ACM SIGPLAN
2003 Workshop on

Interpreters, Virtual Machines and Emulators (IVME '03)

Proceedings

The Association for Computing Machinery, Inc.
1515 Broadway
New York, New York 10036

ACM ISBN: 1-58113-655-2

Additional copies may be ordered prepaid from:

ACM Order Department	Phone: 1-800-342-6626
P.O. BOX 11405	(U.S.A. and Canada)
Church Street Station	+1-212-626-0500
New York, NY 10286-1405	(All other countries)
	Fax: +1-212-944-1318
	E-mail: acmhelp@acm.org

ACM Order Number:

Printed in the U.S.A.

Preface

This is the first workshop on Interpreters, Virtual Machines, and Emulators (IVME). It provides a venue to bring together researchers and practitioners in these areas, to exchange their experiences and present new ideas.

Forty-one papers were submitted to the workshop. Program committee members were invited to submit papers, and seven of the submitted papers came from PC members.

For each paper three reviews were solicited from program committee members (a PC member could use secondary reviewers, but had to know the value of each reviewed paper relative to the others). Were the recommendations of different PC members conflicted, they were asked to resolve the conflict, and additional reviews were solicited. This process resulted in 131 reviews, for an average of 3.2 reviews per paper.

Finally, there were two email-based PC meetings: The full PC selected papers among the 34 papers that were not authored by PC members. A subcommittee consisting of those PC members who had not submitted a paper selected among the seven papers (co-)authored by PC members. Eventually we accepted eight papers (19.5% acceptance rate), two of them by PC members.

We are grateful to Rich Gerber for providing the START conference management system, to Ulrich Hirnschrott and Herbert Pohlai for setting it up and helping with paper processing, and to Michael Franz for arranging the printing of the preprints.

David Gregg, General Chair — M. Anton Ertl, Program Chair

Organizers

General Chair

David Gregg, Trinity College Dublin

Program Committee

Todd Austin, University of Michigan
M. Anton Ertl, TU Wien (chair)
Michael Franz, University of California at Irvine
Chris Fraser, Microsoft Research
Etienne Gagnon, University of Quebec at Montreal
David Gregg, Trinity College Dublin
Jan Hoogerbrugge, Philips Research
Roberto Ierusalimschy, PUC-Rio
Paul Klint, University of Amsterdam
Ian Piumarta, University of Paris 6
Vitor Santos Costa, Universidade Federal do Rio de Janeiro
Mario Wolczko, Sun Microsystems

Additional Reviewers

Lex Augusteijn	Ricardo Lopes
Kevin Casey	Ulrich Neumerkel
Manuel Eduardo Correia	Bernd Paysan
Roxana Diaconescu	Mark Probst
Norbert Esser	Franz Puntigam
Paul Gorissen	Ricardo Rocha
Paul Hoogendijk	Christian Stork
Andreas Krall	Efe Yardimci

Contents

Embedding an Interpreted Language Using Higher-Order Functions and Types

Norman Ramsey
Division of Engineering and Applied Sciences
Harvard University

Abstract Using an embedded, interpreted language to control a complicated application can have significant software-engineering benefits. But existing interpreters are designed for embedding into C code. To embed an interpreter into a different language requires a suitable API. Lua-ML is a new API that uses higher-order functions and types to simplify the use of an embedded interpreter. A typical application-program function can be added to a Lua-ML interpreter simply by describing the function's type.

1 Introduction

Suppose you have an application written in a statically typed, compiled language such as C, C++, or ML. If the application is complicated, like a web server or an optimizing compiler, it will have lots of potential configurations and behaviors. How are you to control it? If you use command-line arguments, you may find yourself writing an interpreter for an increasingly complicated language of command-line arguments. If you use a configuration file, you will find yourself defining, parsing, and interpreting its syntax.

A better idea, which we owe to Ousterhout (1990), is to create a *reusable* language designed just for configuring and controlling application programs, i.e., for *scripting*. Making a scripting language reusable means making it easy to *embed* its interpreter into an application. An application that uses an embedded interpreter is written in two languages. Most code is written in the original, *host* language (e.g., C, C++, or ML). But key parts can be written in the embedded language. This organization has several benefits:

- Complex command-line arguments aren't needed; the embedded language can be used on the command line.
- A configuration file can be replaced by a program in the embedded language.
- It is easy to write an interactive loop that uses the embedded language to control the application.
- The application programmer need not implement lexing, parsing, or evaluation; they come from the embedded interpreter.

To gain these benefits, the major effort required is the effort of writing the *glue code* that grants control of the host application to the embedded language.

The benefits above were first demonstrated by Tcl (Ousterhout 1990), which was followed by embedded implementations of other languages, including Python, Perl, and several forms of Scheme (Laumann and Bormann 1994; Benson 1994; van Rossum 2002; Jenness and Cozens 2002), as well as by another language designed expressly for embedding: Lua (Ierusalimschy, de Figueiredo, and Celes 1996a). But to use any of these embedded languages, you have to write your application in C (or C++). If you prefer a statically typed, functional language like ML, this paper explains how you too can reap the benefits of embedded languages.

To create an embedded language, you must design not only the language itself but also an interface that allows host-language application code to be scripted from within the embedded language. This interface—the *embedding API*—is the subject of this paper, which presents Lua-ML, a new API for embedding. Lua-ML provides two significant benefits:

- Type safety is guaranteed: it is impossible for an error in glue code to lead to an unexplained core dump.
- In almost all cases, glue code for a function is replaced by a simple description of the function's type—and this description is checked for correctness at compile time. Applications therefore require significantly less glue code than similar applications written in C.

Lua-ML is supported by two technical contributions: an adaptation of Danvy's (1996) type-indexed functions for partial evaluation (Section 3.1), which makes it easy to embed host-language functions (Section 3.2); and a programming convention that enables host-language functions to inspect or modify the state of an embedded interpreter (Section 3.3).

To focus attention clearly on the API, Lua-ML does not introduce a new host or embedded language; it uses existing languages. As an embedded language, I have chosen Lua, which is clean, flexible, efficient, and easy to implement. Lua enjoys a modest but growing following; its most visible users may be the developers of such popular games as *Grim Fandango*, *Baldur's Gate*, and *Escape from Monkey Island* (Ierusalimschy, de Figueiredo, and Celes 2001). Although convenient, Lua is not essential; it could be replaced by Tcl, Perl, or some other dynamically typed language.

As a host language, I have chosen Objective Caml (Leroy et al. 2001), a popular dialect of ML. Objective Caml provides algebraic data types, programming by pattern matching, higher-order functions, Hindley-Milner type inference, a sophisticated system of parameterized modules, and an object system that is compatible with type inference. Lua-ML uses higher-order functions and types in essential ways, but Objective Caml could be replaced by Standard ML or some other higher-order, typed language. (Preliminary experiments with Haskell show that a similar API should be possible for Haskell. Haskell's type classes reduce glue code even further, but the API is complicated by the need to use monads to describe Lua functions.)

IVME'03, June 12, 2003, San Diego, California, USA.

2 Lua-ML summarized and compared

The distinctive benefits of Lua-ML come from the part of the API used to integrate application-specific code into the embedded interpreter. Because this part would be hard to understand in isolation, especially for readers who have no experience using a similar API, I summarize the full API here. The summary looks at the Lua language, at the API's high-level view of an interpreter, at the transfer of values between host code and embedded code, and at the integration of application-specific code and data into the interpreter.

Because it is difficult to evaluate the merits of an API in isolation, I include comparisons with three established APIs: Tcl 7.3, Lua 2.5, and Lua 4.0. These comparisons point out not only the distinctive aspects of Lua-ML but also those aspects that are similar to related APIs.

2.1 What the language looks like

An API can be hard to understand unless you know a little about the language behind it. This section highlights the salient features of Lua and Tcl.

Lua 2.5 and 4.0 Lua-ML implements the Lua language version 2.5, which is described by Ierusalimschy, de Figueiredo, and Celes (1996b). Version 2.5 is relatively old, but it is mature and efficient, and it omits some complexities of later versions. The current version as of early 2003 is Lua 4.0; I mention differences where appropriate.

Lua is a dynamically typed language with six types: nil, string, number, function, table, and userdata. Nil is a singleton type containing only the value nil. A table is a mutable hash table in which any value except nil may be used as a key. Userdata is a catchall type, which enables an application program to add new types to the interpreter. Except for table, the built-in types are immutable; userdata is mutable at the application's discretion.

Lua has three significant syntactic categories: expression; statement; and top-level *chunk*, which may be a statement or a function definition. Functions may be defined only at top level; Lua 2.5 has first-class, non-nested functions. Versions 4.0 and 5.0 (beta) offer nested functions, and each version has a different mechanism for giving a nested function access to parameters and variables of enclosing functions.

The Lua language has two unusual features. First, a Lua function may accept a variable number of parameters and return a variable number of results. Moreover, the number of actual parameters in a call need not match the number of formal parameters a function expects. If there is a mismatch, the parameters are *adjusted*: if a function receives more actual parameters than it expects, the extra actual parameters are dropped, and if a function receives fewer actual parameters than it expects, extra formal parameters are set to nil. A similar adjustment is applied to results.

Lua's other unusual feature is *fallbacks*, which provide a dynamic exception mechanism similar to a PL/I on-unit. A fallback is a Lua function that is called when a built-in operation fails, e.g., when Lua code tries to retrieve an index not present in a table. Fallbacks can be used to implement error handling, overloading, inheritance and other features. In Lua 4.0, fallbacks are replaced by a similar but more powerful mechanism called *tag methods*. Further details are not relevant to this paper.

Tcl 7.3 Tcl 7.3 is described by Ousterhout (1994). Tcl has no type system or type checking: the only values are immutable strings and named hash tables, and a hash table is represented as a string containing the table's name. Whether a string stands for itself or the name of a table depends on the context in which it appears.

Tcl has only two syntactic categories: expression and command. The command category includes what in another language might be declarations, statements, and function definitions. For example, the `proc` command introduces a new "procedure" command, which executes its body (a sequence of commands) in a fresh environment with private local variables. Tcl's concrete syntax has almost no grammatical structure: it says only that a command is a name applied to a list of strings. The meaning of each string is determined by the command, which decides if a string represents a value, a command or expression that should be evaluated, or a syntactic keyword. For example, the `if` command evaluates its condition as an expression, evaluates its true or false branch as a command, and treats `then` and `else` as keywords.

2.2 How an interpreter appears in the API

At a high level, Lua-ML looks much like other APIs. An application can create many interpreters, each of which has mutable state; it can manipulate the global variables and functions of an interpreter; and it can evaluate Lua code in the context of an interpreter.

An interpreter The state of a Lua-ML interpreter is represented by a value in the host language, Objective Caml. The state includes a table of global variables; a table of fallbacks; and a summary of the call stack, which is printed in the event of a fatal error. An interpreter is created in two stages: compile-time and run-time. At compile time, the application supplies a (possibly empty) set of libraries to an ML module called `MakeInterp`, which returns an interpreter module we will call `Interp`.[1] At run time, calling `Interp.mk` creates a fresh instance of the interpreter; the instance has type `Interp.state`, which we write simply as `state`.

Lua 4.0 and Tcl 7.3 treat the interpreter similarly: its state is represented by a value in the host language (of type `lua_State*` or `Tcl_Interp`, respectively). In Lua 4.0, the visible state of an interpreter includes a stack of values. This stack can be manipulated only through the API; its representation is not exposed. In Tcl 7.3, an interpreter `i` includes a field `i->result`, which is used to communicate results and error messages from commands. The Lua 2.5 API is unusual in that the interpreter and its state are implicit; interpreters cannot be created or destroyed, so an application contains exactly one interpreter.

Variables, functions, and commands In Lua-ML, application code can add or remove global Lua variables or change their values, all by manipulating the global-variable table in the interpreter's state. Functions are treated the same as variables: a Lua function is simply a variable that has a function value. The Lua 4.0 and Lua 2.5 APIs are similar. Lua 4.0 also includes the capability of bundling a function value with other values to form a kind of closure.

The Tcl 7.3 API is only slightly different. It provides one set of API functions for manipulating values, which are strings, and another set of API functions for manipulating commands. A command is represented as a C function bundled with a value of type `ClientData`, which may be a pointer to mutable state. This representation enables a Tcl command to simulate a closure or an object.

[1] The compile-time composition of libraries to form the interpreter module is beyond the scope of this paper; Ramsey (2003) presents details.

Evaluation In each of these APIs, the host application is ultimately in control: It decides what embedded code is evaluated when. Evaluation requires code and an environment, and each system makes the environment part of the state of the interpreter. All four APIs provide functions that evaluate sequences of definitions (or commands), which may be located in strings or files. Most APIs can evaluate definitions at top level only, but Tcl 7.3 makes it possible to evaluate a command either in the top-level environment or in the environment of the currently active function.

Only the Tcl API provides functions that evaluate expressions. Such functions are necessary because a string passed to a Tcl command may represent a keyword, a value, an expression, or another command. It is up to the implementation of the command to know which is which and to call the appropriate API function.

2.3 How values cross the interface

The Lua-ML, Lua, and Tcl APIs differ most greatly in their treatment of values. We consider how an embedded value appears to host code, how memory is managed, and how values are converted between host and embedded forms.

What an embedded value looks like In Lua-ML, a Lua value is represented by a value in the host language (Objective Caml). Such a value has type `value`, which is exposed to an application as follows:

```
type value
  = Nil
  | Number   of float
  | String   of string
  | Function of srcloc * funty
  | Userdata of userdata
  | Table    of table
and funty = value list -> value list
and table = (value, value) Luahash.t
```

This declaration defines `value` to be one of the six alternatives listed above, and it defines `funty` and `table` to be a function type and a hash table, respectively. The type `funty` doesn't mention `state`, which should surprise you, because a Lua function can inspect and modify the state of its interpreter. All is revealed below, in Section 3.3. Types `srcloc` and `userdata` are abstract types, the declarations of which are not shown. The type `srcloc` represents the source location at which a function is defined. The type `userdata` represents application-specific data declared in libraries.

`Luahash.t` is a type constructor exported by a hash-table abstraction that is part of the Lua-ML API. It can be used independently of a Lua interpreter.

In Lua 4.0, a Lua value cannot be represented as a value in the host language: every Lua value is hidden inside an interpreter. Such a value is either on the interpreter's stack or is accessible through an opaque *reference*. API functions that manipulate values do so by referring to stack positions. A reference can be created from a value on a stack and can later be used to push that same value back on the stack.

In Lua 2.5, a value may be on the stack or hidden behind a reference, but it may also be converted to a C value of type `lua_Object`. Some API functions use stack positions and others use arguments or results of type `lua_Object`.

In Tcl 7.3, a value is always represented as a string and has type `char *`. Such strings contain no embedded nulls.

Hash-table abstractions are included in Lua 4.0 and Lua 2.5, but these abstractions can be used only in conjunction with an interpreter, because the hash-table operations use the interpreter's stack. By contrast, Tcl 7.3, like Lua-ML, provides a hash-table abstraction that can be used independently of an interpreter.

Memory management In Lua-ML, both embedded values and host values are managed by the host's garbage collector; the API need not mention memory management. The other APIs don't have the luxury of a built-in garbage collector, so they have to deal with memory management. Both Lua 4.0 and Lua 2.5 provide a special-purpose garbage collector for Lua values only. Of the two, Lua 4.0 has the simpler API for collection, because every root is either on the interpreter's stack or is pointed to by a reference. Tcl does not use garbage collection: Because every value is a string and therefore contains no pointers, it suffices for the API to specify whether application code or the Tcl implementation is responsible for freeing each string's memory, and similarly for allocating.

Conversion between host and embedded values Lua-ML exposes the representation of a Lua value, so functions that convert between Lua values and ML values are not required. Such functions are, however, extremely convenient. Lua-ML provides type-specific conversion functions in pairs: `embed` and `project`. The `embed` function is the mapping from Caml into Lua, which always succeeds; `project` is the mapping from Lua to Caml, which can fail (and raise an exception) if the Lua value has the wrong type. For example, one might convert a Caml floating-point value to a Lua value by calling `float.embed`, or convert a Lua number to a Caml floating-point value by calling `float.project`. The main innovation in Lua-ML is that it provides *higher-order* functions that can create an *unlimited* supply of conversion functions. The details are the topic of Section 3.1.

Lua 4.0 also provides conversion functions, but there are only six pairs: one for each basic Lua type. Because values are abstract, the API also provides a test function for each basic Lua type. For example, one might use `lua_isnumber` to see if a Lua value is a number, `lua_getnumber` to convert a Lua number to a C floating-point value, and `lua_pushnumber` to convert a C floating-point value to a Lua value. The conversion and testing functions, like other functions in the Lua 4.0 API, refer to Lua values on the Lua stack. The conversion interface in Lua 2.5 is similar, except that only conversion from C to Lua uses the stack; the testing functions and the conversion functions from Lua to C accept arguments of type `lua_Object`.

In Tcl 7.3, the API provides conversion procedures for integers, floating-point numbers, Booleans, and lists. To convert from a Tcl value to a C value, one uses a type-specific procedure that writes its result through a pointer and returns a termination code, which indicates success or failure. For example, the API procedure `int Tcl_GetDouble(Tcl_Interp *i, char *s, double *p)` tries to convert the Tcl value `s` to a floating-point value, stores the result in `*p`, and returns a termination code. To convert from a C value to a Tcl value, an application uses standard C procedures such as `sprintf`; no API procedures are needed.

2.4 Application-specific code and data

An API should make it easy to embed application-specific code or data into an interpreter; we compare the APIs.

2.4.1 Embedding a function or command

The *glue code* used to make host functions available in the embedded interpreter represents most of the work of using an embedded language. For each API we consider, we explain how to embed a function, and we show example glue code used to embed the library function atan2.

Lua-ML In Lua-ML, an application programmer can define a new Lua function by writing an ML function that takes a list of values as arguments and returns a list of values as results. (Such a function's access to interpreter state is discussed in Section 3.3.) It is much more convenient, however, to define an ordinary ML function and to convert it to a Lua function by using the embed member of an embedding/projection pair. For example, my_atan2 is the Lua version of atan2.

```
let my_atan2 =     my_atan2 : value list -> value list
  let f = func (float **-> float **-> result float)
  in  f.embed atan2
```

On the second line, the expression func (float **-> float **-> result float) builds an embedding/projection pair for the ML type float -> float -> float, which takes two floating-point arguments and returns a single floating-point result. The Lua-ML operators func, **->, and result are explained in Section 3.2, and their types are shown in Table 2. The box at the top of the display gives the type of my_atan2.

The expression (func t).embed is so common that we provide an abbreviation efunc t. We could have written only efunc (float **-> float **-> result float) atan2. Thus, the only glue code needed is a description of atan2's type. This lack of glue code is typical of the great majority of embedded functions.

Lua 2.5 and 4.0 In Lua 2.5, an application programmer defines a new Lua function by writing a C function that takes no arguments and returns no results. This function gets its arguments from the Lua stack, and it returns its results, of which there may be more than one, by pushing them onto the stack (on top of the arguments). A function may indicate an error by calling the API function lua_error, which uses longjmp to achieve the effect of raising an exception.

Here is atan2 in Lua 2.5. A Lua function conventionally ignores extra arguments, so the number is not checked.

```
void my_atan2(void) {
  if (!lua_isnumber(lua_getparam(1)))
    lua_error("first arg not a number");
  if (!lua_isnumber(lua_getparam(2)))
    lua_error("second arg not a number");
  lua_pushnumber(
    atan2(lua_getnumber(lua_getparam(1)),
          lua_getnumber(lua_getparam(2))));
}
```

The example has a similar flavor in Lua 4.0, except that the state of the interpreter is passed explicitly throughout.

Tcl 7.3 In Tcl 7.3, an application programmer defines a new Tcl command by writing a C function that is passed an interpreter i and a list of arguments. Each argument is a string, and the list is represented using the C convention of argc and argv. An argument may represent a value or it may represent *syntax*, which may be evaluated to produce a value; the intepretation of each argument is up to the command procedure. A command procedure is also passed an argument of type ClientData, which makes it possible to write a command that simulates a method of an object or that simulates a closure.

When executed, a command procedure has a side effect on i->result and returns a termination code. The code may indicate successful termination, an error, a nonlocal exit such as break, or an application-specific termination condition. The termination code, like longjmp, provides a way to work around the lack of exceptions in C.

