© 2014 IJIRT | Volume 1 Issue 5 | ISSN : 2349-6002

A Virtual Machine: Joeq

Kanika Jindal, Lubna Siddiqui
IT Department
Dronacharya colg of engg., ggn, India

Abstract- We will be talking about a virtual machine that is joeq. A joeq has a compiler infrastructure to help in research on the virtual machines such as the most common problem of garbage collection, run time techniques and many more problems. The component of virtual machine is independently written with a simple defined interface which makes it easier to experiment new ideas for the new innovation. As it is language independent, so its code is easily compiled, linked and is dynamically executed.

I. INTRODUCTION

Joeq is a virtual machine and compiler infrastructure designed to be a platform for research in compilation and virtual machine technologies. We had three main goals in designing the system. First and foremost, we wanted the system to be flexible. We are interested in a variety of compiler and virtual machine research topics, and we wanted a system that would not be specific to researching a particular area. For example, we have interest in both static and dynamic compilation techniques, and in both type-safe and unsafe languages. We wanted a system that would be as open and general as possible, without sacrificing usability or performance.

Second, we wanted the system to be easy to experiment with. As its primary focus is research, it should be straightforward to prototype new ideas in the framework. With this in mind, we tried to make the system as modular as possible so that each component is easily replaceable. Learning from our experience with Jalapeno, another˜ virtual machine written in Java, we decided to implement the entire system in Java. This makes it easy to quickly implement and prototype new ideas, and features like garbage collection and exception tracebacks ease debugging and improve productivity. Being a dynamic language, Java is also a good consumer for many of our dynamic compilation techniques; the fact that our dynamic compiler can compile the code of the virtual machine itself means that it can dynamically optimize the virtual machine code with respect to the application that is running on it. Java's object-oriented nature also facilitates modularity of the design and implementation.

Third, we wanted the system to be useful to a wide audience. The fact that the system is written in Java means that much of the system can be used on any platform that has an implementation of a Java virtual machine. The fact that Joeq supports popular input languages like Java bytecode, C, C++, and even x86 binary code increases the scope of input programs. We released the system on the SourceForge web site as open source under the Library GNU Public License. It has been picked up by researchers for various purposes including: automatic extraction of component interfaces, static whole-program pointer analysis, context-sensitive call graph construction, automatic distributed computation, versioned type systems for operating systems, sophisticated profiling of applications, advanced dynamic compilation techniques, system checkpointing, anomaly detection, Java operating systems, secure execution platforms and autonomous systems. Joeq supports two modes of operation: native execution and hosted execution. In native execution, the Joeq code runs directly on the hardware. It uses its own run-time routines, thread package, garbage collector, etc. In hosted execution, the Joeq code runs on top of another virtual machine. Operations to access objects are translated into calls into the reflection library of the host virtual machine. The user code that executes is identical, and only a small amount of functionality involving unsafe operations is not available when running in hosted execution mode. Hosted execution is useful for debugging purposes and when the underlying machine architecture is not yet directly supported by Joeq. We also use hosted execution mode to bootstrap the system and perform checkpointing, a technique for optimizing application startup times.

