Joeq: A Virtual Machine and Compiler Infrastructure

John Whaley
Computer Systems Laboratory
Stanford University
Stanford, CA 94305
jwhaley@stanford.edu

Abstract

Joeq[1] is a virtual machine and compiler infrastructure designed to facilitate research in virtual machine technologies such as Just-In-Time and Ahead-Of-Time compilation, advanced garbage collection techniques, distributed computation, sophisticated scheduling algorithms, and advanced run time techniques. Joeq is entirely implemented in Java, leading to reliability, portability, maintainability, and efficiency. It is also language-independent, so code from any supported language can be seamlessly compiled, linked, and executed — all dynamically. Each component of the virtual machine is written to be independent with a general but well-defined interface, making it easy to experiment with new ideas. Joeq is released as open source software, and is being used as a framework by researchers all over the world on topics ranging from automatic distributed virtual machines to whole-program pointer analysis.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.4 [**Programming Languages**]: Processors—*Compilers, Interpreters, Memory management, Run-time environments*

1 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.

[1]Joeq (pronounced like the name "Joe" and the letter "Q") means "advanced level" in Japanese.

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 Jalapeño, 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[34], static whole-program pointer analysis[33], context-sensitive call graph construction, automatic distributed computation, versioned type systems for operating systems, sophisticated profiling of applications[30], advanced dynamic compilation techniques[31], system checkpointing[32], anomaly detection[19], Java operating systems, secure execution platforms and autonomous systems[8]. In addition, Joeq is now used as the basis of the Advanced Compilation Techniques class taught at Stanford University.

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[32], a technique for optimizing application startup times.