Here is atan2 in Tcl 7.3.

```
int my_atan2(ClientData d, Tcl_Interp *i,
             int argc, char *argv[])
{
  double x, y;
  if (argc != 3) {
    i->result = "wrong # of args";
    return TCL_ERROR;
  }
  if (Tcl_GetDouble(i, argv[1], &x) != TCL_OK)
    return TCL_ERROR;
  if (Tcl_GetDouble(i, argv[2], &y) != TCL_OK)
    return TCL_ERROR;
  sprintf(i->result, "%f", atan2(x,y));
  return TCL_OK;
}
```

The types of Tcl's conversion procedures enforce an assembly-language style of programming, in which each intermediate result must be named. The Lua 2.5 and 4.0 APIs allow more natural programming, because the result of calling a conversion procedure may be passed directly to another procedure. Lua-ML enables the most natural style: each conversion procedure is used declaratively to describe a function's type, and the ML code for the function itself need not use any conversion procedures.

2.4.2 Embedding application-specific data

Application-specific code often operates on data of application-specific types. Each of the APIs we consider uses a different mechanism to convert an application-specific host value to a value in the embedded language.

In Lua-ML, each application-specific type is declared in a library. The library supplies a definition of the type, a function used to print a value of the type, and a function used to compare two values for equality. Libraries are compiled separately and combined using the ML modules system (Ramsey 2003). The combined libraries define the userdata type used in the interpreter, and they provide an embedding/projection pair for each application-specific type. The design provides extensibility and separate compilation while preserving type safety; the details are beyond the scope of this paper.

In Lua 4.0 and 2.5, a value of application-specific type must be represented by a C value of type void * and by an accompanying *tag*, which is a small integer. The tag can be used to distinguish different application-specific types. A tag and pointer may be converted to a Lua value of type userdata, from which the same tag and pointer can be extracted. Type safety is ultimately left up to the programmer, but unsafe code can easily be isolated in an application-specific conversion routine. An example appears in Appendix A.

In Tcl 7.3, a value of application-specific type must be represented as a string. Tcl lacks the equivalent of Lua's tag: the API provides no help in distinguishing an application-specific string from any other string, and making sure such strings are unique and are used safely is entirely up to the application. An application programmer is advised to give every value a unique name, to keep a hash table in private state, and to use the hash table to map the name to the value (Ousterhout 1994, p. 283). Knowing when to use this hash table is up to the programmer.

3 Technical contributions of Lua-ML

Lua-ML's advantages stem from its handling of functions.

- A function can be embedded with almost no glue code because embedding can be extended to an unbounded number of types, including function types (Section 3.1).

- Objective Caml and Lua use different models of functions, and each language reacts differently to a function call in which arguments are missing. These differences are cleverly hidden by the embedding/projection pairs for function types (Section 3.2).
- Although almost 90% of embedded Caml functions have no need to inspect or modify the state of a Lua interpreter, some do. Lua-ML supports both kinds of functions without messing up the API (Section 3.3).

3.1 Embedding and projection

This section describes the implementation of embedding and projection functions. To represent an embedding/projection pair, we define type `('a, 'b) ep`: an `embed` function for converting a value of type `'a` into a value of type `'b` and a `project` function for the opposite conversion. For the special case where we are embedding into a Lua `value`, we define type `'a map`.

```
type ('a, 'b) ep =
  { embed : 'a -> 'b; project : 'b -> 'a }
type 'a map = ('a, value) ep
```

One example pair is `float`, which has type `float map` and is introduced in Section 2.3 above. The value `float.embed` is the function `(fun x -> Number x)`, which takes the Caml number `x` into the corresponding Lua value, which is built by applying the `Number` constructor to `x`.[2]

Defining `float.project`, which converts from a Lua value to a floating-point number, is more complicated, because in Lua, a string may be used where a floating-point value is expected, provided the string represents a floating-point number. The value `float.project` is the function

```
function
| Number x -> x
| String s when is_float_literal s ->
    float_of_string s
| v -> raise (Projection (v, "float"))
```

This function maps a Lua number to the same Caml number. It also maps a Lua string `s` to the floating-point number represented by `s`, *provided* that `s` satisfies `is_float_literal`, which checks to see that `s` is an appropriate string. If it gets any other kind of value, it raises the `Projection` exception, indicating that the value cannot be converted to a floating-point number. In Lua-ML, every dynamic type error raises `Projection`.

To provide a small set of conversion functions is not new. What Lua-ML adds is the ability to create pairs of conversion functions for *arbitrarily many* ML types. In other words, embedding and projection are a *type-indexed family* of functions. The idea is inspired by Danvy (1996), who uses a similar family to implement partial evaluation. Danvy (1998) credits Andrzej Filinski and Zhe Yang with originating this family, which has also been independently adapted by Benton (2003) for use in embedded interpreters.

We build a type-indexed family of functions as follows.

- For a base type, such as `float`, we provide a suitable embedding/projection pair. Lua-ML includes pairs for `float`, `int`, `bool`, `string`, `unit`, `userdata`, `table`, and `value`.
- For a unary type constructor, such as `list`, we provide a higher-order function that maps an embedding/projection pair to an embedding/projection pair. Lua-ML includes such functions for the `list` and `option` type constructors.
- For a type constructor of two or more arguments, we continue in a similar vein. Such constructors are rare, except for the arrow constructor, which describes a function type. The arrow needs careful treatment because Lua and Caml treat partial application differently.

[2] In Objective Caml, unlike in Haskell or Standard ML, a datatype constructor cannot be used as a function, so the η-expansion is necessary.

```
type 'a map =
  { embed   : 'a -> value ;
    project : value -> 'a }
float    : float    map
int      : int      map
bool     : bool     map
string   : string   map
userdata : userdata map
unit     : unit     map
value    : value    map
table    : table    map
list     : 'a map -> 'a list   map
option   : 'a map -> 'a option map
default  : 'a -> 'a map -> 'a map
```

Table 1: Constructors for embedding/projection pairs

Table 1 gives the types of these functions.

The implementations of these functions are more interesting than you might expect, because Caml and Lua are substantially different. For example, Lua lacks the `int`, `bool`, `list`, and `option` types, and Lua's showcase, the table type, is seldom used in Caml functions. Resolving such inconsistencies requires suitable programming conventions, and the conventions are embodied in embedding/projection pairs. This implementation strategy makes it easy to add new conventions and to use consistent conventions throughout a program.

One such convention is shown above: a string can represent a floating-point number. Here are some others:

- Any Lua value can be interpreted as a Boolean; nil represents falsehood, and every non-nil value represents truth. This convention is embodied by the `bool` pair, which has type `bool map`.

```
let bool =                              bool : bool map
  { embed   = (fun b -> if b then Number 1.0
                        else Nil);
    project = (fun v -> v <> Nil)
  }
```

- A number may be used where a string is expected.
- A list should be represented as a Lua table, where the elements of a list of length n are stored with keys $1, 2, \ldots, n$.

These conventions, code for which is shown in Appendix B, are part of the idiom of Lua 2.5 and 4.0. Some, like the Boolean and list conventions, have syntactic and semantic support in the Lua language. Another common convention is that a function may allow nil to stand for a default argument. We support this convention with the `default` function, which has type `'a -> 'a map -> 'a map`; the pair `default` v `t` behaves just like the pair `t`, except it projects nil to v.

For Lua-ML, we also invented new conventions. For example, ML has a built-in type constructor `option`. A value of type `'a option` may be `None`, which means the absence of any value, or it may be `Some x`, which means the value `x`, where `x` has type `'a`. In our convention, the Lua value nil stands for `None`, and any other value stands for `Some` of that value. (This convention fails if a value v of type `'a` is itself embedded in Lua as nil, since the convention projects nil as `None`, not as `Some` v.) To build an embedding/projection pair for type `'a option`, we need an embedding/projection pair for type `'a`, which here is called `t`:

```
option : 'a map -> 'a option map
```

```
let option t =
  { embed   = (function None   -> Nil
                      | Some x -> t.embed x);
    project = (function Nil -> None
                      | v   -> Some (t.project v))
  }
```

As shown in the box, `option` has type `'a map -> 'a option map`.

Another convention involves embedding and projection of polymorphic functions. For example, Objective Caml's list-reversal function, `List.rev`, has type `'a list -> 'a list`: It is a polymorphic function that can reverse a list containing any type of value. But Lua does not have parametric polymorphism, so what is the embedding/projection pair that corresponds to the type variable `'a`? It is the `value` pair, which embeds and projects using the identity function. The Lua function

```
efunc (list value **-> result (list value)) List.rev
```

reverses any Lua list, no matter what Lua values the list contains.

Most programming conventions are easily embodied in simple embedding/projection pairs such as are shown above. The big exception is the convention for functions.

3.2 Conventional uses of functions

In Objective Caml, a function of multiple arguments is conventionally defined in its Curried form, i.e., as a function that returns another function. For example, the library function `String.index` has type `string -> (char -> int)`. We normally write such a type as `string -> char -> int`, because the type arrow is right-associative. To apply such a function, we write `(String.index "hello") 'e'`, or because function application is left-associative, simply `String.index "hello" 'e'`. In Objective Caml, there is no real difference between a function that takes two arguments and a function that takes one argument and returns a new function. But in Lua, there is a big difference! The difference can be explained by considering what happens when a function is applied to only some of its arguments, i.e., when it is *partially applied*.

In Caml, a partially applied function, such as `String.index "hello"`, creates a closure, which represents a new function that is returned. This new function, when itself applied to an argument such as `'e'`, behaves as would `String.index` applied to the two arguments `"hello"` and `'e'`. In Lua, a partially applied function is *adjusted*, which means that any "missing" arguments are filled in with nils. In Lua, therefore, `String.index("hello", 'e')` is different from `String.index("hello")('e')`[3], which is equivalent to `String.index("hello", nil)('e')`. By convention, the first, unCurried form is expected.

When embedding a multi-argument Caml function into Lua, we have to convert it from Curried to unCurried form. (One can retain the Curried form by using the Lua-ML operator `-->`, which has type `'a map -> 'b map -> ('a -> 'b) map`, but we have never used this operator in a real program, and its implementation has no interesting features.) We convert a function by describing its type using the `**->` operator.[4] The `**->` operator combines the type of each argument with a *result type*. The result type is not an ordinary embedding/projection pair but a special value that has the abstract type `'a mapf` ("map to function"), where `'a` is the type of the value returned by the Caml function.

[3]This syntax is available in Lua 4.0, which provides first-class functions, but in Lua 2.5 such an expression is not even syntactically correct; it would have to be written `x = String.index("hello"); x('e')`.

[4]The operator's name begins with `**` because that is the only way to make a user-defined operator infix and right-associative.

```
type 'a mapf (* abstract type *)
**->   : 'a map  -> 'b mapf -> ('a -> 'b) mapf
result : 'a map  -> 'a mapf
func   : 'a mapf -> 'a map
```

Table 2: Embedding and projection for functions

Table 2 shows the types of the function-conversion operations. To convert a Caml function into unCurried, Lua form, we take the embedding/projection pair for the result, convert it to a result type using `result`, add the argument types with `**->`, and finally convert the result type back to an embedding/projection pair using `func`. For example, if we have a Caml function of type `t -> u -> v -> w`, we turn it into a Lua function of three arguments by using the embedding/projection pair produced by `func (t **-> u **-> v **-> result w)`.

The representation of type `'a mapf`, which is not exposed in the API, is an embedding/projection pair between `'a` and `value list -> value list` (thus the nickname "map to function"). The function-conversion operations that work with `mapf` are a bit tricky. The simplest is `func`: embedding adds `srcloc` and applies `Function`, while projection strips `Function` and ignores `srcloc`.

```
type 'a mapf = ('a, value list -> value list) ep
func : 'a mapf -> 'a map
```

```
let func (arrow : 'a mapf) : ('a map) =
 { embed = (fun (f : 'a) ->
            Function (caml_fun, arrow.embed f));
   project =
    (function
     | Function (_, f) -> (arrow.project f : 'a)
     | v -> raise (Projection (v, "function")))
 }
```

Value `caml_fun` of type `srcloc` identifies the function as an embedded function. A function translated from Lua source code would have a `srcloc` field indicating its source-code location.

The details of `**->` and `result` are a bit technical, but because the resulting embedding and projection functions are novel, they are worth presenting anyway. The `**->` operation converts between Curried Caml functions and unCurried Lua functions. It builds an embedding/projection pair inductively from `firstarg`, which is an embedding/projection pair for the first argument, and from `lastargs`, which is an embedding/projection pair for a function that takes one less argument. To build `firstarg **-> lastargs`, we need an embedding (`apply`) and a projection (`unapply`).

```
**->    : 'a map -> 'b mapf -> ('a -> 'b) mapf
apply   : ('a -> 'b) -> (value list -> value list)
unapply : (value list -> value list) -> ('a -> 'b)
```

```
let ( **-> ) (firstarg : 'a map) (lastargs : 'b mapf)
        : ( ('a -> 'b) mapf ) =
  let apply (f : 'a -> 'b) = fun actuals ->
    let v, vs = match actuals with []     -> Nil, []
                                 | h :: t -> h, t in
    let f_v = f (firstarg.project v) in
    lastargs.embed f_v vs
  in
  let unapply (f_lua : value list -> value list) =
    fun (v : 'a) ->
      lastargs.project
      (fun vs -> f_lua (firstarg.embed v :: vs))
  in
    { embed = apply; project = unapply }
```

The `apply` function takes a Caml function `f` of type `'a -> 'b` and converts it to a Lua function of type `value list -> value list`. This converted function takes its actual arguments `actuals`, puts the first argument in `v`, and puts any remaining arguments in `vs`. (This code also implements *adjustment*: if the list of arguments is empty, it is as if the first argument had been `Nil`.) Because the Caml function `f` is Curried, it can be partially applied to the first argument `v` to produce `f_v`, which has type `'b`. Function `f_v` is then converted to a Lua function (by `lastargs.embed`) and applied to the remaining arguments.

The projection function `unapply` takes a Lua function `f_lua` and converts it to a Caml function of type `'a -> 'b`. The Caml function takes its first argument `v` and must return a function of type `'b`. The Caml function therefore converts `v` to Lua using `firstarg.embed`, then builds a new, anonymous Lua function. This anonymous function takes the remaining arguments `vs` and applies `f_lua` to all the arguments. The anonymous Lua function is then converted to a Caml function of type `'b` by using `lastargs.project`.

The base case for the conversion of functions is a pair for a function that takes no arguments and returns a result of type `'a`. Such a pair is produced by `result`. In turn, `result` uses an embedding/projection pair `r` that converts between a result of type `'a` and a Lua value.

`result : 'a map -> 'a mapf`

```
let result (r : 'a map) : ('a mapf) =
 { embed   = (fun (v:'a) -> fun args -> [r.embed v]);
   project = (fun fl -> r.project (take1 (fl [])))
 }
```

To embed a result as a no-argument Lua function, we take the result `v` and produce a Lua function that ignores its arguments and returns the singleton list holding the result of embedding `v`. To project a Lua function `fl` as a result, we apply `fl` to the empty list of arguments, take the first element of the list of results, and project that Lua value into a Caml value. (If the list of results is empty, `take1` returns Lua's `Nil`, which is another example of adjustment.)

The three functions `func`, `**->`, and `result` cooperate to create a natural mapping between Caml functions and Lua functions. Using this mapping, Caml code defines and uses functions in Curried style, which is natural for Caml. Lua code defines and uses functions in unCurried style, which is natural for Lua. The only aspect that is slightly unnatural is having to apply `result` to the embedding/projection pair for each result type. Programmers soon learn to apply `result`, however, because if it is mistakenly omitted, Caml's type checker complains.

3.3 Functions and the interpreter's state

A Lua function can modify the state of its interpreter, e.g., by changing the value of a global variable. Such a function might naturally be expected to have the type `state -> value list -> value list`. But we actually use `value list -> value list`. How and why to do so is not obvious at first, and the explanation is one of the contributions of this paper.

We first present two designs that use the type `state -> value list -> value list`, and we show the flaws in each design. We then present our preferred design.

The incorrect but obviously attractive design Our first design used the function type `state -> value list -> value list`. As in our current design, the embedding/projection pairs built with `result` and `**->` were designed for pure Caml functions, i.e., those that do not manipulate the state of a Lua interpreter. We therefore needed a slightly different embedding function to be produced by `func`: instead of the function `arrow.embed f`, we used the function `fun s -> arrow.embed f`. Passing the pure Caml function `f` to `arrow.embed` converts `f` to a function of type `value list -> value list`, and wrapping this function in `fun s -> ...` yields a function of type `state -> value list -> value list`. To embed the rare impure Caml function, i.e., one that actually depends on a state, we provided an operation `impure_func` of type `'a mapf -> (state -> 'a) map`.

The sticky part of this design is projecting a function. We are given a function `fl'` of type `state -> value list -> value list`. To project `fl'` to a Caml function that does not expect a state, we need to apply `fl'` to some state, then project the result (which has type `value list -> value list`). We figured that if we were projecting a pure function, it wouldn't use its state, so we could apply `fl'` to an arbitrary state: for example, the empty state. This design worked for a surprisingly long time.

The problem we overlooked occurs in the higher-order case. Suppose we embed a function like `List.map`, which has type `('a -> 'b) -> 'a list -> 'b list`. `List.map` is pure: if we apply it to function `f` and list `l`, it returns a new list containing the results of applying `f` to each element of `l`. But when we *embed* `List.map`, we create a function that *projects* each of `List.map`'s arguments from the type `value` to the Caml type that `List.map` expects for that argument. And when we project a Lua function `fl'`, things go wrong: just because `List.map` is pure, there is no reason to expect its argument to be pure. We were bitten by this bug the first time we applied a higher-order Caml function to a function defined in Lua. The Lua function couldn't find any global variables; even the predefined function `print` was missing. The globals were missing because our projection had applied `fl'` to the empty interpreter state, which does not contain any global variables, not even `print`!

The obviously correct but unattractive design An obvious way to correct the projection problem is to pass the state to each projection function: with each embedding function of type `'a -> 'b`, we can pair a projection function of type `state -> 'b -> 'a`. But the `mapf` type becomes horrifying:

```
type 'a mapf = (* don't try this at home *)
 { embedf : 'a -> (state -> value list -> value list);
   projectf :
    state -> (state -> value list -> value list) -> 'a
 }
```

This design works and is correct, but the loss of symmetry is discouraging. Passing states around requires extra bookkeeping, which is doubly offensive because states are rarely used: in our largest application, we embed 138 functions into the interpreter, and only 17 of these make any use of `state`. (And we could cut this number in half by making a small change in the way errors are handled.) If we ignore Lua-library functions and consider only application-specific functions, only 2 of 83 functions use `state`. In short, the implementation is ugly.

The less obvious, correct, attractive design The type `value list -> value list` leads to nice embedding/projection pairs, and it is the type of the vast majority of our functions. But in the general case, the type of a Lua function is `state -> value list -> value list`. Instead of treating every function as an instance of the general case, our best design treats every function as the special case, and it resolves the mismatch by changing the way functions are called.

When a function is called in one of the flawed designs above, it is applied to state and arguments. But any one interpreter has a unique, mutable state. It is therefore possible to *partially apply* a function to the state at an earlier time, save the resulting closure, and when the function is called, apply the closure only to the argu-

```
strindex : int map
init     : state -> unit
```

```
let strindex =
  { embed   = (fun n -> int.embed (n + 1));
    project = (fun v -> int.project v - 1)
  }

let init interp = register_globals
["strlen",   efunc (string **-> result int)
             String.length;
 "strlower", efunc (string **-> result string)
             String.lowercase;
 "strupper", efunc (string **-> result string)
             String.uppercase;
 "ascii",
   efunc (string **-> default 0 strindex
                             **-> result int)
   (fun s i -> Char.code (String.get s i));
 "strsub",
   efunc (string **-> strindex **-> option strindex
                              **-> result string)
   (fun s start optlast ->
     let maxlast = String.length s - 1 in
     let last = match optlast with
       | None -> maxlast
       | Some n -> min n maxlast in
     let len = last - start + 1 in
     String.sub s start len);
 ... (* many more functions omitted *)
] interp
```

Figure 1: Example embeddings from the Lua string library

ments. The closure keeps its reference to the state throughout its lifetime.

Exactly when to partially apply a function to the state depends on how the function is defined.

