because the system had the required functionality not implemented at this point). It operates on the control flow graph of the resolver.

As the next step garbage collection will be supported. We plan to use a *conservative garbage collector* (e.g., the freely available Boehm-Demers-Weiser GC [BDW]) as a first approach.

Furthermore, OSUIF optimization passes can be written that especially improve the code quality of object-oriented features. OSUIF passes are usually generic, which means that they can be used without modification to optimize OSUIF code that has been generated from other source languages. For example, such an optimization pass could perform *type inference*.

We expect to have a version of the compiler that is ready for release to the public at about the end of '97. This version will have no support for threads/synchronization and exception handling[5].

5 Related Work

Several (de)compilers that translate Java bytecode or Java source to other target languages are available or in development. Mocha [vV] and Krakatoa [PW97] are Java bytecode to Java source decompilers. CACAO [KG97] and Briki [CL97] are both JIT-compilers for Java bytecode. Toba [PTB+97] is a system for generating stand-alone Java applications developed at the University of Arizona. It includes a Java bytecode to C compiler that seems to cover the whole Java language specification. Harissa [MMBC97] tries to reconcile JIT and off-line compilers by permitting to mix compiled and interpreted code. Cygnus Solutions is developing a `gcc`-based Java implementation [Bot97]. A new `gcc` front-end, `cc1java`, translates Java bytecode or Java source to the intermediate representation of `gcc` [Sta95]. Colorado State University is developing a Java front-end for the SUIF 1.0 compiler system [MDG97]. Ironically, but not too surprisingly, they are calling it `j2s` as well. Because SUIF 1.0 has no support to describe object-oriented constructs, annotation have been used to model them. The last two compiler projects are most closely related to our own compiler.

[5] Maybe we provide an inefficient exception handling implementation based on `setjump/longjump`.

Acknowledgments

Many thanks to Chris Wilson for patiently answering lots of questions related to SUIF 2.0 that came up while implementing the `j2s` compiler.

This work is supported in part by National Science Foundation CAREER grant CCR-9624458 and by Sun Microsystems and the State of California MICRO program.

References

[BDW] Boehm–Demers–Weiser garbage collector. URL: http://reality.sgi.com/boehm/gc.html.

[Bot97] Per Bother. A Gcc-based Java implementation. *IEEE Compcon*, February 1997.

[Cif96] Cristina Cifuentes. Structuring decompiled graphs. *Lecture Notes in Computer Science Vol. 1060 — Compiler Construction*, pages 91–105, 1996.

[CL97] Michael Cierniak and Wei Li. Briki: An optimizing Java compiler. *IEEE Compcon*, February 1997.

[DCI+97] A. Duncan, B. Cocosel, C. Iancu, H. Kienle, R. Rugina, U. Hölzle, and M. Rinard. OSUIF: SUIF with objects. *Proceedings of the Second SUIF Compiler Workshop*, pages 1–7, August 1997.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1996.

[JNI97] JavaSoft. *Java Native Interface Specification, Release 1.1*, January 1997.

[KG97] Andreas Krall and Reinhard Grafl. CACAO — a 64bit Java VM just-in-time compiler. *PPoPP Workshop on Java for Science and Engineering Computation*, June 1997.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1996.

[MDG97] Sumith Mathew, Eric Dahlman, and Sandeep Gupta. Compiling Java to SUIF: Incorporating support for object-oriented languages. Technical Report CS-97-114, Computer Science Department, Colorado State University, July 1997.

[MMBC97] Gilles Muller, Barbara Moura, Fabrice Bellard, and Charles Consel. Harissa: a flexible and efficient Java environment mixing bytecode and compiled code. *COOTS '97*, June 1997.

[NCI] The national compiler infrastructure project. URL: http://suif.stanford.edu/suif/NCI.

[PTB+97] T. Proebsting, G. Townsend, P. Bridges, J. Hartman, T. Newsham, and S. Watterson. Toba: Java for applications — a way ahead of time (WAT) compiler. *COOTS '97*, June 1997.

[PW97] Todd Proebsting and Scott Watterson. Krakatoa: Decompilation in Java. *COOTS '97*, June 1997.

[Sta95] Richard M. Stallman. *Porting GNU CC*. Free Sofware Foundation, Boston, MA, November 1995.

[vV] H. P. van Vliet. The Mocha decompiler. URL: http://www.brouhaha.com/~eric/computers/mocha.html.

[WFW+94] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, pages 31–37, December 1994.