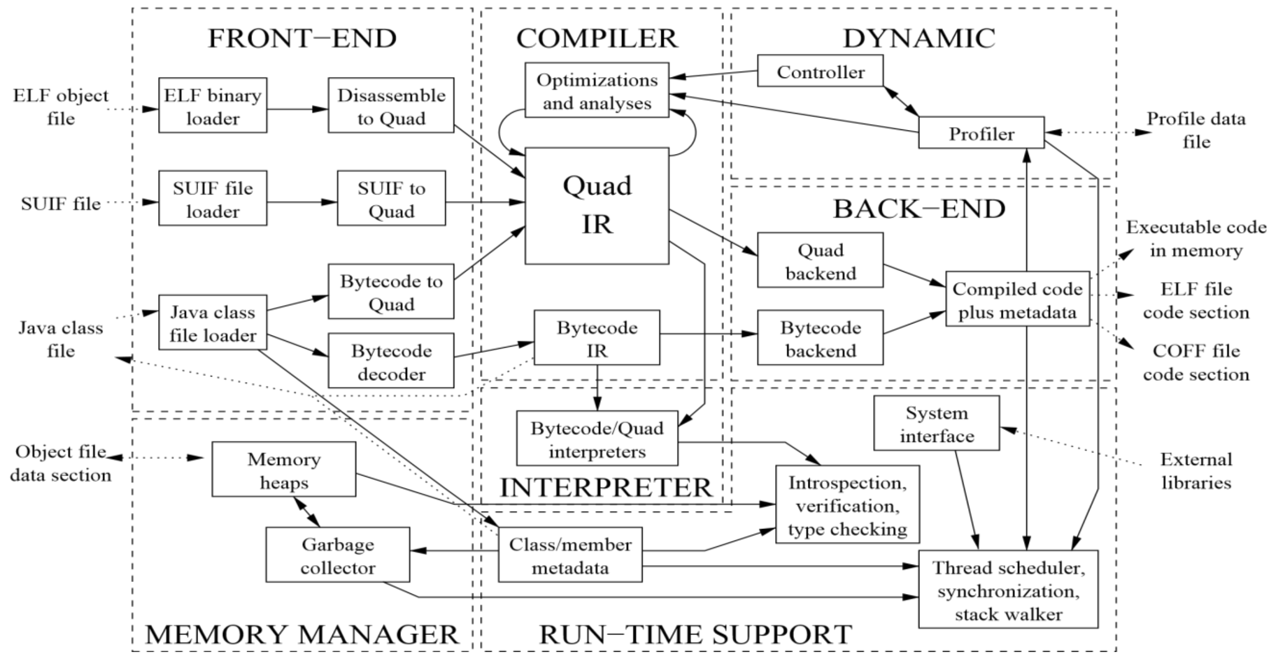


Figure 1 Overview of the Joeq system. Arrows between blocks signify either the flow of data between components, or the fact that one component uses another component.

II. OVERVIEW

As shown in Figure 1, the Joeq system consists of seven major parts:

Front-end: Handles the loading and parsing of input files such as Java class files, SUIF files, and binary object files.

Compiler: A framework for performing analyses and optimizations on code. This includes the intermediate representation (IR) of our compiler.

Back-end: Converts the compiler's intermediate representation into native, executable code. This code can be output to an object file or written into memory to be executed. In addition, it generates metadata about the generated code such as garbage collection maps and exception handling information.

Interpreter: Directly interprets the various forms of compiler intermediate representations.

Memory Manager: Organizes and manages memory. Joeq supports both explicitly-managed and garbage-collected memory.

Dynamic: Provides profile data to the code analysis and optimization component, makes compilation policy decisions, and drives the dynamic compiler.

Run-time Support: Provides runtime support for introspection, thread scheduling, synchronization, exception handling, interfacing to external code, and language-specific features such as dynamic type checking.

III. FRONT-END

The front-end component handles the loading and parsing of input files into the virtual machine. Joeq has support for three types of input files: Java class files, SUIF intermediate representation files, and ELF binary files.

The Java class loader decodes each Java class file into an objectoriented representation of the class and the members it contains. Our class loader fixes many of the nonuniformities and idiosyncrasies present in Java class files. For example, Joeq makes a distinction at the type level between static and instance fields and methods; i.e. there are separate classes for instance methods and static methods and likewise for fields. In the Java class file representation, there is no distinction between member references to static and instance members. We handle this by deferring the creation of the object representing the field or method until we are actually forced to resolve the member, at which point we know whether it is static or instance. We also explicitly include the implicit "this" parameter in the parameter list for instance methods, so code can treat method parameters uniformly.

The SUIF loader loads and parses SUIF files, a standard intermediate format that is widely used in the compiler research community. There are SUIF front-ends available for many languages including C, C++, and Fortran. This allows Joeq to easily load and compile many languages.

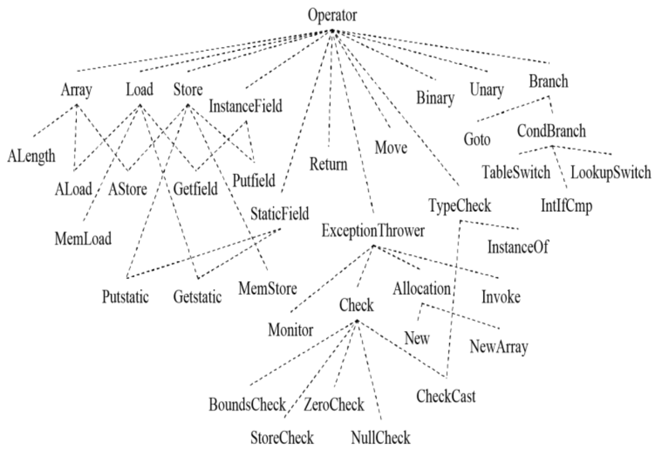
The ELF binary loader can load and decode x86 object files, libraries, and executable images in the popular ELF format. The front-end also includes an intelligent x86 disassembler, which can disassemble the binary code for a function, undoing stack spills and converting the code into operations on pseudo-registers. It also recognizes some common control flow paradigms. This allows Joeq to seamlessly load and analyze binary code as if it were just another front-end.