- If a function is defined in Lua, the interpreter reads the function's definition and builds a closure of type `state -> value list -> value list`. The interpreter has access to its own state, so it can partially apply the closure at that time.
- If a function is defined in Caml, it can't be used until it is *registered* with the interpreter. Registration might involve putting the function in a global variable, or in a table that is stored in a global variable, or indeed in any Lua data structure—but it always requires access to the interpreter's state. So in the rare, general case, a function can be partially applied to the state at the time that it is registered. Such a function can easily be registered with multiple interpreters, because each partial application creates a closure that captures a different state.

Either way, once a function gets into the interpreter, it has type `value list -> value list`. The `state` parameter, if any, is hidden in a closure.

The benefits of this design are that the API matches the common case, the code for embedding and projection is clean, the design is correct, and the general case is accomodated easily. The trick used to handle general functions can also be used to build embedding/projection pairs that have access to state. For example, our optimizing compiler can project a Lua function into an "optimization pass," in which case it uses the state of the interpreter to find a name by which the function should be known.

4 Experience and discussion

We have used Lua-ML to configure and control an optimizing compiler. The glue code for almost every application-specific function is just a type description, as for `atan2` in Section 2.4.1. The glue code for the Lua libraries is more elaborate, because we use the Caml libraries to implement the Lua libraries, and the semantics don't always match. For example, Figure 1 shows the embedding of some representative functions from the Lua string library.

Figure 1 begins with `strindex`, an embedding/projection pair that embodies a programming convention for strings: Lua-strings are 1-indexed, while Caml strings are 0-indexed. Function `init` is the registration function. The first three functions registered require no glue code, because the Lua and Caml versions match exactly. The fourth function, `ascii`, has no Caml version, but it is easy to write, especially using `default` to handle the default parameter. The last function, `strsub`, requires lots of glue code, because in Caml, the third parameter is a length, but in Lua, it is an optional position. This example is atypical and is about as bad as it gets—a cost of choosing existing, incompatible host and embedded languages.

Higher-order functions and types provide great flexibility to the designer of an API for an embedded language. We have exploited that flexibility to make embedding most functions as easy as writing their types. The main idea is that Danvy's (1996) type-indexed family of functions can be adapted to convert values. Making it work requires some trickery in the embedding of functions, plus careful handling of functions that need access to an interpreter's state.

These ideas don't require much code. The parts of Lua-ML discussed here take about 400 lines of Objective Caml; the whole system fits in 3,400 lines. In size, Lua-ML is comparable to the C implementation of Lua 2.5, which is about 6,000 lines.

Others have avoided writing glue code by generating it automatically. For example, `toLua` (Celes 2003) reads a "cleaned" version of a C header file and generates glue code for the functions declared in that file. "Cleaning" must be done by hand. SWIG (Beazley 1996) is more ambitious; version 1.3.16 generates glue code for nine scripting languages. These program generators offer some of the benefits of Lua-ML, but at much greater cost. The `toLua` tool is 8,000 lines of C, and the SWIG system is about 30,000 lines of C; its C parser alone is 4,500 lines. Eliminating glue code using higher-order functions and types takes a fraction of this effort and is easier for users to extend.

Acknowledgements

João Dias, Simon Peyton Jones, Sukyoung Ryu, and anonymous referees helpfully criticized drafts of this paper.

This work is part of the C-- project and was supported by NSF grant CCR-0096069, by a gift from Microsoft, and by an Alfred P. Sloan Research Fellowship. The code can be downloaded from `www.cminusminus.org`.

References

Beazley, David M. 1996. SWIG: An easy to use tool for integrating scripting languages with C and C++. In USENIX, editor, *Proceedings of the fourth annual Tcl/Tk Workshop*, pages 129–139, Berkeley, CA.

Benson, Brent W. 1994 (October). Libscheme: Scheme as a C library. In *Proceedings of the USENIX Symposium on Very High Level Languages*, pages 7–19.

Benton, Nick. 2003. Embedded interpreters. See `http://research.microsoft.com/~nick`.

Celes, Waldemar. 2003 (March). toLua—accessing C/C++ code from Lua. See http://www.tecgraf.puc-rio.br/~celes/tolua.

Danvy, Olivier. 1996. Type-directed partial evaluation. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 242–257. ACM Press.

———. 1998. A simple solution to type specialization. In *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming (ICALP)*, number 1443 in Lecture Notes in Computer Science, pages 908–917. Springer-Verlag.

Ierusalimschy, Roberto, Luiz H. de Figueiredo, and Waldemar Celes. 1996a (June). Lua — an extensible extension language. *Software—Practice & Experience*, 26(6):635–652.

———. 1996b (November). *Reference Manual of the Programming Language Lua 2.5*. TeCGraf, PUC-Rio. Available from the author.

———. 2001 (May). The evolution of an extension language: A history of Lua. In *V Brazilian Symposium on Programming Languages*, pages B14–B28. (Invited paper).

Jenness, Tim and Simon Cozens. 2002 (July). *Extending and Embedding Perl*. Manning Publications Company.

Laumann, Oliver and Carsten Bormann. 1994 (Fall). Elk: The Extension Language Kit. *Computing Systems*, 7(4):419–449.

Leroy, Xavier, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2001. *The Objective Caml system release 3.04: Documentation and user's manual*. INRIA. Available at http://pauillac.inria.fr/ocaml/htmlman.

Ousterhout, John K. 1990 (January). Tcl: An embeddable command language. In *Proceedings of the Winter USENIX Conference*, pages 133–146.

———. 1994. *Tcl and the Tk Toolkit*. Professional Computing Series. Reading, MA: Addison-Wesley.

Ramsey, Norman. 2003 (March). ML module mania: A type-safe, separately compiled, extensible interpreter. Available from http://www.eecs.harvard.edu/~nr/pubs/mania-abstract.html.

van Rossum, Guido. 2002. *Extending and Embedding the Python Interpreter*. Release 2.2.2.

A Application-specific conversion

Here is an example of C code used to convert a Lua value to and from an application-specific host value of type `Mine`.

```
typedef struct mine *Mine;
int mine_tag;
int ismine(lua_Object o) {
  return lua_isuserdata(o)
      && lua_type(o) == mine_tag;
}

Mine getmine(lua_Object o) {
  assert(ismine(o));
  return (Mine) lua_getuserdata(o);
}

void pushmine(Mine p) {
  lua_pushusertag((void *) p, mine_tag);
}
```

The cast in function `getmine` cannot be checked by the C compiler, but it is safe as long as `mine_tag` is unique. In Lua 2.5, the uniqueness of the tag is up to the application programmer, but Lua 4.0 contains an API function that allocates a unique tag.

B More conversion functions

This appendix presents implementations of more of the conversion functions defined by Lua-ML.

A number may be used where a string is expected.

`string : string map`

```
let string =
  { embed = (fun s -> String s);
    project =
      function String s -> s
             | Number x -> string_of_float x
             | v -> raise (Projection (v, "string"))
  }
```

A list of length n is represented as a table with keys $1..n$.

`list : 'a map -> 'a list map`

```
let list (ty : 'a map) =
  let table l = (* convert list to table *)
    let n = List.length l in
    let t = Table.create n in
    let rec set_elems next = function
      | [] -> ()
      | e :: es ->
         ( Table.bind t (Number next) (ty.embed e);
           set_elems (next +. 1.0) es )
    in (set_elems 1.0 l; Table t)
  in
  let untable (t:table) = (* convert table to list *)
    let n = Luahash.population t  in
    let get_i i =
      Table.find t (Number (Pervasives.float i)) in
    let rec elems i =
      if i > n then []
      else ty.project (get_i i) :: elems (i + 1) in
    elems 1
  in { embed = table;
       project =
         (function
          | Table t -> untable t
          | v -> raise (Projection (v, "list")))
     }
```

We frequently allow nil to stand for the empty list, so we define a convenience function `optlist` for this case.

`optlist : 'a map -> 'a list map`

```
let optlist ty = default [] (list ty)
```

If desired, the `-->` operator can be used to create a Curried Lua function.

`--> : ('a map -> 'b map) -> ('a -> 'b) map`

```
let ( --> ) arg res =
  { embed =
      (fun f -> caml_func (fun args ->
         [res.embed (f (arg.project (take1 args)))]))
  ; project =
     (function
       | Function (_, f) ->
           (fun x ->
             res.project (take1 (f [arg.embed x])))
       | v -> raise (Projection (v, "function")))
  }
```

Executable JVM Model for Analytical Reasoning: A Study

Hanbing Liu
Department of Computer Science
University of Texas at Austin
hbl@cs.utexas.edu

J Strother Moore
Department of Computer Science
University of Texas at Austin
moore@cs.utexas.edu

ABSTRACT

To study the properties of the Java Virtual Machine(JVM) and Java programs, our research group has produced a series of JVM models written in a functional subset of Common Lisp. In this paper, we present our most complete JVM model from this series, namely, M6, which is derived from a careful study of the J2ME KVM[16] implementation.

On the one hand, our JVM model is a conventional machine emulator. M6 models accurately almost all aspects of the KVM implementation, including the dynamic class loading, class initialization and synchronization via monitors. It executes most J2ME Java programs that do not use any I/O or floating point operations. Engineers may consider M6 an implementation of the JVM. It is implemented with around 10K lines in 20+ modules.

On the other hand, M6 is a novel model that allows for analytical reasoning besides conventional testing. M6 is written in an applicative (side-effect free) subset of Common Lisp, for which we have given precise meaning in terms of axioms and inference rules. A property of M6 can be expressed as a formula. Rules of interference can be used analytically to derive properties of M6 and the Java programs that run on the model, using a mechanical theorem prover.

We argue that our approach of building an executable model of the system with an axiomatically described functional language can bring benefits from both the testing and the formal reasoning worlds.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.4 [**Software Engineering**]: Software/Program Verification [Formal methods, Validation, Correctness proofs]; F.3 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs [Specification techniques]

IVME'03, June 12, 2003, San Diego, California, USA.

1. INTRODUCTION

Simulators have been an important tool in the process of designing computer hardware systems. For example, SimpleScalar[3] has been widely used in computer architecture research labs to explore the design space of processors to achieve better performance.

Most simulators of hardware artifacts are either used for detecting flaws in the designs of the artifacts or flaws in the software to be used with those physical artifacts before they are built.

It is our observation that those simulators are usually written with the intention of using them for extensive validation tests. Most of them are written in C to strive for the best simulation speed. Unfortunately, the efforts for speed come at a price. The state representations of those simulators are hard to characterize, because the state of a simulated artifact is mingled with the state of the machine on which the simulator is executed. The meaning of operations of the simulated artifacts can only be expressed as the side effects on the underlying machine state caused by executing the C programs that implement the operations. These make it almost impossible to reason about the simulator rigorously.

In this paper, we present our executable JVM model M6. Different from most other simulators, M6, is not written in an imperative programming language like C. Instead, it is written in ACL2, a functional (side-effect free) subset of Common Lisp, which can also be cast as a mathematical logic[11].

We can execute the simulator as a Common Lisp program. It is derived from a careful study of the C implementation of the Java virtual machine KVM[16]. It supports almost all interesting features of the J2ME JVM[18], including class loading, class initialization, synchronization via monitors as well as exception handling. We are still working towards modeling the access permissions checking accurately, as well as floating point operations.

We hope to achieve a simulation speed of millions of instructions/seconds based on Greve and Wilding's experience with using ACL2 to efficiently model Rockwell Collins JEM1, the world's first Java direct-execution microprocessor[4, 6, 12]. Their optimized simulator of the JEM1 chip in ACL2 runs almost as fast as the original C simulator (at 90% of C speed) at Rockwell Collins. They actually used it to replace the

C simulator back-end in their program development environment, and users of the program-debugging front-end are unaware of the change. Similar optimization of our JVM model in ACL2 has not been done yet.

In addition, we can also treat the JVM simulator as a set of formulas in the ACL2 specification language and we can reason about the JVM model analytically with axioms and rules of inference, which define the meaning of the syntactic constructs and evaluation rules. We use the computer aided reasoning tool, ACL2, to help us produce mechanically checked proofs of the properties of the model. In fact, this aspect has been our group's ultimate focus, while building a complete and accurate JVM model is needed as a firm base for stating the properties of interests.

Our approach is different from conventional simulator designs because it provides support for reasoning analytically about the model and programs that run on the model.

The organization of the paper is as follows. Section 2 describes our approach for implementing the JVM simulator. Section 3 presents some highlights of our JVM model as a simulator. For example, we explain how we model class initialization in some detail in this section. Section 4 describes some proofs for properties we established analytically. Then, we conclude after a discussion of related work.

2. M6: APPROACH

To study the properties of the JVM and Java programs, our research group has developed a series of JVM models at different abstraction levels. They are written in the ACL2 specification language to allow both executability of the model and the suitability for reasoning with the ACL2 system. Different from our previous JVM models in the series [10, 9, 14], the emphasis of the one presented here is on the completeness of the model. We intend it to pass a "comprehensive" Java test suite. It executes the complete set of instructions of a JVM, except for floating point arithmetic operations. We also provide the implementation of a subset of Java native APIs to allow executions of a wide range of realistic Java programs. Complicated aspects such as exception handling, dynamic class loading and class initialization are also carefully modeled. The simulator is implemented in 10K lines of Lisp code in 20+ modules and the source code is online at [8].

In the rest of this section, we sketch the simulator implementation. We also comment on our experience in writing a JVM in a functional programming language.

The JVM interpreter loop is modeled with a Lisp function, run, which takes as its input a "schedule" and a Lisp representation of the JVM state and returns the state obtained by stepping individual threads as specified by the schedule. The semantics of each instruction is given by a corresponding state transition function. Primitives for class resolution, class loading, exception propagation as well as Java native APIs are also modeled with respective Lisp functions.

2.1 State Representation

Because our simulator is written in a functional programming language, it is quite different from most simulators written in C. All aspects of the machine state are encoded explicitly in one logical object denoted by a term.

A JVM state in our simulator is a seven-tuple consisting of a global program counter, a current thread register, a heap, a thread table, an internal class table that records the runtime representations of the loaded classes, an environment that represents the source from which classes are to be loaded, and a fatal error flag used by the interpreter to indicate an unrecoverable error.

The thread table is a table containing one entry per thread. Each entry has a slot for a saved copy of the global program counter, which points to the next instruction to be executed when this thread is scheduled next time. Among other things, the entry also records the method invocation stack (or "call stack") of the thread. The call stack is a stack of frames. Each frame specifies the method being executed, a return pc, a list of local variables, an operand stack, and possibly a reference to a Java object on which this invocation is synchronized.

The heap is a map from addresses to instance objects. The internal class table is a map from class names to descriptions of various aspects of each class, including its direct superclass, implemented interfaces, fields, methods, access flags, and the byte code for each method.

All of this state information is represented as a single Lisp object composed of lists, symbols, strings, and numbers. Operations on state components, including determination of the next instruction, object creation, and method resolution, are all defined as Lisp functions on these Lisp objects.

Below we describe the representations of selected state components in some detail.

Objects Each Java object in the heap has three components:

- A *common-info* section that contains a hash code for this object, a monitor for the JVM and Java programs to synchronize on and the type of this object;
- A *specific-info* section to indicate whether this object is a generic object or an object that represents a class, and, if so, which class it represents;
- A *java visible* section to record the information directly visible to a Java bytecode program and that can be accessed with instructions such as getfield, putfield.

For example, the following is an object of type java.lang.-String:

```
(OBJECT
  (COMMON-INFO 0 (MONITOR ...)
               "java.lang.String")
  (SPECIFIC-INFO STRING)
  (("java.lang.String" ("value" . 89)
                       ("offset" . 0)
                       ("count" . 4))
   ("java.lang.Object")))
```

The hash code is 0. The "java visible" part is a list of immediate fields from java.lang.String and the immediate fields from its super classes, in this case, java.lang.Object.

We need the information stored in these three components to implement primitives for "putfield", "getfield" and monitor enter/exit operations. The information is also needed to implement the native APIs such as the getClass method of java.lang.Object,

Thread table entry We have described the structure of the thread table informally in the beginning of this section. Each thread table entry has slots for recording a thread id, a pc, a call stack, a thread state, a reference to the monitor, the number of times the thread has entered the monitor, and a reference to the Java object representation of the thread in the heap.

As a concrete example, the following entry is taken from an actual thread table when we use our model to execute a multi-threaded program for computing factorial. A semicolon (;) begins a comment extending to the end of the line.

```
(THREAD 0                  ; thread id is 0
 (SAVED-PC . 0)            ; slot for saved pc
 (CALL-STACK
  (FRAME (RETURN_PC . 7)   ; pc to return to
         (OPERAND-STACK)   ; empty operant stack
         (LOCALS 104)
         (METHOD-PTR "FactHelper" "<init>" ...)
         (SYNC-OBJ-REF . -1))
  (FRAME (RETURN_PC . 18)
         (OPERAND-STACK 104)
         (LOCALS 102)
         (METHOD-PTR "FactHelper" "compute"...)
         (SYNC-OBJ-REF . -1))
   ...)
 (STATUS THREAD_ACTIVE)     ; thread state
 (MONITOR . -1)             ; lock
 (MDEPTH . 0)               ; entering count
 (THREAD-OBJ . 55))         ; object rep in heap
```

Method representation We have developed a tool, *jvm2-acl2*, which takes Java class files and converts them into a format to be used with our model. Each class file is converted into a Lisp constant. The environment component of a JVM state contains an external class table, which is composed of a list of such Lisp constants. The class loader in our JVM model reads from the external class table and constructs a runtime representation of the class in the internal class table. The following is an example to illustrate how the Java method is represented in the internal class table of our state representation.

```
public static String valueOf(char data[]) {
       return new String(data);
}
```

is represented as

```
(METHOD
 "java.lang.String"
 "valueOf"
 (PARAMETERS (ARRAY CHAR))
 (RETURNTYPE . "java.lang.String")
 (ACCESSFLAGS *CLASS* *PUBLIC* *STATIC*)
 (CODE (MAX_STACK . 3)
       (MAX_LOCAL . 1)
       (CODE_LENGTH . 9)
       (PARSEDCODE
        (0 (NEW (CLASS "java.lang.String")))
        (3 (DUP))
        (4 (ALOAD_0))
        (5 (INVOKESPECIAL
             (METHODCP "<init>"
                       "java.lang.String"
                       ((ARRAY CHAR)) VOID)))
        (8 (ARETURN))
        (ENDOFCODE 9))
       (EXCEPTIONS)
       (STACKMAP)))
```

2.2 State Manipulation Primitives

Our JVM state is represented explicitly as a Lisp logical object. We define a set of primitives to manipulate the object. Many of those primitives correspond to the operations and procedures described in the JVM specification[19]. Some of them correspond to the native APIs that are implementation dependent. Others are utilities that we introduce for implementing our simulator.

For example, to find the next instruction for execution in state s, we use the following primitives. We first use the primitive current-method-ptr to find method-ptr – the method pointer associated with the top frame of the call stack of the current thread, then we use deref-method to look up the the method referred by method-ptr in the instance-class-table of s, and we return the instruction at offset (pc s) of the method. In Lisp, (pc s) denotes the value of the Lisp function pc when applied to s, i.e., $pc(s)$.

```
(defun next-inst (s)
  (let* ((ip  (pc s))
         (method-ptr (current-method-ptr s))
         (method-rep
            (deref-method method-ptr
                          (instance-class-table s)))
       (inst-by-offset ip method-rep)))
```

As another example, the following function is our internal primitive for invoking a Java method.

```
(defun call_method_general (this-ref method s0 size)
  (let ((accessflags (method-accessflags method))
        (s1 (state-set-pc (+ (pc s0) size) s0)))
    (cond ((mem '*native* accessflags)
           (invokeNativeFunction method s1))
          ((mem '*abstract* accessflags)
           (fatalError "abstract_method invoked" s0))
          (t
      (let ((s2 (pushFrameWithPop method ...)))
           (if (mem '*synchronized* accessflags)
```

```
        (mv-let (mstatus s3)
                (monitorEnter this-ref s2)
          (set-curframe-sync-obj this-ref s3))
      s2))))))
```

To invoke a synchronized non-native, non-abstract method, we first need to pop the proper number of values off the current operand stack, and push a suitable call frame on top of the current call frame. We achieve these effects with pushFrameWithPop. Then, we call monitorEnter to obtain the monitor on the this-ref. We mark the call frame to indicate this invocation has the monitor on this-ref. The resulting state is returned so that the top level interpreter loop can proceed.