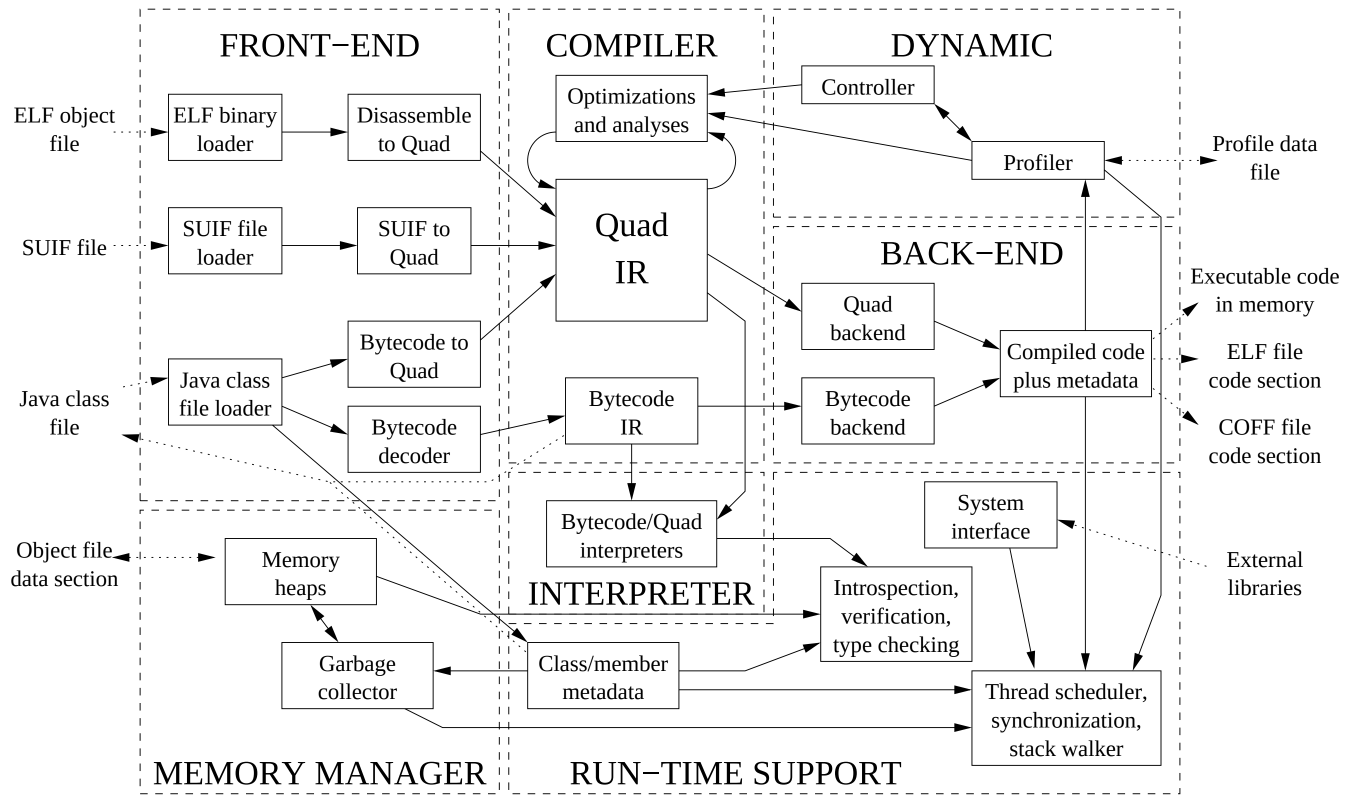

Figure 1. Overview of the Joeq system. Arrows between blocks signify either the flow of data between components, or the fact that one component uses another component.

The remainder of this paper is organized as follows. Section 2 gives an overview of the Joeq system. Sections 3 through 9 cover each of the components in detail. Section 10 covers some related work, and section 11 discusses the state of the project and some future directions.

2 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.

Sections 3 through 9 cover each of the components in detail.

3 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[22], SUIF intermediate representation files[2], and ELF binary files[27].

The Java class loader decodes each Java class file into an object-oriented 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[2]. There are SUIF front-ends available for many languages including C, C++, and Fortran.[2] 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[27]. 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.

All three of these formats are converted into a unified intermediate representation called the Quad form, which is based on pseudo-registers and is covered in more detail in the next section. Because all inputs lead to a unified format, all analyses and optimizations on that format can be performed uniformly across all of the different types of code. This allows us, for example, to inline Java Native Interface (JNI) C function implementations into their callers[25], or analyze arbitrary library calls to see if a passed-in reference can be written to another location. This is especially powerful because it allows us to avoid a lot of redundant checks and marshalling/unmarshalling of arguments and lets analyses and optimizations avoid having to make conservative assumptions about cross-language procedure calls.

4 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, language-dependent analyses and optimizations. The compiler framework was designed with all of these goals in mind.

4.1 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.

The control flow graph in the Quad format does not explicitly represent control flow edges due to exceptions. Instead, edges due to exceptional control flow are *factored* so that there is only one "exception" edge for each basic block[11]. This exception edge points to the exception handlers that can catch exceptions thrown in the block. This means that, due to exceptions, control flow can potentially exit from the middle of a basic block. However, this greatly reduces the number of basic blocks and control flow graph edges and thereby makes compilation more efficient.

[2]So far, we have only attempted to load SUIF files that were compiled from C and C++. Although other languages should theoretically work correctly, they have not been tested yet.

Operators in the Quad format fall into three classes: high, middle, and low. High-level operators correspond to complicated, often language-dependent operations, such as array accesses, method invocations, or dynamic type checks. These operations operate solely on symbolic references and are therefore independent of the virtual machine and object memory layout. Middle-level operators correspond to more basic operations such as loads and stores to calculated addresses. These operations are dependent on the memory layout but are still independent of CPU type. Finally, low-level operators correspond to machine code instructions. At this low level, pseudo-registers are replaced by physical registers, and code can be generated.

As compilation proceeds, translation passes on the Quad format replace higher-level operations with equivalent sequences of lower-level operations. However, the basic data structures stay the same, so one can uniformly execute compiler passes on the IR regardless of its level. This maximizes code reuse in the compiler.

The different types of operators in the Quad format form a hierarchy, as seen in Figure 2. Analyses can use `instanceof` tests or the visitor pattern[7, 17] to select the Quads that match certain characteristics.

The Quad IR supports the inclusion of optional supplementary information such as profile data and mappings back to line numbers in the source code. These are implemented as extensions to the IR and are not required for correct operation. Extensions implement a standard interface that specifies how to update the extra data when performing various IR transformations. This allows the compiler to automatically and implicitly update data even across IR transformations.