IV. CODE ANALYSIS AND OPTIMIZATION

One of the goals of the Joeq infrastructure is a unified framework for both static and dynamic compilation and analyses. Furthermore, we would like to support a wide variety of input languages, from high-level languages like Java all the way down to machine code. However, we would still like to be able to explicitly represent high-level operations in the IR to facilitate sophisticated, languagedependent analyses and optimizations. The compiler framework was designed with all of these goals in mind.

V. THE QUAD IR

The major intermediate representation in Joeq is the Quad format. The Quad format is a set of instructions, called Quads, that are organized into a control flow graph. Each Quad consists of an operator and up to four operands. For efficiency, Quads are implemented as a final class; polymorphism is implemented in the different operators and operands. However, we retain most of the benefit of polymorphism on quads by utilizing strict type checking on the operator type. Operators are implemented using the singleton pattern, so there is only one instance of each operator, which is shared across multiple Quads.



VI. THE BYTECODE IR

In addition, to experiment with rapid code generation from bytecode and also to leverage existing bytecode analyses and tools, Joeq includes a bytecode IR, which corresponds directly to the bytecode in the Java class file. The major difference between the bytecode IR and the Quad IR is that the bytecode IR is stack-based, while the Quad IR is register-based. The design of the bytecode framework is based on the Byte Code Engineering Library (BCEL), a popular open-source library for analyzing and manipulating bytecode.

VII. GENERALIZED INTERFACE

Both the bytecode and the Quad intermediate representations implement a single, generalized compiler interface. Individual bytecodes and Quads implement the CodeElement interface, which provides basic functionality such as finding possible successors and returning the uses and definitions. Code written to this generalized compiler interface will work regardless of the specific IR being used. This allows the implementation of many data-flow analyses, such as calculating def-use chains and dead code elimination, to be shared between the two IRs

VIII. DATAFLOW FRAMEWORK

Joeq includes a standardized dataflow framework. Dataflow problems are specified by subclassing the abstract Dataflow.Problem class. The abstract methods of this class include the standard specification of a dataflow problem: direction, dataflow boundary condition, initial dataflow value on interior points, transfer function, and confluence function. There are also abstract interfaces for pieces of dataflow information (the Dataflow.Fact interface) and transfer functions (the Dataflow.TransferFunction interface).

boolean *direction*()
Returns the direction of this dataflow problem (true=forward, false=backward).

Fact *boundary*()
Returns the dataflow boundary condition.

Fact *interior*()
Returns the initial dataflow value on interior points.

TransferFunction *getTransferFunction*(CodeElement) Returns the transfer function for the given code element.

IX. INTERPROCEDURAL ANALYSIS FRAMEWORK

Joeq includes significant support for performing interprocedural analysis through the use of an advanced call graph interface. The call graph interface supports both precomputed and on-the-fly call graphs with both partial-program and whole-program compilation models. It includes support for profile information to be attached to call graph edges. It also supports the use of context information at call sites, so it can distinguish between calls made under different contexts.

Joeq includes code to perform many common graph algorithms such as finding dominators, calculating a reverse post-order or finding strongly-connected components. The graph algorithms are all written to a generic Graph interface, which is implemented by all graphs in Joeq, including the call graph and the control flow graph. This allows the programmer to easily perform traversals and calculations over any type of graph

X. BACK-END

Joeq includes back-end assemblers that generate executable code from the compiler’s intermediate representation. In addition to the executable code, the back-ends generate metadata about the code, such as reference maps for garbage collection, exception tables, line numbers for debugging and generating exception tracebacks, and the locations of heap and code references in the code. This metadata is used by the runtime system and garbage collector, and to support code relocation. To allow for efficient generated code, the backend allows absolute memory references to be in the code. If the code or the referenced object moves due to compaction in the garbage collector, the absolute reference is updated. The backend also has generalized support for atomically patching code fragments in a thread-safe manner without having to perform synchronization operations

XI. INTERPRETER

Joeq includes interpreters for both the Quad IR and the bytecode IR. These interpreters are implemented using the visitor design pattern. This makes it easy to modify the interpreter to gather profiling information — simply make a new subclass of the interpreter that overrides the visit methods corresponding to the types of instructions that you care about.