Notice monitorEnter mentioned above is just another state manipulating primitive that takes a reference and a state and returns a status flag and a state. There are two kinds of the states that can be returned by monitorEnter. If monitorEnter succeeds, the state returned is representing a state in which the current thread is holding the monitor and the current thread is still active. If monitorEnter does not succeed — because some other thread is holding the monitor — then state returned represents a state in which the current-thread is suspended and moved to the waiting queue for the monitor.

As our last example in this section, we introduce the following Lisp function to implement the native method currentThread of java.lang.Thread. It takes a state as its input and pushes a reference to the Java object that represents current thread on the operand stack of the current frame.

```
(defun Java_java_lang_Thread_currentThread (s)
 (let*
  ((tid (current-thread s))
   (thread-rep (thread-by-id tid (thread-table s)))
   (thread-ref (thread-ref thread-rep)))
  (pushStack thread-ref s)))
```

2.3 State Transition Function

We model the semantics of the JVM instructions operationally, in terms of such primitives as discussed above. The meaning of executing a JVM instruction is given by a state transition function on the JVM states. Here is the state transition function for the IDIV instruction.

```
(defun execute-IDIV (inst s)
 (let ((v2 (topStack s))
       (v1 (secondStack s)))
  (if (equal v2 0)
      (raise-exception
          "java.lang.ArithmeticException" s)
   (ADVANCE-PC
     (pushStack (int-fix (truncate v1 v2))
                (popStack (popStack s)))))))
```

Here, inst is understood to be a parsed IDIV instruction. ADVANCE-PC is a Lisp macro to advance the global program counter by the size of the instruction. PushStack pushes a value on the operand stack of the *current frame* (the top call frame of the current thread) and returns the resulting state. When the item on the top of the operand stack of the current frame is zero, the output of execute-IDIV is a state obtained from s by raising an exception of type java.lang.ArithmeticException. If the top item is not zero, the resulting state is obtained by changing the operand stack in the current frame and advancing the program counter. The operand stack is changed by pushing a certain value (described below) onto the result of popping two items off the initial operand stack. The value pushed is the twos-complement integer represented by the low-order 32-bits of the integer quotient of the second item on the initial operand stack and by the first item on it.

For a more complicated instruction, here is our semantics for invokestatic.

```
(defun execute-invokestatic (inst s)
  (let* ((cp    (arg inst))
         (method-ptr (methodCP-to-method-ptr cp)))
   (mv-let (method new-s)
           (resolveMethodReference method-ptr t s)
    (if method
     (let*
      ((class (static-method-class-rep method new-s))
       (cname (classname class))
       (cref  (class-ref class)))
      (if (class-rep-in-error-state? class)
          (fatalError "class in error state!" new-s)
       (if (not (class-initialized? cname  new-s))
           (initializeClass cname new-s)
         (call_static_method cref method new-s))))
      (fatalSlotError cp new-s))))
```

To invoke a static method, we first get the constant pool entry and obtain a symbolic method-ptr. Then we call the method resolution procedure resolveMethodReference to find the method that the method-ptr resolves to. Notice that the method resolution may cause class loading so the procedure returns a pair (method, new-s) as declared in the mv-let. The fatal error flag is set when the method is nil, i.e. the method resolution failed. If the class to which the method belongs has not been initialized, we choose not to advance the program counter, instead returning a state prepared for starting the class initialization procedure using (initializeClass cname new-s). Otherwise, we use call_static_method to return a state properly prepared for the interpreter to start executing the resolved method.

3. M6: A SIMULATOR

Not only have we modeled the semantics of most JVM instructions, we have also provided implementations for native APIs. As a result, we can use the formal model as a simulator to execute realistic JVM programs.

We have translated the entire Sun CLDC API library implementation into our representation with 672 methods in 87 classes[16]. We provide implementations for 21 out of 41 native APIs that appear in Sun's CLDC API library. The remaining ones are mostly related to the I/O of the JVM, which we do not model in our current simulator.

We have written several test programs to run on this model to exercise various aspects of the simulator such as exception handling, synchronization, class initialization. One of the test programs we run is a multi-threaded Java program that implements an impractical but illuminating parallel factorial algorithm.

The program takes a command line parameter represented in a java.lang.String. It calls the method parseInt of java.lang.Integer, which in turn invokes a dozen API functions, to parse the string and return an integer. Then it spawns a specified number of FactHelper threads (5 in the current version). Those threads share an instance of FactJob, in which the intermediate result is stored. Those FactHelpers repeatedly compete for the monitor on the FactJob, compute one iteration and then release the monitor by executing monitorexit. The main thread prints the result and quits when it is awakened by a notify call from any of the FactHelpers indicating that the computation has terminated.

```
class FactJob {
    int value = 1;
    int n;
    ... };

class FactHelper implements Runnable {
    FactJob myJob;
    public void run() {
      ... wait for notifyAll on myJob.
      for (;;) {
        synchronized(myJob) {
         if (myJob.n<=0) {
             myJob.notify();
             return; // done
         } else {
             myJob.value = myJob.value*myJob.n;
             myJob.n = myJob.n - 1;
         }}}}
    ...};

public class Fact {
    static int HELPER_COUNT = 5;

    public static int fact(int n) {
        FactJob theJob = new FactJob(n);
        ... spawn HELPER_COUNTER FactHelpers.
        ... send notifyAll to FactHelpers
        try{
         synchronized (theJob){ theJob.wait();};
         //wait for at least one Helper finishes.
        } catch ...
       return theJob.value;};
}
```

The main method is:

```
public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    System.out.println(""+Fact.fact(n));
}
```

To compute factorial of 10, the simulator executes 1748 steps, most of which are spent in the class initialization, parsing an integer from a string, string copying and printing the result character by character. In the process, 5 threads are spawned, 118 heap object are created, and 18 classes are loaded and initialized. A transcript of this execution is online.[1]

In the rest of this section, we pick some aspects of our simulator and explain them in some more detail.

3.1 Interpreter Loop

As mentioned previously, our JVM model takes a "schedule" (a list of thread ids) and a state as the input and repeatedly executes the next instruction from the thread as indicated in the schedule, until the schedule is exhausted.

```
(defun run (sched s)
  (if (endp sched)
      s
    (let ((nid (car sched))
          (cid (current-thread s)))
      (if (equal cid nid)
          (run (cdr sched) (step s))
        (run (cdr sched)
             (loadExecutionEnvironment nid
               (storeExecutionEnvironment s)))))))
```

The scheduling policy is thus left unspecified. Any schedule can be simulated. However to use the model as a JVM simulator without providing a schedule explicitly, we have implemented some simple scheduling policies. One of them is a not very realistic round-robin scheduling algorithm, which does a rescheduling after executing each instruction.

3.2 Class Initialization

Class initialization is a very complicated aspect of the JVM. Multi-threading adds to the complexity. To initialize a class, the JVM needs to execute a special class initialization method, thus the initialization of a class cannot be made atomic. The process needs careful synchronization between threads that may try to initialize the same class at the same time. We explain this aspect of our simulator in the most detail.

Modeled after the KVM implementation[16], our simulator follows the 11-step algorithm described in the JVM specification Sec. 2.17.5.[19] and completes a class initialization in 6 stages . Recall the state transition function that describes invokestatic. If the class to which the resolved method belongs has not been initialized the function does not advance the pc; instead it returns a state by calling (initializeClass classname s). Roughly, the state returned is a special state recognized by the interpreter loop as indicating the start of the class initialization process. Each thread maintains a simple finite state machine to keep track of the class initialization stage that the thread is currently in. The complete details are in jvm-static-initializer.lisp of [8].

```
(defun runClinit1 (classname s)
 (let ((cid (current-thread s)))
```

[1]See `pftrace.lisp` in [8]

```
(mv-let (mstatus new-s)
        (classMonitorEnter classname s)
        (if (not (equal mstatus 'MonitorStatusOwn))
            (set-cinit-stage  cid 2 new-s)
          (runClinit2 classname new-s))))))
```

To initialize a class, the current thread first needs to acquire the monitor associated with the class. This is necessary because we need to cope with the situation that multiple threads may be trying to initialize the same class at the same time. If the acquisition attempt fails, i.e. the mstatus is not equal to MonitorStatusOwn, we return the following state to the interpreter loop: the original current thread is suspended (as a consequence of failing to acquire the monitor) and the class initialization process stage of the thread is set to 2 with (set-cinit-stage cid 2 new-s).

```
(defun runClinit2 (classname s)
 (let*
  ((class-rep  (class-by-name classname ...))
   (initThread (init-thread-id class-rep))
   (cid        (current-thread s)))
  (cond
   ((and (not (equal initThread -1))
         (not (equal initThread (current-thread s))))
    (let ((new-s (classMonitorWaitX classname s)))
        (set-cinit-stage  cid 2 new-s)))
   ((or (equal initThread (current-thread s))
        (class-initialized? classname s))
    ....)
   (t (let ((sinit
             (setClassInitialThread classname cid s)))
      (mv-let (mstatus exception-name s-new)
              (classMonitorExitX classname sinit)
              (runClinit3 classname s-new)))))))
```

In stage 2, we first check whether some other thread is initializing the class. If some other thread is initializing the class, we return a state in which the current thread is suspended to wait for a notify signal from the monitor and the class initialization stage remains at 2. If no other thread is initializing the class, we check whether the class has already been initialized. If it has not been initialized, we mark the state to indicate that the current thread is initializing the class and return the resulting state to the interpreter loop.

```
(defun runClinit3 (classname s)
  (let ((class-rep (class-by-name classname ...))
        (cid (current-thread s)))
     (if (not (isInterface class-rep))
         (if (and (super-exists class-rep)
                  (not (class-initialized?
                             (super class-rep) s)))
             (initializeClass
                          (super class-rep)
                          (set-cinit-stage cid 4 s))
           (runClinit4 classname s))
       (runClinit4 classname s))))
```

In stage 3 of class initialization, we first check whether the super class has been initialized. If the super class is not initialized yet, we use initializeClass to prepare a state so that the interpreter loop can recognize and start the initialization process for the super class. We also properly mark the state, so that when class initialization for the super class is completed, the process of initializing the current class enters stage 4.

```
(defun runClinit4 (classname s)
  (let* ((clinit-ptr (clinit-ptr classname))
         (thisMethod (getSpecialMethod clinit-ptr s))
         (cid (current-thread s)))
    (if thisMethod
        (pushFrame clinit-ptr nil
                   (set-cinit-stage cid 5 s))
      (runClinit5 classname s))))
```

Now we are in class initialization stage 4. To complete the class initialization, the interpreter must execute the class initialization method of the class. We achieve this by pushing a call frame for the class initilization method to the call stack. We also mark the state so that when the interpreter is done with the invocation of the class initilization method, it can recognize that it needs to resume the class initilization process from stage 5. In stage 5, we try to acquire the monitor associated with the class. And in stage 6, the class initialization process is completed, we send notifyAll signal, so that threads that are waiting in stage 2 can proceed.

4. M6: A FORMAL MODEL

The focus of our research has been developing a practical methodology for reasoning about complex hardware and software artifacts, including models of the complexity of the one described here. We have shown that our executable JVM model includes enough detail to permit its use as a simulator. But it has dual use as a formal semantics of the JVM and as such permits us to reason about the JVM and its bytecoded methods. It is easy to formulate simpler, less accurate models of the JVM (we have a series of less accurate models ourselves). Whether there is a simpler semantics that is as accurate as this one is an open question as far as we are concerned. Even should a simpler useful semantics be developed, this model is an excellent stepping-stone toward the verification of an implementation of the JVM (e.g., the J2ME KVM[16] from which it was produced) and is worthy of study in that context.

We study two broad categories of the properties using this accurate JVM model. We use our JVM model to study the properties of the JVM specification and the J2ME KVM implementation. We also use the formal model to study the properties of Java programs that run on "real" (practically efficient) JVMs. In the rest of the section, we present some work we have done in these two categories.

We are interested in studying dynamic class loading in the JVM. We have formally proved an invariant of dynamic class loading in our JVM model. The invariant says if class A is loaded, and if a value of type class A is assignable to a slot of type class B, then, class B must be already correctly loaded. As a relevant note, the concept of *assignable to* is rather complicated. For a value of type class A to be assignable to a slot of type class B, B must be a direct superclass of A, one of the direct super interfaces of A, or there must be a class C, where values of type class C are assignable to a slot of type class B, and class C must be a direct superclass of class A or one of the direct super interfaces of class A.

The JVM relies on this invariant about loaded classes to behave correctly in field resolution and method dispatching. In fact, any operations that involves examining the class hierarchy by searching through super class chain rely on this property to operate correctly. It is useful to be sure that this alleged "invariant" is in fact preserved by the class loader in the JVM.

We have proved this invariant is preserved by the class loader in this formal model of the JVM. We achieve this in two steps. As the first step, we proved a different formulation of the same invariant, which roughly says, all classes reachable from any loaded class through its superclass chain and super interfaces chains are also loaded. We defined a Lisp function collect-assignableToName that crawls along the class hierarchy and collects all reachable classes. The invariant loader-inv asserts that for all classes currently in the class table, the following loader-inv-helper1 is true. We have proved that load_class preserves the invariant[2].

```
(defun loader-inv-helper1
   (class-rep class-table env-class-table)
   (let* ((classname (classname class-rep))
          (supers (collect-assignableToName
                    classname env-class-table)))
     (all-correctly-loaded?
          supers class-table env-class-table)))

 (defthm loader-inv-is-inv-respect-to-loader
     (implies (loader-inv s)
              (loader-inv (load_class classname s)))
```

In the second step, we show the above alternative formulation implies our original formulation of the invariant.

```
(defthm inv-and-isAssignableTo-env
  (implies
    (and (loader-inv s)
         (no-fatal-error? s)
         (isAssignableTo-env A B
                       (env-class-table (env s))))
     (implies (correctly-loaded? A
                   (instance-class-table s)
                   (env-class-table (env s)))
              (correctly-loaded? B
                   (instance-class-table s)
                   (env-class-table (env s))))))
```

A more ambitious project that uses this formal JVM model is to study the adequacy of current CLDC bytecode verification algorithm as described in JSR-139 for J2ME JVMs[17]. We want to show that "verified" bytecode will behave as we expected if the CLDC's bytecode verification algorithm is adequate.

Our approach to the ultimate goal of verifying a bytecode verifier is taken from Cohen's original work[1]: We introduce a "defensive" version of our model that checks for "all possible" runtime anomalies. We study whether the current CLDC specification can provide the guarantee that verified programs never cause anomalies in this JVM model by trying to prove a theorem that states the property.

We already have some partial results about the correctness of the bytecode verification alogrithm on one of our simpler JVM models, M3, see [7]. For example, we have proved that executing any program on a regular JVM (M3) computes the "same" result as executing it on a "defensive" but inefficient JVM, when the "defensive" JVM does not detect any anomaly. We also proved that any executions on "defensive" JVM preserves a *consistent-state* predicate, which implies that starting from a consistent initial state, a "defensive" JVM always has well-formed heaps, for example, pointers in the system always point to valid objects. We also proved that the original iterative bytecode verification algorithm terminates on all bytecode programs.

Besides reasoning about the properties of the JVM model itself, we use the model to reason analytically about the programs that run on it. This gives us opportunities for finding program flaws beyond those likely to be uncovered by simulation of executions.

For example, we have proved a single threaded program for computing factorial is correct. The property is stated with the following ACL2 formula. The ACL2 system mechanically checked our proof of the theorem.

```
(defthm factorial-is-correct
 (implies
     (and (poised-for-execute-fact s)
          (integerp n)
          (<= 0 n)
          (intp n)
          (equal n (topStack s)))
   (equal (simple-run s (fact-clock n))
          (state-set-pc (+ 3 (pc s))
                        (pushStack (int-fix (fact n))
                                   (popStack s)))))
```

where poised-for-execute-fact specifies that the state is a well formed state, the next instruction will invoke the factorial method, and the class table is a particular one where, after method resolution, the method found is actually a particular bytecoded factorial method. The theorem itself says the effect of executing fact-clock(n) steps from a state "poised" for executing the factorial program, is equal to setting the program counter to point to 3+pc(s) and replace the top of the operand stack with the lower 32 bits twos-complements of the mathematical fact(n).

As an observation, this theorem is a specification for total functional correctness of the corresponding bytecode facto-

[2] To prove load-inv is an invariant of any execution, we need to show that it is preserved not only by load_class primitive, but also by all machine steps. We have not done that yet.

rial program. The proof of this specification is mechanically checked by the ACL2 system.

On a simpler model, we have also proved theorems about methods that change the heap such as a linked-list implementation of insertion sort[13]. We proved that executing insertion sort method with a pointer to the first node of a linked list. The sorting method returns returns a pointer that points to an ordered permutation of the input list.

We have proved theorems about multi-threaded Java programs that synchronize through a monitor to gain exclusive access to the shared resources[14]. In this example, an unbounded number of threads repeatedly compete for a lock on a container to increase a counter in the container. We proved that the counter goes up monotonically until it wraps around because of an overflow.

Our group has proved theorems relating program execution on one model to execution on another, simpler model. One such example is reported in [15] where the theorem allows us to move from a multi-threaded model to a single-threaded model for programs that are, in some suitable sense, inherently single-threaded. The final theorem in that paper basically says that for any program that satisfies a certain syntactical restriction, effects of executing it on the multi-threaded JVM model can be obtained by executing the program on a single-threaded JVM. We intend to prove such reduction theorems about this model as well, to simplify code proofs for certain classes of methods.

5. RELATED WORK

Cohen uses ACL2 to formalize a single threaded "defensive" machine[1]. His emphasis is on using a formal language to precisely specify the correct behaviors of the JVM. His work inspired us to develop a series of models to study the various aspects of the JVM formally.

Unlike Cohen's work, we model class loading and class initialization, as well as exception handling. We also provide implementations for 21 of 41 native APIs to allow the execution of a wide range of realistic Java programs. The model can be used as a realistic JVM simulator.

Our work is different from most other simulators because it is implemented in a functional programming language for which there is automated reasoning support.

The simpler single threaded Java Card Virtual Machine was modeled at INRIA, Sohphia-Antipolis, France [5]. They give a formal executable semantics for the Java Card Virtual Machine in Coq[2], an ML-like language. They claim the model is quite complete and can execute any JavaCard program — given they provide the native implementations of necessary API functions. However it is not clear whether they have actually provided the native implementation.

Work in progress in our group includes both the study of the bytecode verifier and the development of tools for easing the human burden of producing proofs of bytecode programs. We have formalized the CLDC bytecode verifier algorithm as described in [17] with the aim of mechanically proving that verified programs never cause anomalies on our JVM model.

Another member of our research group, Jeff Golden, is developing a methodology for systematically reasoning about the behavior of programs on our formal JVM models. As part of his research, he is developing facilities to support symbolic execution of bytecode. We believe symbolic execution can be helpful in catching software bugs. In addition, he is developing tools to produce mechanically from bytecode the "functional semantics" described in ACL2. It is our intention that these tools will also generate machine-checked proofs of the correctness of that semantics with respect to our operational models. Thus, the problem of verifying a bytecoded method will be reduced to proving a theorem about a simpler (albeit still complicated) ACL2 function. The two projects mentioned above – the CLDC bytecode verifier work and the automated semantics work – are works in progress.

6. CONCLUSION

In this paper, we describe several selected aspects of our realistic JVM simulator. We presented our general approach for writing such a detailed JVM model in a functional programming language. The state is represented with a Lisp object. The semantics of executing a bytecode instruction in a JVM state is defined by a state transition function. We show that this approach allows us to model the delicate internals of a realistic JVM, use it as a simulator, and prove theorems about the resulting behaviors. This approach brings the benefits of both simulation and machine-checked analytical reasoning.

7. REFERENCES

[1] R. Cohen. Defensive Java Virtual Machine Version 0.5 alpha Release. Available from `http://www.cli.com/software/djvm/index.html`, 1997.

[2] The Coq Development Team. Technical report.

[3] D. Burger and T.M. Austin. The SimpleScalar tool set, Version 2.0. Technical Report 1342, Universit of Wisconsin-Madison Computer Science Department, June 1997.