4.2 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[13]. We took the opportunity to clean up some of the BCEL interfaces; for example, by construction BCEL uses separate objects for loaded classes versus classes that can be modified — Joeq merges these into a single object. Most code written to use BCEL will work with Joeq with only a few trivial modifications. Joeq can also output Java class files using the bytecode IR along with the class and member metadata. For simple code analyses and transformations, the bytecode IR is very efficient because it avoids the step of converting the stack-based bytecode to the register-based Quad format.

4.3 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.

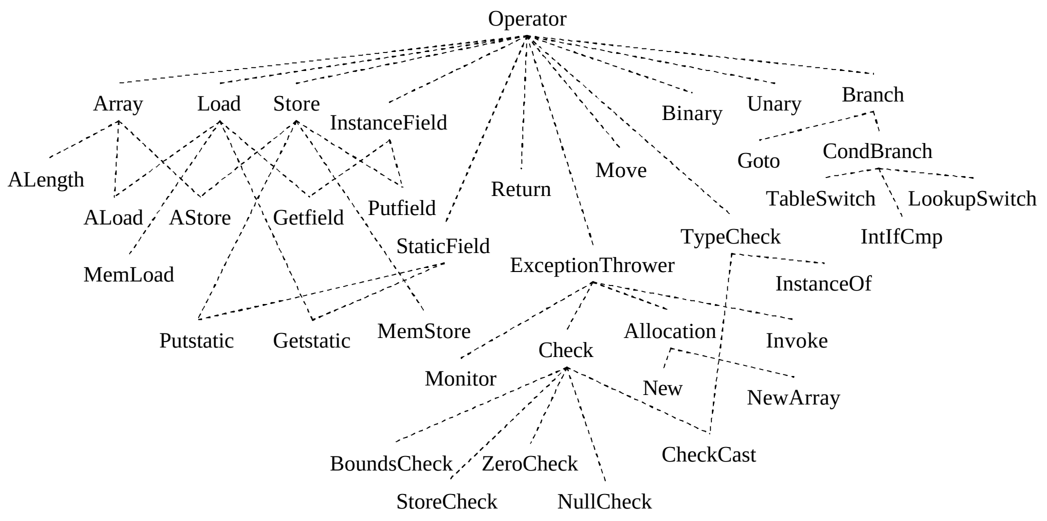

Figure 2. Overview of the operator hierarchy. Visitors visit in most-specific to least-specific order.

The compiler makes very extensive use of the visitor design pattern[7, 17]. The visitor pattern allows the traversal order to be effectively separated from the operations performed on each element. This makes writing compiler passes very easy — all compiler passes are implemented as visitors over various structures in the IR. Operators form a hierarchy as shown in Figure 2. Visitors visit in most-specific to least-specific order. For example, when encountering a `NullCheck` instruction, the first method that will be called is `visitNullCheck`, followed by `visitCheck`, `visitExceptionThrower`, and finally the generic `visitQuad` method.

Figure 3 gives the code for a simple side-effect analysis. This analysis keeps track of which static and instance fields can be touched by a method, along with the set of invocations in that method. To use the analysis, we simply instantiate an analysis object and pass it to the default visit method. The default visit method traverses the control flow graph and visits each instruction, calling the appropriate visitor methods on the supplied visitor object.

```
public class SideEffectAnalysis
    extends QuadVisitor.EmptyVisitor {
  Set touchedStaticFields = new HashSet();
  Set touchedInstanceFields = new HashSet();
  Set methodCalls = new HashSet();
  void visitInstanceField(CodeElement q) {
    touchedInstanceFields.add(InstanceField.getField(q));
  }
  void visitStaticField(CodeElement q) {
    touchedStaticFields.add(StaticField.getField(q));
  }
  void visitInvoke(CodeElement q) {
    methodCalls.add(q);
  }
}
  ...
  a = new SideEffectAnalysis();
  cfg.visit(a);
  ...
```

Figure 3. Example code for a simple side-effect analysis.

4.4 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). The API for the `Dataflow.Problem` class is contained in Figure 4.

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.

Fact *apply*(TransferFunction, Fact)
  Returns the result of applying a transfer function to a given dataflow fact.

boolean *compare*(Fact, Fact)
  Returns true if two dataflow facts are equal, false otherwise.