Both interpreters have two modes of interpretation: direct and reflective. In direct interpretation, the interpreter uses the same stack as the compiled code, reading and writing values through direct memory accesses.

XII. MEMORY MANAGER

Joeq includes a general framework for memory management. It supports both managed (explicitly allocated and deallocated) and unmanaged (garbage collected) memory. The interface is based on the Java Memory Toolkit (JMTk) for the Jikes RVM. It supports a wide variety of garbage collection techniques such as compacting versus non-compacting, exact versus conservative, generations, concurrency, and reference counting. The specifications of the memory manager are accessible through the GCInterface interface, which includes query methods on whether garbage collection requires safe points and the nature of those safe points, whether objects can move or not, whether the collector supports conservative information, what types of read/write barriers are necessary, and interfaces to the allocator for various types of object allocations.

XIII. DYNAMIC RECOMPILATION

In native execution mode, Joeq supports dynamic recompilation based on profile information. Joeq includes two profilers. The first is a sampling profiler, which collects information about the timeconsuming methods by periodically sampling the call stacks of the running threads. The sampling profiler supports the collection of context-sensitive sampling information through the use of a partial calling context tree[30]. The sampling profiler is integrated into the thread scheduler, which is described in the next section. The sampling profiler is useful because it is easy to adjust the trade-off between overhead and accuracy by varying the sampling rate.

XIV. RUN-TIME SUPPORT

Joeq contains implementations of many necessary runtime routines written in Java. It includes a complete reflection and introspection mechanism. When Joeq is running in native mode, the reflection implementation directly accesses memory. When Joeq is running in hosted mode, reflection calls in Joeq get mapped to the corresponding reflection calls on the host virtual machine. The interpreter always accesses data through the reflection mechanism, and therefore works seamlessly in both native mode and hosted mode.

XV. CONCLUSION

Joeq is a virtual machine and compiler infrastructure designed to be a platform for research in compilation and virtual machine technologies. It was designed to be flexible, easy to experiment with and useful to a wide audience. It supports a variety of input languages and output formats, both dynamic and static compilation and both explicitly-managed and garbage-collected memory. It is completely written in Java and supports both native execution and hosted execution on another virtual machine. The design is modular and it is easy to replace components with different implementations to try out new ideas.

REFERENCES

[1] J. WHALEY. SYSTEM CHECKPOINTING USING REFLECTION AND PROGRAM ANALYSIS. IN PROCEEDINGS OF REFLECTION 2001, THE THIRD INTERNATIONAL CONFERENCE ON METALEVEL ARCHITECTURES AND SEPARATION OF CROSSCUTTING CONCERNS, VOLUME 2192 OF LNCS, PAGES 44–51, KYOTO, JAPAN, SEPT. 2001. SPRINGER-VERLAG.

[2] J. WHALEY AND M. S. LAM. AN EFFICIENT INCLUSION-BASED POINTSTO ANALYSIS FOR STRICTLY-TYPED LANGUAGES. IN PROCEEDINGS OF THE 9TH INTERNATIONAL STATIC ANALYSIS SYMPOSIUM (SAS'02), PAGES 180–195, SEPT. 2002.

[3] J. WHALEY, M. C. MARTIN, AND M. S. LAM. AUTOMATIC EXTRACTION OF OBJECT-ORIENTED COMPONENT INTERFACES. IN PROCEEDINGS OF THE ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS (ISSTA'02), PAGES 218–228. ACM PRESS, JULY 2002.

[4] J. WHALEY AND M. RINARD. COMPOSITIONAL POINTER AND ESCAPE ANALYSIS FOR JAVA PROGRAMS. IN PROCEEDINGS OF THE ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS (OOPSLA'99), PAGES 187–206. ACM PRESS, NOV. 1999.

[5] F. YERGEAU. RFC 2279: UTF-8, A TRANSFORMATION FORMAT OF ISO 10646, JAN. 1998.

[6] HTTP://WWW.NOVELLSHAREWARE.COM/INFO

[7] HTTP://JOEQ.SOURCEFORGE.NET/GETTING_STARTED.HTML