[4] D. Hardin, D. Greve, M. Wilding, and J. Cowles. Single-threaded formal processor models: Enabling proof and high-speed execution. Technical report, Rockwell Collins Advanced Technology Center, Cedar Rapids, IA 52498, 1999.

[5] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A formal executable semantics of the javacard platform. In D. Sands, editor, *Proceedings of ESOP'01*, 2001.

[6] D. Greve. Symbolic simulation of the JEM1 microprocessor. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD'98)*, pages 321–333, Palo Alto, CA, 1998.

[7] H. Liu and J S. Moore. A formal study of bytecode verification in ACL2. Partial results on the correctness

of the bytecode verification algorithm for the JVM model M3, unpublished, March 2002.

[8] H. Liu and J S. Moore. JVM model: M6 source code. http://www.cs.utexas.edu/users/hbl/pub/M6/ivme03/, March 2003.

[9] J S. Moore and G. Porter. An executable formal java virtual machine thread model. In *Proceedings of 2001 JVM Usenix Symposium*, Monterey, California, April 2001. USENIX.

[10] J S. Moore, R. Krug, H. Liu, and G. Porter. Formal models of Java at the JVM level a survey from the ACL2 perspective. In *Workshop on Formal Techniques for Java Programs*. 2001.

[11] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-aided Reasoning: An approach*. Kluwer Academic Publishers, 2000.

[12] M. Wilding, D. Greve, and D. Hardin. Efficient simulation of formal processor models. *Formal Methods in System Design*, 18(3), May 2001.

[13] J S. Moore. Proving theorems about Java and the JVM with ACL2. In M. Broy, editor, *Lecture Notes of the Marktoberdorf 2002 Summer School*. Springer, 2002.

[14] J S. Moore and G. Porter. The apprentice challenge. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):193–216, 2002.

[15] G. Porter. A commuting diagram relating threaded and non-threaded JVM models. Technical report, Honors Thesis, Department of Computer Sciences, University of Texas at Austin, 2001.

[16] Connected Limited Device Configuration (CLDC) and the K Virtual Machine. http://java.sun.com/products/cldc/.

[17] Connected Limited Device Configuration (CLDC) Specification 1.1. http://jcp.org/en/jsr/detail?id=139.

[18] Java 2 Platform, Micro Edition. http://java.sun.com/j2me/.

[19] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publisher, second edition, 1999.

A Portable Virtual Machine Target For Proof-Carrying Code

Michael Franz, Deepak Chandra, Andreas Gal, Vivek Haldar, Fermín Reig, Ning Wang
{franz,dchandra,gal,vhaldar,reig,wangn}@uci.edu
Department of Computer Science
University of California
Irvine, CA 92697-3425

ABSTRACT

Virtual Machines (VMs) and Proof-Carrying Code (PCC) are two techniques that have been used independently to provide safety for (mobile) code. Existing virtual machines, such as the Java VM, have several drawbacks: First, the effort required for safety verification is considerable. Second and more subtly, the need to provide such verification by the code consumer inhibits the amount of optimization that can be performed by the code producer. This in turn makes just-in-time compilation surprisingly expensive. Proof-Carrying Code, on the other hand, has its own set of limitations, among which are the sizes of the proofs and the fact that the certified code is no longer machine-independent. In this paper, we describe work in progress on combining these approaches. Our hybrid safe-code solution uses a virtual machine that has been designed specifically to support proof-carrying code, while simultaneously providing efficient just-in-time compilation and target-machine independence. In particular, our approach reduces the complexity of the required proofs, resulting in fewer proof obligations that need to be discharged at the target machine.

1. INTRODUCTION

A considerable amount of effort has recently been invested into (mobile) code safety. The general idea is simple: rather than *trusting* a piece of code because it came from a specific provider (for example, because it was purchased in a box in a reputable store or because it was digitally signed), we *verify* the code prior to execution. Verification means determining the code's safety *by examining the code itself* rather than where it came from.

Over the past few years, three main approaches to safe code[1] have been developed. They are, in turn, *virtual machines with code verification* [10], *proof-carrying code* [11], and *inherently safe code formats* [3, 7, 2].

In virtual machines with code verification, the code is examined to ensure that the semantic gap between the source language and the virtual machine instruction format is not exploited. For example, virtual machines have general *GOTO* instructions but not all possible control flows are actually legal[2]. As another example, the language definition of Java requires every variable to be initialized before its first use—unless control flow is strictly linear, this property cannot be inferred trivially from the virtual machine program but requires the verifier to perform dataflow analysis.

In proof-carrying code solutions, the code producer attaches a *safety proof* to an executable. Upon receiving the code, the recipient examines it and calculates a *verification condition* from the code. The verification condition relates to all the potentially unsafe constructs that actually occur in the executable. It is the task of the code producer to supply a proof that discharges this verification condition, or else the code will not be executed.

In the third approach, an *inherently safe code format* is used to transport the mobile program, making most aspects of program safety a well-formedness criterion of the mobile code itself. Checking the well-formedness of such a format is much simpler than verifying bytecode. The disadvantage is that a much more complex and memory-intensive machinery is required at the code recipient's site, as inherently safe formats are based on compression using syntax and static program semantics. As a consequence, this approach is less well suited for resource-constrained client environments.

In the following, we describe our hybrid approach that combines the first two solutions and applies elements of the third. The aim of our research is to find the "sweet spot" reconciling high execution performance of the final code, high dynamic compilation efficiency, small proof size of the proof carrying code component, and limited resource consumption on the client computer.

The paper is structured as follows: First, we discuss the Java

[1] We much prefer the term *safe code* to the term *mobile code,* for two reasons. First, all code is becoming "mobile", with program patches and whole applications increasingly distributed via the Internet. Second, we believe that in the not so far future, *all* code resident on desktop computers (outside of a small hardware-secured trusted computing base) will be verified prior to every execution.

[2] We note that code obfuscators often work by producing irreducible control flows that, while verifiable, don't easily map back onto the control structures of the source language.

IVME'03, June 12, 2003, San Diego, California, USA.

Virtual Machine, not only because it is a good representative of the VM genre, but also because it is the de-facto standard for transporting mobile code. Then, we give an overview of proof-carrying code (Section 3). Section 4 presents the case for a new virtual machine specifically designed for proof-carrying code. Section 5 gives a sketch of the architecture we are currently implementing—this architecture is still being refined and we are merely presenting the current state of a work in progress. This section also contains several examples. Section 6 expands on the proof-carrying code aspects of our work. Section 7 presents related work. Finally, we give an outlook to future work and conclude the paper.

2. THE JAVA VIRTUAL MACHINE

The Java Virtual Machine's bytecode format ("Java bytecode") has become the de facto standard for transporting mobile code across the Internet. However, it is generally acknowledged that Java bytecode is far from being an ideal mobile code representation—a considerable amount of preprocessing is required to convert Java bytecode into a representation more amenable to an optimizing compiler, and in a dynamic compilation context this preprocessing takes place while the user is waiting. In addition, it has been shown that the rules for bytecode verification don't exactly match those of the Java Language Specification, so that there are certain classes of perfectly legal Java programs that are rejected by all compliant bytecode verifiers[13].

Further, due to the need to verify the code's safety upon arrival at the target machine, and also due to the specific semantics of JVM's particular security scheme, many possible optimizations cannot be performed in the source-to-Java bytecode compiler, but can only be done at the eventual target machine—or at least they would be very cumbersome to perform at the code producer's site.

For example, information about the redundancy of a type check may often be present in the front-end (because the compiler can prove that the value in question is of the correct type on every path leading to the check), but this fact cannot be communicated safely in the Java bytecode stream and hence needs to be re-discovered in the just-in-time compiler. By "communicated safely", we mean in such a way that a malicious third party cannot construct a mobile program that falsely claims that such a check is redundant. Or take common subexpression elimination: a compiler generating Java bytecode could in principle perform common subexpression elimination and store the resulting expressions in additional, compiler-created local variables, but this approach is incomplete because it cannot eliminate common address calculations (for arrays etc.)—since verification requires preserving the language abstractions, such optimizations for the JVM can be performed only *after* verification (i.e., on the target machine).

The problem here is that just-in-time compilation occurs while interactive users are waiting for execution to commence. If dynamic compilation time were unbounded, one would be able to perform extensive optimization and obtain a high code quality. Because of their interactive setting, however, just-in-time compilers often need to make a trade-off between *(verification+compilation)-time* on one side, and *code quality* on the other—for example, by employing faster linear-scan register allocation instead of slower but better graph-coloring algorithms.

3. PROOF-CARRYING CODE

Proof-carrying code (PCC) is a framework for ensuring that untrusted programs comply with a safety policy defined by the system where the programs will execute. Typical policies are type, memory and control-flow safety, but in the framework described by Necula [11], any property of the program that can be expressed in first order logic constitutes a valid safety policy.

Upon reception of an untrusted program, the code consumer examines the code and emits a proof obligation for all operations that are potentially unsafe with respect to the safety policy. For instance, we might require a proof that a memory write lies within the bounds of a certain array allocated in the stack. (Proof obligations are traditionally called *verification conditions*, or VCs.)

The provider of an untrusted program must supply a proof for all the VCs. If a proof is missing, or the code consumer determines that it does not constitute a proof of the VC, the program is determined potentially unsafe and will not be executed. All this is done by examining the program and the proofs alone, without dependence on digital signatures or other mechanisms based on trust.

An important practical aspect of PCC is the size of proofs and the time spent in proof checking. It has been shown that proofs for Java type safety can be compressed to 12%–20% of the size of x86 machine code (a factor of 30 reduction with respect to a previous scheme), but unfortunately this increases the proof checking time by a factor of 3 [12]. Notice that this work compresses the proofs, but does not reduce the number of facts that need to be proved.

In some mobile code contexts, 20% space overhead or large checking times are considered too high. Our work aims at reducing the size of proofs and the proof checking time by generating smaller VCs. For instance, by having a separate register file for booleans, no VCs are required to state the type of values contained in such registers.

Also, PCC has only been demonstrated in the context of machine code. Our work carries over the same ideas to a machine-independent format, with all the advantages that this represents.

4. THE CASE FOR A VIRTUAL MACHINE TO SUPPORT PROOF-CARRYING CODE

One of the reasons why PCC often has very large proofs is because the level of reasoning is very low, i.e., machine code level. At this level, registers and memory are untyped, and worse still, there is no differentiation between data values and address values (pointers). A large portion of each proof typically re-establishes typing of data, for example, distinguishing Integers from Booleans and from pointers.

Interestingly, current research on PCC (such as *Foundational Proof Carrying Code* [4]) appears to be directed solely at reducing the size of the trusted computing base on the target platform. Unfortunately, this increases the volume of the proofs that are required even further.

We believe that it is much more promising to go the other way: by raising the semantic level of the language that the proofs reason about, the proofs can become much smaller. Facts that previously required confirmation by way of proof now can be handled by axioms. Our goal is to find such a higher semantic level that is at once effective at supporting proof-carrying code in this manner, and that can also be translated efficiently into highly performing native code on a variety of target platforms.

As an example of what we mean by "higher semantic level", imagine a virtual machine that supports the concept of *tagged memory*, areas of memory that have a tag stored at an offset from the pointer that is unreachable via the regular memory access instructions. The virtual machine guarantees this property, i.e., the regular memory access instructions can access only locations that lie *within the data area* of a memory block—accesses are verified to lie within the range (beginning of block-end of block), which doesn't include the tag. Conversely, access to the tag area requires one of two *privileged instructions*, only one of which reads and the other of which writes the tag value. Such an architecture greatly simplifies certain proofs: any fragment of code that doesn't use the privileged "write-to-tag" instruction cannot have changed the tag. Since at the higher level the tag relates to the (dynamic) type of a memory object, that implies that the type has remained constant.

Our aim is to create a software defined layer at which proof carrying code and dynamic translation can meet effectively. Hence, we also need to demonstrate the second half of the equation: how to make such a virtual machine efficiently implementable. The key here is our use of type separation and referentially-safe encodings on one hand (as previously demonstrated in the safeTSA project [3]), and of an intricate memory addressing scheme on the other hand.

5. ARCHITECTURE

The central element of our architecture is a division of concerns between the proof-carrying code mechanism and the virtual machine layer.

The virtual machine layer is designed in a way to reduce the burden of proof by providing a number of inherently safe operations. As such safe operations often involve a runtime overhead, the safe vs. the non-safe properties of the VM have to be carefully balanced. We choose to include only those safe-by-construction mechanisms which we expect will have no or very little overhead compared to equivalent non-safe operations. The proof-carrying code mechanism only has to provide proof for the safety of the remaining parts of the VM architecture.

Our VM architecture provides a finite number of scalar data types and complete type separation between these types, except for explicit conversion operations. The VM has a separate register set (of unbounded size) for each of the basic types.

i_n: integer registers
b_n: boolean registers
p_n: pointer registers
a_n: address registers

The typed register sets in our architecture play a key role in reducing the complexity of the proofs while maintaining safety at the same time. As the register sets are disjoint, type integrity of scalar values is enforced syntactically.

Some high-level types, such as enumerations, can be implemented as integers at the VM level. Constraints on these types (e.g., no arithmetic allowed on the corresponding VM integer) cannot be enforced by the VM. Instead, such constraints must be enforced at the proof-carrying code level.

Being a register architecture, our VM allows to apply more ahead-of-time optimizations than stack-based code representations like Java bytecode [10] or IL [8]. For example, constant folding, common subexpression elimination and copy propagation are all performed ahead of time in our architecture.

The architecture of our VM aims to ease the task of providing proof of safety for memory access operations by offering an instruction set which guarantees memory integrity. There is a single instruction to allocate memory:

$$p_j = new(size_v, size_p, i_k)$$

new allocates an array of objects of length i_k. (Single objects are arrays with a single element.) Allocated objects are divided into two sections: one section for values and one section for pointers. $size_v$ (resp. $size_p$) is the size of the values (resp. pointers) section. The VM guarantees that pointers and values are not intermixed. The pair $(size_v, size_p)$ is also referred to as the *characterizing tuple* or *ctuple*.

To guarantee safe access to an object's contents, all memory accesses via a pointer must use the same *ctuple* that was used to allocate the object. This must be enforced by the proof-carrying code layer. This mechanism can be understood as a rudimentary static type checking.

Separate instructions are provided to access values and pointers stored inside objects. There are read and write operations for the value register types:

$$\begin{gathered} i_j = iload(size_v, size_p, p_k, offset) \\ istore(size_v, size_p, p_k, offset, i_l) \\ b_j = bload(size_v, size_p, p_k, offset) \\ bstore(size_v, size_p, p_k, offset, b_l) \end{gathered}$$

Pointers are read and written using the pointer accesss instructions:

$$\begin{gathered} p_j = pload(size_v, size_p, p_k, offset) \\ pstore(size_v, size_p, p_k, offset, p_l) \end{gathered}$$

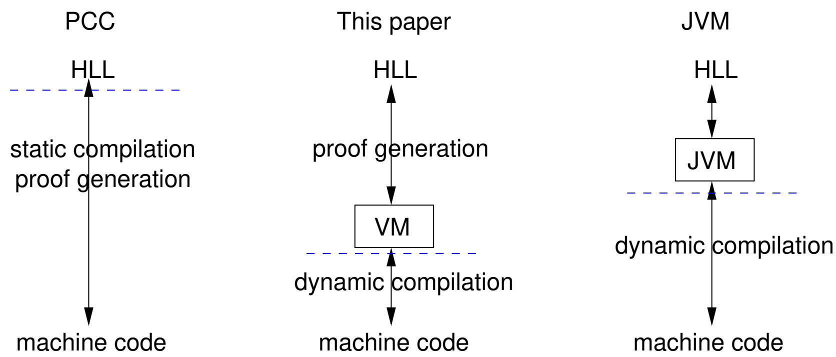


Figure 1: Everything above the dashed line is machine independent. Compared to current PCC implementations, our framework requires shorter proofs and is machine independent. Compared to current VMs, dynamic compilation is simpler because the semantic distance to actual target machines is smaller. Also, we are able to perform more optimizations ahead of time.

The verification layer has to ensure that the base pointer p_k was allocated with exactly the same allocation layout (expressed by $size_v$ and $size_p$), as specified for the load or store operation, and that the *offset* does not exceed $size_v$ or $size_p$ respectively. The proof-carrying code layer also has to ensure that pointer registers are always defined before their use.

Memory integrity can be implemented with little runtime overhead, while dynamically guaranteed type-safety is usually much more expensive in terms of runtime cost.

Pointer registers always point to the beginning of arrays. To access other array members, there is an instruction taking a pointer p_k and an array index i_l and returning an address:

$$a_j = adda(size_v, size_p, p_k, i_l)$$

As for all memory access operations, proper memory layout has to be provided and ensured by the PCC layer. The *adda* instruction does not perform a mandatory array bounds check. If the proof-carrying code layer cannot provide a static proof of the index being within array bounds, a dynamic guard has to be inserted using the *checklen* instruction.

$$checklen(p_k, i_l)$$

checklen raises an exception if i_l exceeds the length of the array referred to by p_k. To implement this, the VM stores the array length of every allocated memory block in a memory block header. The *getlen* instruction loads the length of an array into a register.

$$i_l = getlen(p_k)$$

Address registers may point inside an allocated memory block, unlike pointer registers, which point only to the start. This has implications for garbage collection. Consider the following example:

$$\begin{array}{c} 1\text{: } p_0 = new(size_v, size_p, i_{length}) \\ 2\text{: } a_0 = adda(size_v, size_p, i_{offset}) \\ 3\text{: } p_0 = null \end{array}$$

An array of object is allocated and a reference to the array is assigned into the pointer register p_0. *adda* is used to gain access to a member of the allocated array. A garbage collection cycle could be triggered after line 3, when the original reference to the array (in p_0) is not live but a_0 is. a_0 is a *derived pointer* that keeps the array reachable [5]. If an accurate garbage collector is used, it either must be able to mark objects as reachable starting from derived pointers, or we must make sure that base pointers are considered part of the root set where the derived pointers are still live.

As for pointers, the proof-carrying code layer has to guarantee that address registers are defined before their use.

The regular memory access operations do not accept an address register as base address. Instead, the specialized address access instructions have to be used:

$$\begin{array}{c} i_j = iloada(size_v, size_p, a_k, offset) \\ istorea(size_v, size_p, a_k, offset, i_l) \\ b_j = bloada(size_v, size_p, a_k, offset) \\ bstorea(size_v, size_p, a_k, offset, b_l) \\ a_j = ploada(size_v, size_p, a_k, offset) \\ pstorea(size_v, size_p, a_k, offset, p_l) \end{array}$$

The rationale of this split is that the regular pointer-based memory access operations have to take the memory block header into account when calculating the target address while address access operations do not. Note that the address generation instruction *adda* cannot be applied to address registers, but only to pointer registers.

The VM offers, in addition to *checklen*, two more forms of dynamic guards which can be placed into the instruction stream if no static proof of certain properties can be provided. The PCC layer can treat the existence of these instructions in the instruction stream as proof that the associated condition will be true at runtime (or an exception will be raised).

checknotnull raises an exception if a pointer register contains the value *null*.

$$checknotnull(p_i) \quad \rightarrow_? exception$$

checklayout verifies that a pointer points to a memory block with the specified memory layout and raises an exception otherwise.

$$checklayout(p_i, size_v, size_p) \quad \rightarrow_? exception$$

To support the *checklayout* instruction, the VM stores the *ctuple* of every allocated memory block along with the array length in the memory block header. As the location of the memory block header is unknown for addresses in address registers, *checklayout* only takes pointer registers as input. Instead, *checklayout* has to be applied to the base pointer previously used to generate the address in question.

6. PROOFS FOR OUR VM

Our VM provides the following guarantees: memory safety, and type-safety for the primitive types of int, floats and bools.

Given this, the proof burden of a code producer is greatly reduced. Since memory safety and primitive type safety are taken care of by the virtual machine, only proofs of type safety for non-primitive types are needed.

Consider, for example, the code for a simple factorial procedure, and the VM code for it.

```
procedure fact (n : integer) : integer
begin
        f : integer;
        f := 1;
        while (n > 0) do
                f := f * n;
                n := n - 1;
        end;
        fact := f;
end
```

```
iconst 1, i3
imov i3, i2
loophead :
iconst 0, i4
bls i4, i1, b0
brfalse b0, loopend
imul i2, i1, i5
imov i5, i2
iconst 1, i6
isub i1, i6, i7
imov i7, i1
goto loophead
loopend :
imov i2, i0
```

Note that since this procedure only uses primitive types, *it is type-safe by construction.* The instruction set does not allow any type-unsafe operations (such as assigning integers to booleans). This further reduces the proof burden of the code producer, since type-safety proofs need only be produced for non-primitive types such as pointers, arrays and records. Since substantial fractions of even object-oriented programs manipulate primitive types, we expect smaller proofs.