Fact *meet*(Fact, Fact)
  Returns the result of meeting two dataflow facts.

TransferFunction *compose*(TransferFunction, TransferFunction)
  Returns the composition of two transfer functions.

TransferFunction *closure*(TransferFunction)
  Returns the transfer function that is the Kleene closure of the given transfer function.

Figure 4. Overview of the API for the `Dataflow.Problem` class.

Joeq includes three standard solvers. The first is an iterative solver that iterates over the basic blocks in a given order (typically reverse post-order) until the dataflow values converge. The second is a worklist solver that utilizes a worklist of basic blocks that need to be (re-)computed. The worklist is implemented with a priority queue, where the priority of a basic block is its reverse-post-order number. The third is a strongly-connected component solver that solves the dataflow by finding and collapsing strongly-connected components. Only the last solver uses the *compose* and *closure* methods on transfer functions.[3]

The simplicity and completeness of the standardized dataflow framework makes the implementation of new analyses and optimizations very quick and easy. We have already implemented many of the standard dataflow analyses and optimizations. Work on more optimizations and analyses is ongoing.

4.5 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.

The interprocedural analysis framework is similar to the intraprocedural dataflow framework, but in addition it supplies calling context information to the analysis. The default solver begins with an initial call graph, which can be obtained via class hierarchy analysis[14], rapid type analysis[6], or whole-program pointer analysis[33]. The solver breaks the call graph into strongly-connected components, and then performs a bottom-up traversal, iterating around the strongly-connected components until they converge and generating a summary for each unique entry into a strongly-connected component. The analysis framework also includes support for discovering the call graph on the fly. We have used the interprocedural analysis framework to implement various context-sensitive and context-insensitive, whole-program and partial-program pointer analyses[33, 35]. The framework is efficient, with whole-program analysis times that are competitive to that of a C implementation[20].

5 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 back-end also has generalized support for atomically patching code fragments in a thread-safe manner without having to perform synchronization operations[21].

The code from the back-end can be output in three formats. The first is to put the code into memory for immediate execution. The second and third are to output the code in standard ELF or COFF object file formats respectively. The object files include the code relocations contained in the metadata. These object files can be linked into standalone executables.

6 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. Direct interpretation can only be used when Joeq is running natively. In reflective interpretation, the interpreter keeps a separate stack frame and performs all of the operations through reflection. Reflective interpretation can be used even when Joeq is running on a host virtual machine. The reflective interpreter also includes support for executing methods that cannot be interpreted in hosted execution mode, such as native methods, via reflective invocation.

7 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.

Specific allocation strategies are handled through the `Heap` interface. Conceptually, a `Heap` represents a bundle of memory. Different allocation strategies are implemented as subclasses of the abstract `Heap` interface. For example, a `FreeListHeap` allocates memory using a free list allocator, a `MallocHeap` allocates memory through a system call to `malloc()`, and a `FixedSizeBinHeap` allocates memory using fixed-size bins. Multiple heaps can be in use at the same time to handle allocations of different types. Entire heaps can be thrown away at once when they are explicitly deallocated or the collector determines that there are no more live refer-

[3]The `Dataflow.Problem` class contains default implementations of the *compose* and *closure* methods that perform the general operations. However, some dataflow analyses can be made more efficient by overriding the default implementations.

ences into a heap.

Joeq supports the simultaneous use of both managed and unmanaged memory. Managed memory is used when the code contains explicit deallocations, such as calls to `free` or `delete`, or uses a region-based deallocation scheme. Unmanaged memory is used when the code does not contain deallocations and instead relies on a garbage collector to discover unused memory and reclaim it. Joeq supports a mixture of these techniques. For example, user-level Java code uses unmanaged memory at the same time that native method implementations use `malloc` and `free` and the Just-In-Time compiler uses a region-based allocation scheme. When searching for live references, garbage collectors must still trace through managed and unmanaged memory to find live references to unmanaged storage, but it reclaims storage only in the unmanaged regions.

Raw addresses in Joeq are represented by special types: `HeapAddress`, `CodeAddress`, and `StackAddress` refer to addresses that refer to locations on the heap, in the code, or on the stack, respectively. Operations on addresses are implemented as method calls on these types; for example, the `peek()` method dereferences the given address and returns its contents. Abstracting these types in the Java implementation provides a unified interface to memory, which makes it possible to reuse much of the code for both native execution and hosted execution. In native execution, operations on addresses are directly compiled down to their machine-level equivalents. In hosted execution, the different address types are implemented as subclasses to the given address types, and proxy objects are created for addresses in the system. Operations on these proxy objects are reflected back into the system. During bootstrapping, the types on the proxy objects allow the bootstrapper to know the correct relocation to emit for various addresses. Using well-typed addresses also enforces a limited notion of type safety, even when dealing with raw addresses. In addition, it makes the implementation of Joeq independent of the address width, so Joeq could very easily be ported to a 64 bit architecture.

The object layout is also parameterized by an `ObjectLayout` interface. This interface includes methods to initialize objects and extract various types of metadata such as object class or lock status. Experimenting with various object layout schemes is as easy as subclassing `ObjectLayout` and implementing a few essential methods.

The framework to support advanced garbage collection is in place, but the implementation of the more advanced garbage collection algorithms is still ongoing. We are attempting to leverage the implementations of the garbage collectors contained in the JMTk toolkit.

8 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 time-consuming 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.

Joeq also includes an instrumentation-based profiler, which interfaces with the compiler. This profiler operates by inserting instrumentation code into the compiled code; every time the code executes, the instrumentation can record an event. This provides more precise information than the sampling profiler at the expense of generally higher profile overhead and more difficult control. Instrumentation code can be disabled through the use of code patching.

Data from both profilers can be output into files, which can be loaded on subsequent runs or by the static compiler. The profile information is also used by various compiler optimizations to improve their effectiveness, for example, inlining frequently-executed call sites or moving computation off of the common paths[31].

Joeq also includes interfaces for an online compilation controller to control dynamic recompilation based on profile information. The controller implementation is still work-in-progress.

9 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.

Joeq includes a generalized stack walking interface to walk a thread's stack to generate stack traces, deliver exceptions, profile the application, enforce security, or collect references for garbage collection. This stack walking interface works both when executing machine code under native execution and when interpreting with the interpreter under either native or hosted mode.

Joeq implements a fast subtype checking algorithm with positive and negative polymorphic caches[12]. The subtype test typically takes only three instructions and eleven words of storage per class. This subtype checking algorithm is used both at execution time to perform runtime type checks, as well as by the compiler passes that perform type analysis.

Joeq includes a complete M:N thread scheduler implementation written in Java. An M:N thread scheduler schedules M user-level threads across N native level threads. The number of native level threads, N, correspond roughly to the number of CPUs. The scheduler supports work stealing and thread migration, synchronization, wait and notify queues, suspending/resuming, single-stepping, and profiling. It supports three models of thread switching: fully preemptive where threads can switch at any time, semi-preemptive where thread switching can be disabled in critical regions, and cooperative where thread switching can only occur at specific locations.

Joeq uses Onodera's efficient bimodal field locking algorithm to implement Java monitors[23], a modified version of Bacon's thin lock algorithm[5]. We also implemented Gagnon's extension to avoid object instance overhead[16]. The locking algorithm is integrated with the thread scheduler, to avoid busy-waiting on locks and to hand off execution to the appropriate waiting threads when using the Java `wait()` and `notify()` features.

The Joeq runtime interfaces with the underlying system through shared library calls. Classes that make use of external calls access them through a general `ExternalLink` interface, which holds the library and symbol name. Libraries are loaded and symbols are resolved only as they are needed.

Internally, Joeq stores strings in UTF-8 format rather than Unicode[36]. All UTF-8 strings are immutable and unique; for any given string, there is exactly one `Utf8` object that corresponds to that string. UTF-8 is a more efficient representation than Unicode for the mostly-ASCII strings used in Joeq. The `intern()` functionality in `java.lang.String` (whereby there is guaranteed to be only one instance of a given string in the system) is also implemented using the uniqueness inherent in the UTF-8 implementation.