This is in sharp contrast to the Java bytecode instruction set. Java bytecode instructions do indicate the type of the operand being used (for example, `iload` for loading an integer, `fload` for loading a float, etc.). However, bytecode verification, as well as the technique of using stackmaps[14], must prove type-safety for bytecodes operating on values of primitive types as well. This is because of the Java virtual machine's stack-based memory model. The stack is a typeless entity, and once a datum is pushed onto the stack, information about its type is lost, and must be inferred again at the point that data is read from the stack. Thus, every load from the stack has to be proven type-safe, even for primitive types.

We measured the fraction of Java bytecodes that operate on primitive types (ints, floats, doubles and longs) in section 3 (large scale applications) of the JavaGrande benchmark. Between 5% and 56% (with an average of 24%) of all bytecodes were of this type. For specJVM98, the range was between 22% and 43%, with an average of 29%. This is a crude measure of how much proof burden our virtual machine saves right away compared to the Java virtual machine.

So proofs are only needed for pointers and records. For every instruction that manipulates an address, we need to make sure that the resulting pointer

1. Points the beginning of an array, or record, or value (e.g. we must not have pointers that point into the middle of an int)

2. Points to an object of the correct type (e.g. a pointer to an integer must not be allowed to point to a boolean)

For field accesses (using the `adda` instruction, condition 1 can be checked because offsets are known at compile time.

For condition 2 and all pointers, our type-safety proofs take the form of *typemaps.* A typemap is simply a mapping from pointers to their types. At procedure entry, these are simply the declared types of the formal parameters and local variables. During code generation, these are emitted for the beginning of every basic block.

Before letting the VM execute code, our type-checker makes sure the code is type-safe. Given that the code has been annotated with typemaps as explained above, the checker does the following:

- At beginning of a basic block, set derived typemap to annotated typemap

- Iterate through each instruction in the basic block, simulating its effect on the derived typemap

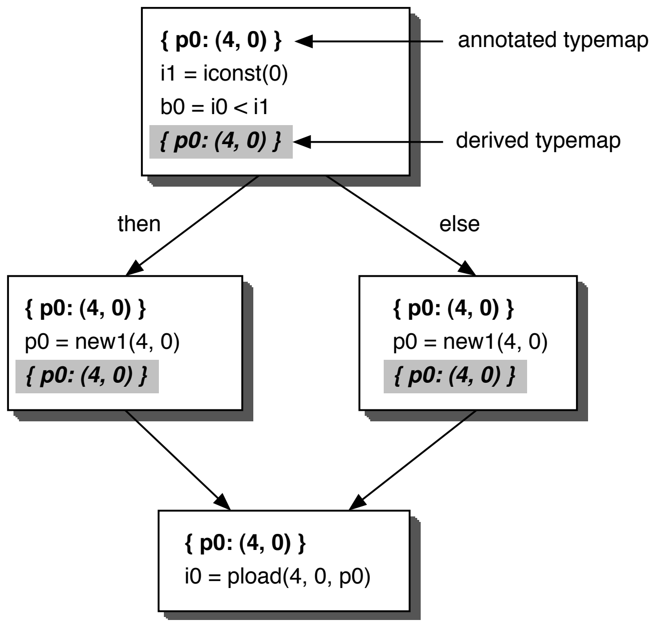


Figure 2: Simple example illustrating the use of typemaps for proofs of type safety. In this example, we are using the instruction "new1" as a shorthand for "new(*, *, 1)" to achieve syntactic similarity with the typemap.

- At the end of the block, for every successor B1 of this block, *match* the derived typemap with the annotated typemap at the beginning of B1. This matching procedure simply checks that every type in the successors typemap is more general than the corresponding type in the derived typemap.

Note that this takes one linear pass over the code.

For a very simple example consider the following code snippet.

```
p: pointer int
i: int

if (i > 0) then
        p = new(1, sizeof(int));
else
        p = new(1, sizeof(int));
end;
```

Figure 2 shows the corresponding VM code with annotated and derived typemaps. The type `int` translates to a *characterizing tuple* of $(4, 0)$ in the VM, since it has 4 bytes in the data section and none in the pointer section. The derived typemap at the end of a block is compared with the annotated typemap at the beginning of all its successors. In this case, the two are the same, and thus, checking succeeds.

7. RELATED WORK

There are several virtual machine(VM) designs that use a low level instruction set similar the one we are proposing. Keeping instruction sets close to real machine not only makes translation to real machine code fast and efficient but also makes it an attractive compilation target. However the primary focus of most of these VMs is code optimizations rather than safe code.

The LLVM project [9] proposes to optimize the program not only at compile time but also during link and run time. To achieve this they use a strongly typed SSA based intermediate representation(IR). Being SSA based and having type information makes optimizations fast and efficient. However, it supports some type unsafe cast operations for unsafe languages and hence does not provide any type safety. The project focuses only on the IR and does not have any runtime available.

Tao's Intent [15], a commercial system, focuses on binary portability mainly of multimedia/games application. The VM at the heart of the system is very simple and low level. Even though it aims at binary portability it does not provide any type safety. This is partly because they wanted to support legacy code written in C/C++ and partly for the reasons of performance, both being important factors for multimedia applications.

The Dis virtual machine [16] provides an execution environment for the Inferno system. It is a CISC like architecture with support for some high level data structures like list and strings and operators to manipulate them. It allows for type unsafe operations and hence does not provide any type safety guarantees.

The Omniware system [1]was designed to provide an open system for producing and executing mobile code. The system was designed so that it is language and processor architecture neutral. It has an instruction set based on RISC architecture with several CISC like functionalities.It uses Software Fault Isolation (SFI) to provide module level memory safety. It does not provide any type safety and supports unsafe languages.Since the VM is very close to the real architecture it is able to achieve near to native speeds.

Typed Assembly Language(TAL)[6] is another framework for verifying the safety of a program for a low level representation. TAL uses the type system of the source language to prove the safety of the program. It achieves this by annotating the assembly code generated with high level type information available in the source code. These annotations are easily verifiable and are used as proofs of safety. Before translation of the assembly code into a binary executable, the code is quickly verified for safety using these annotations. An important point to note here is that there is no trust relationship between the compiler and the proof checker. The proof checker does not trust the compilation process or the annotations. At the time of compilation it will quickly check for the correctness of the annotations.

Unlike PCC, which can use any safety policy expressible in first order logic, TAL only uses the typing rules of the programming language to express safety. This loss in generality of safety policies however leads to simpler, more compact and easy to generate proofs. It is not obvious how to automate the proof generation for policies more complex than memory and type safety.

Our system is in ways close to TAL as it also only supports type safety proofs. Our VM instruction set is strongly typed and has primitive types like int and boolean. Type safety for higher order types has to be proved using verifiable proofs. Since our VM provides a higher level of abstraction than a machine assembly code and also provides some memory safety guarantees we claim that the proofs will be shorter and even faster to verify. In case of segments of programs that use only primitive types the proofs are implicit in the instruction set and doesn't need any additional proofs.

SafeTSA[3] is a type-safe intermediate representation based on Static Single Assignment form (SSA). SafeTSA solves the problem of making SSA easily verifiable so that it can be used as a safe software transportation format. It does so by a combination of *type separation* and by introducing a *referentially safe* naming scheme for SSA values. As a consequence, the type and reference-safety of SafeTSA can be verified in linear time. Like the Java Virtual Machine itself, SafeTSA is tightly coupled to the Java type system and does not easily support languages with highly different type systems. SafeTSA is semantically further removed from the machine layer than the VM we have described in this paper, and hence requires a more substantial dynamic translation machinery at the target machine.

8. CONCLUSION AND OUTLOOK

We are exploring the design space of hybrid solutions between virtual machines and proof-carrying code. Our goal is to find the "sweet spot" that reconciles high execution performance and just-in-time compilation speed on one hand, and small and efficient type-safety proofs on the other hand. In this paper, we have reported on the design of our first solution to populate this design space; undoubtedly, there will be further candidates to follow. Preliminary evidence suggests that hybrid solutions have advantages over single-mode VM or PCC approaches.

9. ACKNOWLEDGEMENTS

Parts of this effort are sponsored by the National Science Foundation under grants CCR-TC-0209163, by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-99-1-0536, and by the Office of Naval Research under grant N00014-01-1-0854. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and should not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the National Science foundation (NSF), or any other agency of the U.S. Government.

10. REFERENCES

[1] S. Lucco A. Adl-Tabatabai, G. Langdale and R. Wahbe. Efficient and language independent mobile programs. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and ImplementationProgramming Language Design and Implementation*, pages 128–136, May 1996.

[2] Wolfram Amme, Niall Dalton, Peter H. Frohlich, Vivek Haldar, Peter S. Housel, Jeffrey von Ronn, Christian H. Stork, Sergiy Zhenochin, and Michael Franz. Project transprose : Reconciling mobile-code security with execution efficiency. In *DARPA Information Survivability Conference and Exposition*, June 2001.

[3] Wolfram Amme, Niall Dalton, Jeffery von Ronne, and Michael Franz. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 137–147, Snowbird, Utah, June 20–22, 2001. *SIGPLAN Notices,* 36(5), May 2001.

[4] A. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 247–258, Washington - Brussels - Tokyo, 2001. IEEE.

[5] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*, pages 273–282, 17–19 June 1992. *SIGPLAN Notices* 27(7), July 1992.

[6] Karl Crary Greg Morrisett, David Walker and Neal Glew. From system f to typed assembly language. In *"25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages"*, pages 85–97, San Diego, CA, USA, "January" 1998.

[7] Vivek Haldar, Christian H. Stork, and Michael Franz. The source is the proof. In *New Security Paradigms Workshop*, Sep 2002.

[8] ISO/IEC 23271. Common Language Infrastructure (CLI), Partition III, CIL Instruction Set, Dec 2002.

[9] Chris Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, University of Illinois, Urbana Champaign, Urbana, Illinois, 2000.

[10] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison Wesley Longman, Inc., second edition, 1999.

[11] George C. Necula. Proof-carrying code. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 15–17, 1997.

[12] George C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *Conference Record of POPL'01: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–154, London, United Kingdom, January 17–19, 2001. *SIGPLAN Notices,* 36(3), March 2001.

[13] Robert F. Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: definition, verification, validation*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2001.

[14] Sun Microsystems Inc. Connected, Limited Device Configuration, Apr 2000.

[15] Introduction To Intent Technology. Available at `http://withintent.biz/aux-files/Introduction_to_intent_Technology.pdf`, 2002.

[16] Phil Winterbottom and Robert Pike. The Design of the Inferno virtual machine. *Hot Chips 9 symposium*, August 1997.

Generation of Fast Interpreters for Huffman Compressed Bytecode

Mario Latendresse
Northrop Grumman IT/
Technology Advancement Group
FNMOC/U.S. Navy
7 Grace Hopper, Monterey, CA, USA
mario.latendresse.ca@metnet.navy.mil

Marc Feeley
Département d'informatique et
recherche opérationnelle
Université de Montréal
C.P. 6128, succ. centre-ville
Montréal, H3C 3J7, Canada
feeley@IRO.UMontreal.CA

ABSTRACT

Embedded systems often have severe memory constraints requiring careful encoding of programs. For example, smart cards have on the order of 1K of RAM, 16K of non-volatile memory, and 24K of ROM. A virtual machine can be an effective approach to obtain compact programs but instructions are commonly encoded using one byte for the opcode and multiple bytes for the operands, which can be wasteful and thus limit the size of programs runnable on embedded systems. Our approach uses canonical Huffman codes to generate compact opcodes with custom-sized operand fields and with a virtual machine that directly executes this compact code. We present techniques to automatically generate the new instruction formats and the decoder. In effect, this automatically creates both an instruction set for a customized virtual machine and an implementation of that machine. We demonstrate that, **without** prior decompression, fast decoding of these virtual compressed instructions is feasible. Through experiments on Scheme and Java, we demonstrate the speed of these decoders. Java benchmarks show an average execution slowdown of 9%. Compression factors highly depend on the original bytecode and the training sample, but typically vary from 30% to 60%.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.4 [**Programming Languages**]: Processors—*Interpreters*

Keywords

Code compression, canonical Huffman code, decoder, Java

1. INTRODUCTION

Embedded systems are resource-constrained devices requiring careful attention to memory usage and power consumption. To attain these goals, several researchers are taking the approach of reducing program size [5, 13, 12].

We focus on the context where code decompression cannot be performed prior to the program's execution. This constraint is reasonable for embedded systems where a bulk decompression of programs, or even parts of programs, before execution, might exceed the available RAM.

Some researchers [9, 10, 3] have stated the possibility of using Huffman codes to compress bytecode, usually to conclude that this would, at the software level, increase decoding time to an unacceptable level or need too much space for table look-up.

Reducing space taken by operands is also important since they usually account for a large part of the code size. Instruction formats with small operand fields can further reduce size.

Our work shows that using canonical Huffman code for opcodes, new customized instruction formats, replacement of sequences of repetitive instructions by one opcode and no byte boundary alignment can significantly reduce bytecode size and still allow fast direct execution by an interpreter.

The primary focus of this paper is to show that there are techniques to efficiently decode such compressed instructions.

For speed, canonical Huffman codes should not be decoded bit by bit; instead, blocks of k bits should be used. Such an idea has been explored previously by Turpin and Moffat [19]. We have extended their work to allow multiple k bits look-ups and generate decoders given a space constraint.

In the next section, we give a general presentation of the compression algorithm. In section 3 canonical Huffman codes are presented along with a compact but slow decoding method. Section 4 presents much faster but slightly less compact decoders. Section 5 explains the C code's structure for all canonical decoders. Section 6 discusses how decoders access memory for opcodes and operands. Experimental results showing that the approach is practical are presented in section 7. Section 8 presents some of the related work.

2. THE COMPRESSION ALGORITHM

Figure 1 presents our general framework. The sample of programs is bytecode encoded with an unmodified compiler. An instruction set encoding to compactly represent the sample is then generated by a tool. This requires an analysis

IVME'03, June 12, 2003, San Diego, California, USA.

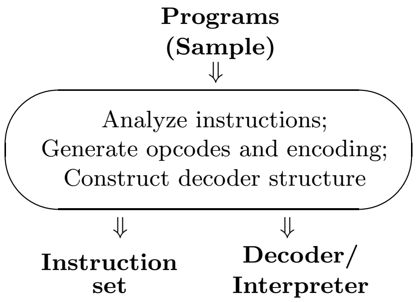


Figure 1: Creation of instruction set, its decoder and interpreter.

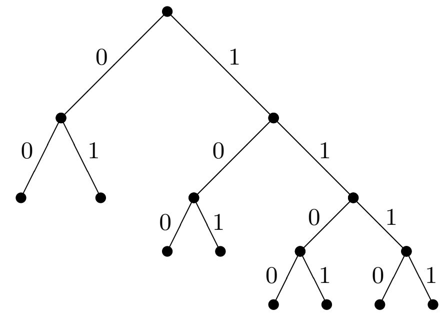


Figure 2: A canonical ascending Huffman tree.

of the instruction frequencies, the length of operands, etc. of the sample. The decoder is generated given a space constraint parameter, along with the interpreter. The sizes of the decoder and interpreter are taken into account to reduce program sizes. This approach is transparent for the compiler writer since the compression of programs can be done from the original bytecode.

Our compression approach creates new instructions and an encoding for them. These instructions are either macro-instructions to replace a sequence of instructions, or a basic instruction with a new format for the operands. We proceed as follows to create them. From the sample of programs:

1. A dictionary of (possibly overlapping) repetitive sequences is built. We limit their number by their length and a minimum of their occurrences in the sample.

2. A dictionary of formats to encode all basic instructions using as few bits as possible is created. It includes the original formats of the virtual machine to be able to encode all possible programs.

3. A greedy algorithm repetitively selects either a new format or a sequence of instructions, based on the maximum space saving, until no space gain can be obtained.

The greedy algorithm takes into account the opcode lengths, the new formats, and the space of the decoder. Further details on the selection algorithm can be found in [15].

Henceforth, the following setting is used: the opcodes are variable length canonical Huffman codes generated using the static frequencies of the opcodes from a sample of programs; and operands are uncompressed but of a length that is not restricted to a multiple of eight bits. Thus, opcodes and operands are not byte-aligned. The deplacements of branching instructions are in bits, but instructions following subroutine calls are byte-aligned—return addresses are in bytes.

3. HUFFMAN ENCODING OF OPCODES

We encode opcodes using canonical Huffman codes [24]. These are similar to Huffman codes built by the original bottom up method of [11], but the numerical values of the codes of a given length form a consecutive sequence. As will be shown, they have a very compact representation of the bijection between the codes and the encoded object.

Figure 2 shows such a canonical tree, where branches are pushed to the right: it is an ascending tree, since it is possible to order the codes in increasing length and numerical value.

Let l_c be the length in bits of code c, $v(c)$ its value, $k \geq l_c$ a constant, and $V^k(c) = v(c)2^{k-l_c}$; in other words, $V^k(c)$ is the value of c left justified in a k bits processor register. Left justification allows the creation of a very compact decoder as presented in Section 3.1.

Let $C = \{c_i\}$ be a set of canonical Huffman codes, $l_{\max}$ their maximum length and w a constant such that $w \geq l_{\max}$. Define the vector $base^w[1 \ldots l_{\max}]$ such that $base^w[j]$ is the smallest value $V^w(c)$ for all codes c such that $l_c = j$. Define the vector $disp[1 \ldots l_{\max}]$ such that $disp[j]$ is the number of codes c for which $l_c < j$. The index of code c of length l_c is:

$$\frac{V^w(c) - base^w[l_c]}{2^{w-l_c}} + disp[l_c] \qquad (1)$$

If the length of c is known, its index is given by that equation. Given the index, a computed branch would jump to the implementation of the virtual instruction.

To show examples of opcode frequencies, independently of a specific instruction set and samples, assume the n probabilities p_i of a special case of Zipf's law: $p_i = 1/(iH_n)$, $1 \leq i \leq n$, where H_n is the nth harmonic number $\sum_{j=1}^{n}(1/j)$. Such probabilities model well the static frequency of instructions in programs. Table 1 presents vectors $base^w$ and $disp$ for the Zipf-200 opcodes partly listed in table 2. Their average length[1] is 6.0267.

3.1 Very Compact but Slow Decoding

Assume that the beginning of an instruction is left justified in a variable `rd`. According to equation 1, decoding the opcode can be reduced to finding its length which can be done by a sequential search in *base*. Figure 3 shows a fragment of C code for this slow but very compact decoder: Line 2 does the sequential search; the index of the code is calculated in `crd` by line 3 using 1; line 4 removes the opcode; line 5 does the actual branching to the virtual instruction implementation (using `gcc`'s computed `goto`).

This is a very compact decoder since its code is small and the vectors `base_w` and `disp` only contain $l_{\max}$ elements each. For Zipf-200 on a 32 bit processor, $l_{\max} = 10$ and $w = 32$, so the two vectors use a total of 80 bytes. Even for Zipf-400, that is 400 opcodes, a mere eight more bytes are needed.

But in general, this search is way too slow. The next section shows a better approach flexible in space and in speed.

[1]The average length is $\sum_{1 \leq i \leq n} l_{c_i} p_i$ where the opcode for the probability p_i is c_i and its length is l_{c_i}.

```
i = l_max;
while (rd < base_w[i]) i--;
crd = (rd-base_w[i] >> w-i) + disp[i];
rd <<= i;
goto *adr[crd];
```

Figure 3: C code for a very compact, but slow, decoder for canonical ascending Huffman codes.

i	$disp$	$base^w$
3	1	$000{\cdot}2^{w-3}$
4	2	$0010{\cdot}2^{w-4}$
5	5	$01010{\cdot}2^{w-5}$
6	9	$011100{\cdot}2^{w-6}$
7	16	$1000110{\cdot}2^{w-7}$
8	33	$10101110{\cdot}2^{w-8}$
9	65	$110011100{\cdot}2^{w-9}$
10	135	$1110111110{\cdot}2^{w-10}$

Table 1: The vectors `base_w` (aka $base^w$) and `disp` ($disp$) for Zipf-200.