In addition to supporting the Java Native Interface (JNI) for calling native methods, Joeq also supports loading and compilation of the native code through the front-end. JNI methods that are implemented in C or C++ can be loaded through SUIF files. Even if only binaries are available, the ELF file loader and disassembler can sometimes load and disassemble the native method implementations. This allows native calls to be analyzed and inlined into call sites, among other things.

Joeq also includes a mechanism to support the implementation of native methods in Java. Joeq contains a special “mirror” package, and every time a class is loaded, the class loader looks for a class with the same name in the “mirror” package. If it exists, the mirror class is also loaded and its contents are added to the original class; method and field definitions in the mirror class replace methods and fields in the original class that have matching names and descriptors. Thus, this mechanism allows one to append or replace code to classes without touching the original implementation. Joeq provides implementations of the core of the Java class library using this technique: during the bootstrapping phase, it hijacks the class library of the host Java virtual machine and injects its own implementations of key methods using a mirror package that corresponds to the class library version on the host.

10 Related Work

Joeq has some similarities to another virtual machine written in Java, called Jalapeño[1, 9]. Before Joeq, the author of this paper worked on Jalapeño, and some of the ideas from Jalapeño were reimplemented in Joeq. In particular, the bootstrapping technique and the compiler and garbage collection infrastructures were heavily influenced by the designs in Jalapeño.

However, there is a difference in focus between the two systems, which shows up in the design. Jalapeño is heavily geared towards being a virtual machine for server-side Java, and many of the design decisions reflect that philosophy. For example, Jalapeño completely forgoes an interpreter and takes a compile-only approach. The runtime, data structures and IR are fairly Java-specific. Because all virtual machine data structures, including code, are treated as objects, all code must be compiled as relocatable. As a result, Jalapeño avoids storing absolute memory references. Jalapeño makes no use of the visitor design pattern; it relies on switch statements instead. There is limited support for Jalapeño as a static analysis engine or compiler. Joeq, on the other hand, was designed from the start to be language-independent and to include significant support for static analysis and compilation. It includes support for analyzing C/C++ and even binary object files. It includes a static compiler and a significant interprocedural analysis framework.

The design of the compiler infrastructure drew from two intermediate representations that the author of this paper has extensive experience with: the Jalapeño optimizing compiler IR[29] and the MIT Flex compiler IR[3]. We tried to extract the good ideas from these systems while leaving out difficult, ineffectual, or counterintuitive pieces. Like Joeq, both Jalapeño and Flex use an explicitly-typed, pseudo-register based representation with high-level and low-level operations. They both support the same notion of factored control flow as our Quad format. What is more interesting than the similarities are the differences: In Jalapeño, the IR is simply a list of instructions and the CFG is separate. We decided to make the CFG part of the core IR in Joeq because almost every use of the IR requires control flow information, and maintaining both the ordering of the instructions in the list and of the control flow graph made control-flow transformations more difficult than they needed to be. Flex uses the notion of code factories to generate code and perform compiler passes. We dropped this idea in Joeq in favor of using the visitor pattern, which (to us) is simpler, easier to understand, and easier to program correctly. Flex also includes a pointer in every instruction back to its context. Although this can be useful, we found it to be too space-consuming to justify.

The Ovm virtual machine is a set of tools and components for building language runtimes[24]. Like Joeq, Ovm makes significant use of design patterns. Both systems use a type hierarchy to classify instructions into groups with common behavior, and use this classification with a visitor pattern. To implement the visitor pattern, Ovm uses runabouts, in which visit methods are found by reflection and invoked by dynamically-generated helper classes[18]. The motivation behind using runabouts rather than visitors in Ovm was to avoid two problems in the straightforward implementation of the hierarchical visitor pattern: abstract or empty visit methods and redirecting to other visit methods in the hierarchy. Joeq avoids these problems by providing default empty visitor classes and by putting the traversal of the hierarchy in the `accept()` methods, rather than the `visit()` methods. This has the added benefit of eliminating the dependency on user code to explicitly call the visit method of the superclass, a common source of subtle errors.

Ovm uses a high-level stack-based intermediate representation, OvmIR, that is very similar to the bytecode IR in Joeq. Ovm does not include a register-based IR comparable to Joeq’s Quad format, but the design of OvmIR seems flexible enough to be able to be extended to support a register-based representation. Like the Joeq IR, the instructions are self-describing and can be inspected introspectively by the compiler.