4. FAST DECODING

To increase speed, the linear search for the length of the opcode must be eliminated. This is done by a table look-up using the leftmost k bits of `rd`. The table contains branching addresses at which either decoding continue or the virtual instruction is emulated.

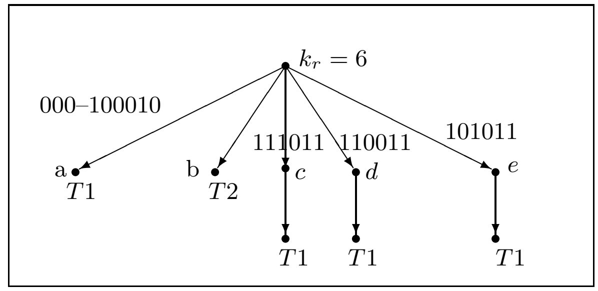
For the table look-up on k bits, three situations can arise:

1. The opcode is recognized.
2. The opcode is not recognized but its length is known.
3. The opcode is not recognized and its length is unknown.

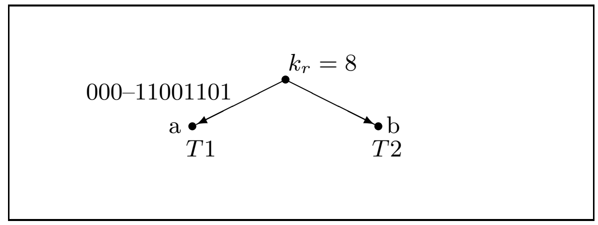
Case 1 is ideal, which occurs for all opcodes c where $l_c \leq k$. A direct jump is done to the implementation of the virtual instruction. In case 2, the length of the opcode is used to compute its index by equation 1; then a jump to the implementation of the virtual instruction is done. In case 3, the next bits are used to continue decoding using another look-up. Thus, the decoder has a tree structure where each interior node is case 3, simply called type 3 nodes. In case 1 and 2 we have leaf nodes, simply called type 1 and 2 nodes. Note that each type 3 node requires a vector of addresses of its own, whereas type 2 nodes share the same vector.

In general, interior nodes do not use the same number of k bits to do a table look-up. For a node ν of type 3, k_ν is the number of bits used to do the table look-up. In particular, k_r denotes the number of bits used by the root r of a decoder.

Each node requires some time to execute. The time spent in a node of type i is denoted t_i. Note that $t_1 = 0$ because no further decoding is needed for type 1 nodes. These timing values do not have to correspond to any real unit of time, but simply be relative to a known base value. For example, they could be approximated by the number of host processor cycles used at each node.



Tree D_1, $S(D_1) = 563$, $T(D_1) = 15.93$



Tree D_2, $S(D_2) = 1084$, $T(D_2) = 13.8$

Figure 4: Two decoder trees D_1 and D_2 for Zipf-200, generated using the parameters $s_a = 4$, $s_2 = 30$, $s_3 = 25$, $t_2 = 10$, $t_3 = 7$. We have $k_c = 4$, $k_d = 3$ and $k_e = 2$.

To evaluate the space taken by the decoder, three constants are used: s_a is the number of bytes of an address (e.g. 4); s_2 is the number of bytes used by the machine code implementing a type 2 node and s_3 is for a type 3 node. We therefore take into account the space for look-up tables and the code to implement the decoding.

Figure 4 presents two decoder trees D_1 and D_2 for Zipf-200. Decoder D_1 does, at the root, a table look-up using 6 bits, and has three internal nodes doing table look-ups using 4, 3 and 2 bits; whereas decoder D_2 does, at the root, a table look-up using 8 bits and has one type 2 node. Note that there are opcodes of up to 10 bits, but no table look-up is done using that many bits. The total space for decoder D_1 is 563 bytes and for D_2 it is 1084 bytes. The average decoding time for D_1 is 15.93 and for D_2 it is 13.8.

In table 2 each opcode is shown along with the final node of decoding by the two decoders and corresponding relative time.

Given a space constraint, the basic parameters s_i and t_i, and the (static or dynamic) frequencies of the opcodes, we generate the fastest decoder. A branch and bound algorithm to do so is presented in [16]. It searches from the fastest to the slowest decoders and when the space constraints are met, it stops. (For all our experiments, it takes a few seconds to find the fastest decoder.)

To construct the decoder structure, the algorithm is general enough to accept static or dynamic (run-time) opcode frequencies. Dynamic frequencies are harder to obtain as they not only depend on the program samples but also on the input data of those programs. It is up to the designer of the virtual machine to assess the accurancy and relevance of dynamic frequencies and use them when they greatly differ from the static ones.

i	opcode	Tree D_1		Tree D_2	
		ν	Time	ν	Time
1	000	a	7	a	7
2	0010	a	7	a	7
3	0011	a	7	a	7
4	0100	a	7	a	7
...	...				
15	100010	a	7	a	7
16	1000110	b	17	a	7
17	1000111	b	17	a	7
...	...				
31	1010101	b	17	a	7
32	1010110	e	14	a	7
33	10101110	e	14	a	7
34	10101111	e	14	a	7
35	10110000	b	17	a	7
...	...				
62	11001011	b	17	a	7
63	11001100	d	14	a	7
64	11001101	d	14	a	7
65	110011100	d	14	b	17
66	110011101	d	14	b	17
68	110011111	d	14	b	17
69	110100000	b	17	b	17
...	...				
124	111010111	b	17	b	17
125	111011000	c	14	b	17
126	111011001	c	14	b	17
...	...				
135	1110111110	c	14	b	17
136	1110111111	c	14	b	17
137	1111000000	b	17	b	17
138	1111000001	b	17	b	17
...	...				
199	1111111110	b	17	b	17
200	1111111111	b	17	b	17

Table 2: Zipf-200 and timing for two decoders.

L_decode:
1 {Transfer bytes from program to `rd` such that it has at least $l_{\max}$ bits, and increase `nb_rd` accordingly. }
2 `crd = rd >>` $w - k_r$`;`
3 `goto *adr_[crd];`
L_i `: /* opcodes of length` i `(type 2) */`
4 `crd = rd >>` $w - i$`;`
5 `goto *adr_inst[crd -` $base(C^{t2})_i + disp(C^{t2})_i$`];`
Lp_*prefix*: `/* sub-decoder (type 3) */`
6 `crd = rd >>` $w - l_{prefix} - k_{prefix}$`;`
7 `goto *adr_`*prefix*`[crd-`$v(prefix)2^{k_{prefix}}$`];`
I*mne*`:/* C code for` *mne* `(type 1) */`
8 { If *mne* has parameters, transfer them to p_i}
`/* eliminate opcode and parameters */`
9 `rd <<=` $l_{\texttt{opcode}}$+$l_{\texttt{parm}}$`;`
10 `nb_rd -=` $l_{\texttt{opcode}}$+$l_{\texttt{parm}}$`;`
11 { C code to emulate *mne* }
12 `goto L_decode;`

Figure 5: General C code of canonical decoders.

```
#define BYTE(i) (unsigned int)prgm[pc+i]

rd    |= (BYTE(0) << 24 | BYTE(1)  << 16
         | BYTE(2)<< 8  | BYTE(3)) >> nb_rd;
pc    += (32-nb_rd) >> 3;
nb_rd += (32-nb_rd) & ~7;
```

Figure 6: A simple technique for line 1 of Figure 5.

5. THE DECODER C CODE

Figure 5 shows the general structure of the C code for canonical decoders. Decoding begins at label `L_decode`. There is a label `L_`i for each case where more than one opcode of length i is not directly recognized by a node of type 3. These are type 2 nodes. There is a label `Lp_`*prefix* for each node of type 3, where *prefix* corresponds to the prefix of all codes for that node. For each virtual instruction *mne* the label `I`*mne* is the entry point of its implementation.

Line 1 loads, if necessary, some additional bytes in `rd`. The exact C code for this depends on the form of memory access used as discussed in Section 6. The incoming bits are justified in the high part of `rd` and `nb_rd` is adjusted to contain the number of bits in it. It always loads a multiple of eight bits, since the program counter points to a byte in memory, but `rd` does not necessarily contain a multiple of eight valid bits. Figure 6 presents a simple and inefficient portable implementation for line 1, for $w = 32$. Section 6 presents better portable techniques.

Line 2 is the root of a decoder where the first look-up is done; line 3 jumps to a type 2 or 3 node, or to the emulation of a virtual instruction. $w - k_r$ is a constant. At line 5, the term $base(C^{t2})_i + disp(C^{t2})_i$ is a constant: $base(C^{t2})_i$ is the ith value of $base^w/2^{w-i}$ but where $base^w$ is defined using only the codes C^{t2}, that is all codes treated by type 2 nodes. Using this subset of C might very well decrease the length of vector `adr_inst`. To be more precise, all addresses of virtual instructions in `adr_` are not duplicated in `adr_inst`. They also do not appear in any vectors `adr_`*prefix* for type 3 nodes. The vector $disp(C^{t2})$ is the corresponding vector of $base(C^{t2})$. Line 5 necessarily jumps to a virtual instruction. In line 6, the term $w - l_{prefix} - k_{prefix}$ is a constant, l_{prefix} being the length of the prefix and k_{prefix} the number of bits decoded by this node. So the shifting `rd >>` $w - l_{prefix} - k_{prefix}$ leaves in `crd` not only the k_{prefix} bits to decode but also the previous l_{prefix} bits. Line 7 applies the proper adjustment using the term $v(prefix)2^{k_{prefix}}$, which is the extra value left in `rd` before this node. This avoids shifting some bits out of `rd` until the end of decoding.

At line 8, decoding is complete and this is the emulation of the virtual instruction *mne*. If *mne* has some parameters, they are obtained here. This may use up all bits in `rd` or just part of them; it may also access memory. In most cases, bits should transit through `rd`. What lines 9 and 10 say, which is done differently depending on memory access forms (see Section 6), is that `rd` should contain the following bits and `nb_rd` should be maintained accordingly.

Finally, line 12 returns to the beginning of the decoding cycle. Again, this depends on the form of memory access as presented in section 6. It could return to a point in the block of line 1 where it loads a specific number of bytes according to the number of bits consumed by *mne*.

6. PREFETCHING OF CODE

One important part of the decoder C code was left unspecified, namely line 1, which loads bytes from memory into `rd`. We investigated several portable ways, three of which are reported in this section.

Getting opcodes and operands from memory into `rd` can be time consuming since multiple byte loads and bit manipulation operations are possibly needed. We have explored three different techniques to access memory. The first one, form-a, is simple, but shows major slowdowns on many benchmarks. The other two, form-b and form-c, show competitive speed; form-c being often faster than form-b but using more space. Our algorithm to generate decoders provides the option of using one of these three forms. Benchmarks in section 7 show their relative merits. For all forms, enough bits are in `rd`, at the root of the decoder, to decode one opcode without accessing memory.

6.1 Simple form (Form-a)

This version uses the number of bits in `rd` to load the minimum number of bytes necessary to maintain between $w-7$ and w bits in `rd` at line 2. This can simply be done using a case analysis based on the value of `nb_rd`, reading from memory the required bytes, shifting them to the left, and merging them to `rd`. The number of bytes to read is $\lfloor (w - \texttt{nb\_rd})/8 \rfloor$ and the number of bits to shift is $(w - \texttt{nb\_rd}) \bmod 8$.

This technique loads in `rd` as many bytes as possible. The advantages is a reduced number of merging operations and faster access to operands since they are most often hauled in before decoding the opcode. The other advantage is a smaller interpreter, since it reduces the number of instructions accessing operands in memory. The disadvantage is a slow operation at every cycle to verify and load the correct number of bytes.

6.2 Several-roots form (Form-b)

In this form, as in the previous form-a, there are between $w-7$ and w bits in `rd` at the beginning of the decoding. But instead of one entry point with complex verification of the number of bytes to load, there are several entry points to the root of the decoder each one loading either x or $x+1$ bytes. The decision between case x and $x+1$ is faster than the general case of form-a.

It is faster, since each virtual instruction knows the number of bits extracted from `rd` (at lines 9 and 10), it knows approximately the number of bytes to load after its emulation. Indeed, suppose that a virtual instruction uses $b \le w-7$ bits, including its opcode. At the entry of its implementation there are between w and $w-7$ bits in `rd`, therefore there are, after its emulation, between $w-b$ and $w-b-7$ bits remaining in `rd`. So, there are between $\lceil (b-1)/8 \rceil$ and $1 + \lceil (b-1)/8 \rceil$ bytes to load in `rd`. If b is a constant, which is quite a common case in practice, it is possible to jump to the proper root r_x without any test. If b is not a constant, that is a variable length instruction, the virtual instruction implementation has to calculate the number of bits left in rd anyway. In the case where $b > w-7$, the virtual instruction itself has to load bytes from memory, thus also knows, after its emulation, the exact number of bytes to load. Note that no dynamic test is done to verify between the two cases, if b is a constant. It is hardcoded in the implementation of the interpreter.

In some way, the proper number of bytes to load falls back to each virtual instruction which simply branches to one of the roots that does one integer relational test between a constant and `nb_rd`. The disadvantage of this method is a slightly bigger decoder.

6.3 Conditional form (Form-c)

In this form, there is a verification of the number of bits in `rd` at the root of the decoder. Memory is accessed, at the root, if and only if `nb_rd` is under $l_{\max}$, the longest opcode. This ensures that the decoder does not access memory while decoding an opcode. If it is under $l_{\max}$, as many bytes from memory as possible are merged to `rd`. For example, if $l_{\max} = 14$, $w = 32$, and `nb_rd`$= 6$, three bytes are loaded and merged to `rd`. This means that access to memory is delayed as much as possible.

The advantage of this method is a reduced number of merging operations to `rd`. The disadvantage is a larger interpreter, since if the virtual instruction uses more than $l_{\max}$ bits, it is necessary to verify if there are enough bits in `rd` to access the operands. This case occurs less frequently in form-b for which there are $w-7$ bits in `rd` after decoding the opcode (assuming $l_{\max} < w-7$).

The disadvantage is that more virtual instruction implementations have to access memory for their operands.

7. EXPERIMENTAL RESULTS

In order to evaluate our approach, we applied it to the Java Virtual Machine (JVM) on ten benchmarks [1] and the entire JDK 1.0.2 library; to the Scheme language on seven benchmarks and the $\mathrm{R^4RS}$ library; and to six synthetic benchmarks to demonstrate the worst case scenarios.

For all benchmarks two processors are used: a 600MHz Pentium III and a 200MHz Sparc Ultra-1 with respectively 32KB and 1MB level 1 cache. All C programs were compiled using `gcc` version 2.8.1 for SunOS and version 2.91.66 for Linux with the same optimizing parameter, namely `-O3`.

7.1 Java benchmarks

We use the Java Virtual Machine to demonstrate our approach on a widely available bytecode using Harissa [20]. Most virtual instructions' implementation are unchanged but branching instructions must be modified to branch on non-byte boundaries. Harissa uses a C switch statement to decode bytecoded instructions. All cases of this switch are transformed into C macro-instructions and are used by the canonical decoder to implement each instruction in the JVM machine for compressed code. The switch is removed and replaced by a decoder automatically generated from our tool.

Table 3 presents the timing results and the compression factors of bytecodes for the BYTEmark Java benchmarks [1]. These are moderate size benchmarks suited to evaluate the speed of JVM implementations. The compression factor is the length of compressed code divided by the uncompressed code (bytecode). It takes into account the compression of opcodes, the compact operands, and the use of macro-instructions.

The training set is the `classes.zip` from JDK 1.0.2, containing over 400 class files. The resulting shortest opcode has three bits and the longest opcodes have twelve bits. Forty of the existing instructions were duplicated but with shorter parameter fields resulting in a 241 instructions JVM machine. This extension was done automatically by our tool to generate virtual instruction sets from a sample of programs [15]. The sole choices of macro-instructions and parameter lengths were done to better compress the classes and not for speed. All class files for the BYTEmark benchmarks, including all libraries in `classes.zip`, are compressed based

Benchmark	Absolute Time Uncompressed		Relative Time Compressed				Compression Factor of JVM Code	Size of JVM code in bytes
	Pentium	SPARC	Pentium		SPARC			
			$C_{k_r=7}$	$C_{k_r=10}$	$C_{k_r=7}$	$C_{k_r=10}$		
NumericSort	2.75	3.99	1.11	1.05	1.21	1.03	56.4%	773
StringSort	7.68	10.35	1.08	1.02	1.20	1.03	56.5%	1541
BitfieldOps	5.11	6.21	1.42	1.32	1.43	1.27	65.8%	833
FPemulation	3.82	5.29	1.25	1.17	1.31	1.15	67.0%	3724
Fourier	1.83	2.24	1.30	1.24	1.44	1.24	64.7%	640
Assignment	1.49	2.42	1.02	0.97	1.22	1.02	60.1%	1634
IDEAencryption	5.40	6.46	1.44	1.33	1.38	1.09	64.2%	1800
Huffman	2.50	3.98	1.11	1.09	1.23	1.09	60.7%	1395
NeuralNet	27.8	46.64	1.03	0.99	1.13	1.03	51.6%	7467
LUdecomposition	3.29	4.60	1.09	1.03	1.16	0.98	59.2%	1602
Average			**1.18**	**1.12**	**1.27**	**1.09**	**58.8%**	

Table 3: Relative speed and compression factors of Java benchmarks with modified Harissa JVM.

	Relative time Pentium					**Relative time** SPARC				
	$C_{k_r=6}$	$C_{k_r=7}$	$C_{k_r=8}$	$C_{k_r=9}$	$C_{k_r=10}$	$C_{k_r=6}$	$C_{k_r=7}$	$C_{k_r=8}$	$C_{k_r=9}$	$C_{k_r=10}$
fib	0.88	0.85	0.82	0.85	0.85	0.80	0.72	0.71	0.85	0.66
tak	1.34	1.30	1.28	1.29	1.29	1.07	1.00	1.00	1.11	0.88
earley	0.96	0.93	0.90	0.90	0.90	1.00	0.94	0.93	0.94	0.82
conform	1.12	1.10	1.08	1.06	1.01	1.08	1.06	1.01	0.97	0.95
mm	1.36	1.31	1.28	1.30	1.29	1.24	1.14	1.18	1.11	1.04
destruct	1.10	1.05	1.02	1.04	1.04	0.94	0.88	0.87	0.85	0.79
qsort	0.90	0.89	0.87	0.87	0.87	0.73	0.68	0.70	0.67	0.60

Table 4: Relative execution time of compressed Scheme **programs, using form-c on** Pentium **and** SPARC**.**

on the new Huffman opcodes, the new formats, and the macro-instructions. For `classes.zip`, a 60.9% compression factor is obtained and an overall average of 58.8% for the benchmarks.

We use memory access form-c with two decoders having the following structures: 1) $k_r = 7$, five nodes of type 2, namely L_{8-12}, and three nodes of type 3, all directly below the root; 2) $k_r = 10$, two nodes of type 2, namely L_{10-11}, and one node of type 3.

The worst speed results are the Fourier and Bitfieldops benchmarks. This is due to the frequent execution of instructions having long opcodes and small granularities. Some of them are floating-point virtual instructions, not statically frequent in `classes.zip`. They also do not access object fields as frequently as the other benchmarks. Since the **`getfield`** and **`putfield`** instructions have a moderate granularity, they increase execution time compared to decoding. On the other hand Assignment, StringSort, NeuralNet, NumericSort, and LUdecomposition show a small slowdown.

The benchmarks Assignment, StringSort, and NeuralNet have a large number of virtual method calls as well as field accesses. As mentioned, field accesses hide decoding overhead, and this is also true for method invocation, be it static or virtual. They show little slowdown for the SPARC with a good performance for the Pentium.

Half of the benchmarks have a 40% reduction in size with a negligible slowdown ($\leq 3\%$).

	Bytecode Size	Schemina **Factor**	gzip **Factor**
libScheme	32040	23%	16%
fib	169	18%	77%
tak	582	26%	37%
earley	26271	31%	19%
conform	28599	23%	17%
mm	2550	30%	29%
destruct	3371	22%	22%
qsort	5827	57%	45%

Figure 7: Compression factors for Scheme **programs.**

7.2 The Scheme language

Our approach has also been applied to the Scheme language[15]. From a general stack machine called Machina our tools create a new set of instructions called Schemina.