There are some other virtual machines written in Java. JavaInJava is an implementation of a Java virtual machine written entirely in Java[26]. Rivet is an extensible tool platform for debugging and testing written in Java that is structured as a Java virtual machine[10]. Both JavaInJava and Rivet run on a host Java virtual machine using a technique similar to hosted execution in the Joeq virtual machine.

Marmot is an optimizing compiler infrastructure for Java that is written almost entirely in Java[15]. It includes an optimizing native-code compiler, runtime system, and libraries for a large subset of Java. The compiler implements many standard scalar optimizations, along with a few object-oriented optimizations. It uses a multi-level IR and a strongly-typed SSA form. Marmot supports several garbage collectors written in C++. It is difficult to evaluate the design of Marmot because the source code is not available. Intel’s virtual machine is written in C++, and it also uses a typed intermediate language[4]. One feature in common with Joeq is that it supports garbage collection at every instruction.

SableVM is a portable Java virtual machine written in C[16]. Its goals are to be small, fast, and efficient, as well as provide a platform for conducting research. It implements many interesting techniques such as bidirectional object layouts, a threaded interpreter and efficient locking. Soot is a framework for analyzing and optimizing Java[28]. It includes three intermediate representations — Baf, a streamlined bytecode representation, Jimple, a typed 3-address representation, and Grimp, a version of Jimple with aggregated high-level information. The first two representations are similar to our bytecode and Quad IR's, respectively. In the future, we are planning to extend our IR to include high-level information, a la Grimp.

The Microsoft .NET framework has similar goals of supporting multiple languages. The Common Language Infrastructure platform is a development platform that includes a virtual machine specification (named Virtual Execution System, or VES) that has a full-featured runtime environment that includes garbage collection, threading, and a comprehensive class library. It also includes a general language specification (named Common Language Specification, or CLS) that compiler writers can output to if they want to generate classes and code that can interoperate with other programming languages. The intermediate representation that they use is called Common Intermediate Language, or CIL. There are frontends that output CIL from a huge number of languages: Managed C++, Java Script, Eiffel, Component Pascal, APL, Cobol, Oberon, Perl, Python, Scheme, Smalltalk, Standard ML, Haskell, Mercury and Oberon. In the future, we plan to add a CIL loader to Joeq so that we can leverage the various frontends and other work on CIL. The upcoming Microsoft Phoenix project also has similar goals of using a single framework for static and dynamic compilation of safe and unsafe languages with support for both manual and automatic storage.

11 Conclusion

In this paper, we described the basic design and components of the Joeq system. 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.

While implementing the system, we tried to stick to the design principles of minimizing programmer burden, maintaining modularity and maximizing code reuse. We think that we succeeded — we have found it easy to extend the system, try out new ideas and implement new functionality. The entire system is approximately 100,000 lines of Java code, which is rather small when you consider its functionality. We believe that we were able to keep the size of the implementation small by factoring out common code and taking advantage of object-oriented features and convenient software patterns like visitors.

Although the design and interfaces are basically complete, many components are implemented with only the most basic functionality. There are still many improvements that could be made on the implementation side; for example, the code generation is not very intelligent, optimizations are limited, there is no implementation of the dynamic compilation controller, we do not yet have an advanced garbage collector, etc. Much of what is implemented is lacking documentation, a very important piece for the system to be useful to researchers. Over time, these gaps in implementation and documentation will disappear. We would also like to investigate extensions to the system, such as a CIL front-end and back-ends for more architectures.

The Joeq system is available as open source at `http://joeq.sourceforge.net`.

Acknowledgements

I would like to acknowledge the other contributors to the core Joeq infrastructure: Monica Lam, Michael Martin, Chrislain Razafimahefa, Laudney Ren, and Daniel Wright. I would like to thank Monica Lam and the anonymous reviewers for helpful comments on this paper, and Miho Kurano for proofreading assistance. I would also like to thank the students of the CS243 Winter 2002 class at Stanford for their feedback on the Joeq compiler infrastructure. This research was supported in part by NSF award #0086160 and an NSF Graduate Fellowship.

12 References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, T. Ngo, M. F. Mergen, J. C. Shepherd, and S. Smith. Implementing Jalapeno in Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 314–324, 1999.

[2] S. Amarasinghe, J. Anderson, M. Lam, and A. Lim. An Overview of a Compiler for Scalable Parallel Machines. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing (LCPC'93)*, volume 768 of *Lecture Notes in Computer Science*, pages 253–272, Portland, Oregon, 1994. Springer-Verlag, Berlin, Germany.

[3] C. S. Ananian. FLEX compiler infrastructure. http://www.flex-compiler.lcs.mit.edu, 2001.

[4] J. M. S. anf Guei-Yuan Lueh and M. Cierniak. Support for garbage collection at every instruction in a Java compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'99)*, ACM SIGPLAN Notices, pages 118–127, Atlanta, May 1999. ACM Press.

[5] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'98)*, pages 258–268. ACM Press, 1998.

[6] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96)*, pages 324–341, Oct. 1996.

[7] G. Baumgartner, K. Laufer, and V. F. Russo. On the interaction of object-oriented design patterns and programming languages. Technical Report CSD-TR-96-020, Purdue University, 1998.

[8] C. Bryce, C. Razafimahefa, and M. Pawlak. Lana: An approach to programming autonomous systems. In *Proceedings*

of the 16th European Conference on Object-Oriented Programming (ECOOP'02), Malaga, Spain, June 2002.

[9] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno dynamic optimizing compiler for Java. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, CA, United States, June 1999. ACM.

[10] J. Chapin. The Rivet virtual machine. http://sdg.lcs.mit.edu/rivet.html, 1999.

[11] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Workshop on Program Analysis For Software Tools and Engineering (PASTE'99)*, pages 21–31, 1999.

[12] C. Click and J. Rose. Fast subtype checking in the HotSpot JVM. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande*, pages 96–107. ACM Press, 2002.

[13] M. Dahm. Byte code engineering with the BCEL API. Technical Report 8-17-98, Freie Universit at Berlin, Institut fur Informatik, Apr. 2001.

[14] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, 1995.

[15] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: an optimizing compiler for Java. *Software: Practice and Experience*, 30(3):199–232, Mar. 2000.

[16] E. M. Gagnon and L. J. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01)*, pages 27–40, Berkeley, CA, Apr. 2001. USENIX.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse in object-oriented designs. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP'93)*, Berlin, 1993. Springer-Verlag.

[18] C. Grothoff. Walkabout revisited: The runabout. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP'03)*, Berlin, Germany, 2003. Springer-Verlag.

[19] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering (ICSE'02)*, May 2002.

[20] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 146–161, 2001.

[21] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a Java just-in-time compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, pages 294–310. ACM Press, 2000.

[22] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[23] T. Onodera and K. Kawachiya. A study of locking objects with bimodal fields. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 223–237. ACM Press, 1999.

[24] K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek. Engineering a customizable intermediate representation. In *Proceedings of the Workshop on Interpreters, Virtual Machines, and Emulators (IVME'03)*, San Diego, California, 2003.

[25] Sun Microsystems. *Java Native Method Invocation Specification*, 1997. v1.1.

[26] A. Taivalsaari. Implementing a Java virtual machine in the Java programming language. Technical Report SMLI TR-98-64, Sun Microsystems, Mar. 1998.

[27] Tool Interface Standard (TIS) Committee. *Executable and Linking Format (ELF) Specification*, 1995. v1.2.

[28] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[29] J. Whaley. Dynamic optimization through the use of automatic runtime specialization. Master's thesis, Massachusetts Institute of Technology, May 1999.

[30] J. Whaley. A portable sampling-based profiler for Java virtual machines. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 78–87. ACM Press, June 2000.

[31] J. Whaley. Partial method compilation using dynamic profile information. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*, pages 166–179. ACM Press, Oct. 2001.

[32] 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.

[33] J. Whaley and M. S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the 9th International Static Analysis Symposium (SAS'02)*, pages 180–195, Sept. 2002.

[34] 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.

[35] 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.

[36] F. Yergeau. RFC 2279: UTF-8, a transformation format of ISO 10646, Jan. 1998.