Table 7 compares the compression factors of our technique with gzip. We only used it to compare compression performances since gzip encoding cannot be executed without prior decompression. gzip can have better performances for two major reasons: it compresses disregarding basic block boundaries, and it disallows non sequential decompression.

In several cases our approach is close or better than gzip and we can still efficiently execute our compressed code.

Table 4 presents the relative execution time of the com-

Decoder	Machine$_1$			Machine$_2$			Machine$_3$		
Pentium									
	a	**b**	**c**	**a**	**b**	**c**	**a**	**b**	**c**
$k_r = 4, L_5, L_6$	1.81	1.76	1.52	2.19	1.64	1.48	1.38	1.07	0.96
$k_r = 5, L_6$	1.60	1.71	1.47	2.13	1.64	1.55	1.32	0.99	0.96
$k_r = 6$	1.60	1.58	1.34	2.08	1.49	1.42	1.31	0.95	0.90
SPARC									
	a	**b**	**c**	**a**	**b**	**c**	**a**	**b**	**c**
$k_r = 4, L_5, L_6$	2.77	1.61	1.52	2.04	1.60	1.39	1.09	0.99	0.97
$k_r = 5, L_6$	2.77	1.51	1.43	2.50	1.42	1.35	1.02	0.91	0.88
$k_r = 6$	2.39	1.63	1.21	2.12	1.23	1.18	0.93	0.81	0.78
Decoder	**MachineP$_1$**			**MachineP$_2$**			**MachineP$_3$**		
Pentium									
	a	**b**	**c**	**a**	**b**	**c**	**a**	**b**	**c**
$k_r = 4, L_5, L_6$	1.80	1.62	1.47	1.57	1.55	1.38	1.37	1.22	1.18
$k_r = 5, L_6$	1.67	1.62	1.44	1.57	1.53	1.40	1.34	1.20	1.14
$k_r = 6$	1.70	1.45	1.30	1.46	1.41	1.26	1.25	1.10	1.14
SPARC									
	a	**b**	**c**	**a**	**b**	**c**	**a**	**b**	**c**
$k_r = 4, L_5, L_6$	2.06	1.59	1.55	1.91	1.51	1.44	1.70	1.41	1.35
$k_r = 5, L_6$	2.20	1.42	1.37	1.76	1.37	1.34	1.54	1.30	1.25
$k_r = 6$	1.90	1.29	1.16	1.60	1.27	1.16	1.46	1.19	1.14

Table 5: Relative time to execute compressed programs, based on Zipf-20, for six virtual machines, three memory access forms, and on two processors.

pressed Scheme programs. For several benchmarks there are speedups since macro-instructions increase speed and many of the new instructions have short opcodes and operands.

7.3 Synthetic benchmarks

The Java and Scheme benchmarks demonstrate the applicability of the approach in a realistic setting. But it raises the question of hidden overhead by the emulation of the virtual instructions. Also, inlined macro-instructions increase speed. Therefore, we also present synthetic benchmark timings, where the frequency of instructions, their granularity, and their operand lengths are precisely defined; there are no macro-instructions used for these. In other words, the synthetic benchmarks more clearly show the overhead of Huffman decoding and non-byte alignment.

For the synthetic benchmarks, we use six virtual machines of different granularities allowing better measurement of decoding overhead. They all have twenty instructions, without parameter for the first three machines, but for the last three machines, six instructions have a parameter of length 2, 2, 3, 4, 5 and 7 bits. The opcodes are encoded based on Zipf-20 probabilities.

In the first machine, all twenty virtual instructions add one to a counter c_i; in the second machine each instruction does two additional integer operations; in the third one, each instruction does two additional memory accesses to simulate a stack. Machines 4 to 6 have parameters and do the same work as machines 1 to 3 respectively, but six instructions have parameters and add them to their own counter c_i. We use the same program for the six machines: it is a sequence of the twenty instructions, from instruction 1 to 20, performing $4 \cdot 10^5$ iterations; that is the last instruction does a jump to the first instruction which stops the execution when counter c_1 reaches this value. The opcodes are compressed based on the Zipf-20 probabilities which have an average length of 3.6 bits. Three decoders are applied on all six machines executed on two host processors.

An interpreter was used to decode the uncompressed programs. These programs are bytecoded, one byte for each opcode and, if applicable, two bytes by operand. The decoding is a computed branch, indexed by the opcode, to the virtual instruction implementation. It loads its operands, emulates and jumps back to the beginning of the decoding cycle.

Table 5 presents the timing results for compressed programs, relative to the uncompressed ones. The simple memory access form-a is disappointing but form-b and form-c are good. The two forms are close in performance even though form-c is often the better. Since form-b generates more compact interpreters there is an informed compromise to make.

From this experiment we can conclude that even with virtual instructions doing almost no work, as in machine 1, and with a small decoder, the decoding is around a 50% overhead (as in form-c used with a $k_r = 4$ decoder). This is an extreme artificial setting aimed to demonstrate the worst case performance. On the other hand, if we have instructions with no parameter and enough granularity, a speedup can be observed.

7.4 Summary of the experiments

In conclusion, for Java, the average compression factor is around 60% for 400 classes of the JDK 1.0.2 and the ten benchmarks. For half the benchmarks, the slowdown is hardly noticeable. This shows the practicality of the approach. The synthetic benchmarks show more explicitly the overhead of decoding our compressed bytecode, demonstrating that even a speedup can be achieved in some cases without macro-instructions. The Scheme results show that starting from a very general machine, our compression creates efficient and compact instructions. They also show that we can come close to gzip compression performance and still efficiently decode the compressed instructions.

8. RELATED WORK

Decoding of Huffman encoded instructions has also been studied at the hardware level by several researchers [14, 17, 2]. They usually decompress between the memory and the instruction cache. They do not use fast decoding methods applicable at the software level.

Ernst *et al.* [8] compress native code, using macro-instructions and fixing parameters, by generating a tailored VM from the intermediate form emitted by a C compiler. It is similar to Proebsting's [21] work. Their technique is competitive with gzip on native code. But it is not reported if the compression obtained is due to the use of the VM or the compression of the virtual program. Moreover, no timing of the execution of compressed programs is reported.

Cooper and McIntosh [5] reduce program size by replacing particular repetitive sequences of instructions with a branch. The code saving is on average 5%. Cooper *et al.* [6] searches, using a genetic algorithm technique, a combination of compilation techniques to reduce code size. These works differ from ours since they are done on native code and no Huffman encoding and argument compacting are applied.

Pugh [22] applies several techniques to compress Java class files. This work differs from ours since decompression must be performed before execution.

The work of Rayside *et al.* [23] also applies to class files, but these techniques does not apply to the bytecode itself.

Hoogerbrugge *et al.* [10] uses a similar strategy of the Thumb and MIPS16 processors [25, 13] to compress some parts of the program. But instead of applying compression on the binary executable, they automatically generate a tailored virtual machine for the intermediate form of the C program. When the intermediate form is translated into a virtual program, frequent sequences of virtual instructions are replaced by one opcode. This particular technique gives a 30% reduction in size compare to the virtual program. Our work is complementary by further reducing the size of the virtual programs using compressed virtual instructions.

Lucco [18] applies compression to x86 native code using a dictionary technique to keep track of repeated short sequences of instructions. At least one decompression must be performed before the execution of a basic block, requiring a buffer space to keep the decompressed copy. Our work differs as we apply it to the context of virtual machines and directly decode compressed instructions.

Clausen *et al.* [4] compresses bytecode by replacing repetitive sequences of JVM instructions by macro-instructions. They obtain an average of 85% compression factor with a slowdown from 19% to 27%.

The work of Evans and Fraser [9] has an identical goal as ours: direct execution of compressed bytecode. Their technique avoids variable length instructions contrary to our technique. They do not report any execution times.

Debray and Evans [7] use canonical Huffman code on binary executables, but using the slow decoding technique as in sub-section 3.1. They avoid compression on frequently executed parts of the code to obtain reasonable execution speed.

9. SUMMARY

This work has shown that decoding canonical Huffman encoded opcodes, at the software level, in the context of virtual instructions, can be done efficiently. The speed of decoding increases with the size of the decoder. A general structure of compact decoders has been shown effective, permitting a gradual compromise between speed of decoding and space constraints.

Huffman decoding is not the only difficulty for quickly interpreting compressed virtual instructions, memory access for variable length bit fields is also important. Two prefetching techniques were shown to achieve good results.

The efficiency of the decoders have been demonstrated on simple synthetic benchmarks, on the Scheme language, and on Java benchmarks showing an average slowdown ranging from 2% to 27% depending on the processor and the size of the decoders. Actually, half of the Java benchmarks have a 40% reduction in size with a negligible slowdown ($\leq 3\%$).

10. REFERENCES

[1] Java BYTEmark benchmarks: source code and results. `http://www.igd.fhg.de/~zach/benchmarks`, 1999.

[2] M. Benes, S. M. Nowick, and A. Wolfe. A fast asynchronous Huffman decoder for compressed-code embedded processors. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Sept. 1998.

[3] J. P. Bennet and G. C. Smith. The need for reduced byte stream instruction sets. *The Computer Journal*, 32:370–373, 1989.

[4] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems*, 22(3):471–489, 2000.

[5] K. D. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *Proc. Conf. on Programming Languages Design and Implementation*, 1999.

[6] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. *ACM SIGPLAN Notices*, 34(7):1–9, July 1999.

[7] S. Debray and W. S. Evans. Profile-Guided code compression. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[8] J. Ernst, C. W. Fraser, W. Evans, S. Lucco, and T. A. Proebsting. Code compression. In *Proc. Conf. on Programming Languages Design and Implementation*, pages 358–365, June 1997.

[9] W. S. Evans and C. W. Fraser. Bytecode compression via profiled grammar rewriting. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 148–155, 2001.

[10] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Wiel. A code compression system based on pipelined interpreters. *Software - Practice and Experience*, 29(11):1005–1023, Sept. 1999.

[11] D. A. Huffman. A method for the construction of minimum redundancy codes. In *Proc. IRE*, volume 40, pages 1098–1101, Sept. 1952.

[12] T. Kemp, R. Montoye, J. Harper, J. Palmer, and D. Auerbach. A decompression core for PowerPC. *IBM Journal of Research and Development*, 42(6), Nov. 1998.

[13] K. Kissell. *MIPS16: High-density MIPS for the Embedded Market*. Silicon Graphics MIPS Group, 1997.

[14] M. Kozuch and A. Wolfe. Compression of embedded system programs. In *Proc. Int'l Conf. on Computer Design*, pages 270–277, 1994.

[15] M. Latendresse. Automatic generation of compact programs and virtual machines for Scheme. In M. Felleisen, editor, *Proceedings of the Workshop on Scheme and Functional Programming*, Sept. 2000. Available at www.iro.umontreal.ca/~latendre/publications/.

[16] M. Latendresse and M. Feeley. Fast and compact decoding of Huffman encoded virtual instructions. Technical Report DIRO-1219, University of Montreal, Nov. 2002. Available at www.iro.umontreal.ca/~latendre/publications/.

[17] H. Lekatsas and W. Wolf. Code compression for embedded systems. In *Proc. ACM/IEEE Design Automation Conference*, 1998.

[18] S. Lucco. Split-stream dictionary program compression. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 27–34, Vancouver, British Columbia, June 18–21, 2000.

[19] A. Moffat and A. Turpin. On the implementation of minimum redundancy prefix codes. *IEEE Transactions on Communications*, 45(10):1200–1207, Oct. 1997.

[20] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 1–20, Berkeley, June 16–20 1997. Usenix Association.

[21] T. A. Proebsting. Optimizing a ANSI C interpreter with superoperators. In *Proc. Symp. on Principles of Programming Languages*, pages 322–332, 1995.

[22] W. Pugh. Compressing Java class files. In *Proc. Conf. on Programming Languages Design and Implementation*, pages 247–258, 1999.

[23] D. Rayside, E. Mamas, and E. Hons. Compact java binaries for embedded systems. In *Cascon*, pages 1–14, Nov. 1999.

[24] E. S. Schwartz and B. Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7(3):166–169, Mar. 1964.

[25] J. L. Turley. Thumb squeezes ARM code size. *Microprocessor Report*, 9(4), Mar. 1995.

The Case for Virtual Register Machines*

Brian Davis, Andrew Beatty, Kevin Casey, David Gregg and John Waldron
Department of Computer Science
Trinity College
Dublin 2, Ireland
David.Gregg@cs.tcd.ie

ABSTRACT

Virtual machines (VMs) are a popular target for language implementers. Conventional wisdom tells us that virtual stack architectures can be implemented with an interpreter more efficiently, since the location of operands is implicit in the stack pointer. In contrast, the operands of register machine instructions must be specified explicitly. In this paper, we present a working system for translating stack-based Java virtual machine (JVM) code to a simple register code. We describe the translation process, the complicated parts of the JVM which make translation more difficult, and the optimisations needed to eliminate copy instructions. Experimental results show that a register format reduces the number of executed instructions by 34.88%, while increasing the number of bytecode loads by an average of 44.81%. Overall, this corresponds to an increase of 2.32 loads for each dispatch removed. We believe that the high cost of dispatches makes register machines attractive even at the cost of increased loads.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.4 [**Programming Languages**]: Processors—*Interpreters*

Keywords

Interpreter, Virtual Machine, Register Architecture, Stack Architecture

1. MOTIVATION

Virtual machines (VMs) are a popular target for language implementers who wish to distribute programs in a portable, architecture-neutral format, which can easily be interpreted or compiled. Virtual machine code is also the intermediate format of choice for efficient, general-purpose virtual machine interpreters. The most popular virtual machines use a virtual stack architecture for evaluating expressions, rather than the register architectures that are commonly used in real processors.

Conventional wisdom tells us that stack architectures can be implemented with an interpreter more efficiently, since the location of operands is implicit in the stack pointer. In contrast, the operands of register machine instructions must be specified explicitly. The interpreter must fetch these operands from the virtual machine code, increasing the interpreter overhead, when compared with a stack architecture. It is also widely believed that stack architectures allow more compact virtual machine code, again because operands are specified implicitly. In addition, stack code is easier to generate in the compiler than register code. At the very least, a stack machine eliminates the need for a complicated register allocator. For these reasons, stack architectures have been used as the intermediate representations for a number of popular virtual machines, such as the Java VM, .NET VM, and Pascal P-code.

More recently, a number of authors and implementors of virtual machines have suggested that virtual register machines could be more efficient. Gregg et al. [8] mentioned the possibility in a general discussion of interpreter optimisations. Furthermore, the Parrot VM - the intermediate representation for Perl 6 - will use a register architecture because the implementers belief in the superiority of register machines. The Parrot VM has provoked a number of lively debates on newsgroups such as comp.compilers and comp.lang.perl on the relative merits of virtual stack and register machines. Despite the controversy, neither side has presented significant quantitative results comparing the two approaches, so no conclusion could be reached.

In this paper, we present a working system for translating stack-based Java virtual machine (JVM) code to a simple register code. We describe the translation process, the complicated parts of the JVM which make translation more difficult, and the optimisations needed to eliminate copy instructions. We also present a number of design choices, which can have a significant impact on the number of instructions in the resulting register machine program. We present quantitative results for real, large programs: the standard SPECjvm98 and Java Grande benchmark suites. We find that a virtual register architecture significantly reduces (34.88%) the number of executed instructions in Java programs , while increasing the number of bytecode fetches by only 44.81%.

The rest of this paper is organised as follows. Section 2 describes the basic functioning of a virtual machine in-

*This work was supported by Enterprise Ireland Research Innovation Fund, Grant IF/2001/350.

IVME'03, June 12, 2003, San Diego, California, USA.

```
typedef enum {
  add /* ... */
} Inst;

void engine()
{
  static Bytecode program[] = { iadd /* ... */ };

  Bytecode *ip;
  int *sp;

  while (1)
    switch (*ip) {
    case iadd:
      dest = ip[1];
      s1 = ip[2];
      s2 = ip[3];
      reg[dest]=reg[s1]+reg[s2];
      ip+=4;
      break;
    /* ... */
    }
}
```

Figure 1: Instruction dispatch using `switch`

```
typedef void *Inst;

void engine()
{
  static Bytecode program[] = { iadd /* ... */ };

  Bytecode *ip;
  Inst dispatch_table = { &&nop, &&aload_null, .... };
  int *sp;

  goto dispatch_table[*ip];

 iadd:
  dest = ip[1];
  s1 = ip[2];
  s2 = ip[3];
  reg[dest]=reg[s1]+reg[s2];
  ip+=4;
  goto dispatch_table[*ip];
}
```

Figure 2: Instruction dispatch using token threading in GNU C

terpreter, and the most important types of instruction dispatch. In section 3 we describe the main differences between virtual stack and virtual register machines. Section 4 looks at the particular strengths and weaknesses of the two types of virtual architecture, and estimates the relative advantages of each. In section 5 we present our translation system to convert stack Java bytecode to an equivalent virtual register code. Section 6 examines techniques for eliminating move instructions from register code. Finally in section 7 we present results showing that a register format can significantly reduce the number of executed instructions for the same program.

2. VIRTUAL MACHINE INTERPRETERS

The Java Virtual Machine uses a stack-based bytecode to represent the program. Interpreting a bytecode instruction consists of accessing arguments, performing the function of the instruction, and dispatching (fetching, decoding and starting) the next instruction.

Instruction dispatch typically consumes most of the execution time in virtual machine interpreters. The reason is that most VM instructions require only a small amount of computation, such as adding two numbers or loading a number onto the stack, and can be implemented in a few machine code instructions. In contrast, instruction dispatch can require up to 10-12 machine code instructions, and involves a time consuming indirect branch. For this reason, dispatch consumes a large proportion of the running time of most efficient interpreters [3].

Switch dispatch is the simplest and most widely used approach. The main loop of the interpreter consists of a large `switch` statement with one `case` for each opcode in the JVM instruction set. Figure 1 shows how this approach is implemented in C.

Switch dispatch is simple to implement, but rather inefficient for a number of reasons. First, most compilers produce a range check to ensure that the opcode is within the range of valid values. In the JVM this is unnecessary, since the bytecode verifier already checks that bytecodes are valid. Secondly, the `break` is translated into an unconditional jump back to the start of the loop. Given that the loop already contains a jump, it would be better to structure the loop as a set of routines that jump to one another. A final source of inefficiency results from there being only a single indirect branch for dispatching instructions. On machines with programmer visible pipelines, such as the Philips Trimedia processor for embedded systems, it is difficult to overlap this branch with other instructions [9]. On processors with branch predictors, this branch is very unpredictable [3]

An alternative to using a `switch` statement is threaded dispatch. Threaded dispatch is based on making explicit the sequence of steps generated by a compiler to implement a `switch` statement. Once these steps appear at the source level, the programmer can optimize the code by removing unnecessary work. Unfortunately, it is not possible to break a `switch` statement into its component parts in ANSI C, because there is no facility for `goto` statements that can jump to multiple different locations. To implement threaded dispatch, one requires a language with labels as first class value, such as GNU C, the language accepted by the GCC compiler.

Figure 2 shows how *token threaded* dispatch can be implemented using GNU C. The range check has been eliminated, as has the jump back to the dispatch routine at the end of the code for each VM instruction. Instead, the dispatch code is appended to the end of the code for each virtual machine instruction. This increases the size of the interpreter slightly, although it is usually faster. Another effect of replicating the dispatch code is that it allows the dispatch branch to be scheduled more efficiently with the code to implement the bytecode instruction, and it also greatly increases the prediction accuracy of the indirect branch on processors with branch target buffers (45% versus 2%–20% for switch dispatch) [3].

3. STACK VERSUS REGISTERS

The cost of executing a virtual machine instruction consists of three components:

- Instruction dispatch
- Operand access
- Performing the computation

The cost of dispatching an instruction is essentially the same for virtual register and stack machines. However, a given computation can often be expressed using fewer register machine instructions than stack ones. For example, the local variable assignment `a = b + c` might be translated to JVM code as `iload c, iload b, iadd, istore a`. In a virtual register machine, the same code would be the single instruction `iadd a, b, c`. Thus, virtual register machines have the potential to significantly reduce the number of executed instructions. By how much? It depends on how often values must be loaded to, stored from, or shuffled around the stack. If the computation can be organised so that operands can always be found on top of the stack, changing to a register architecture would give no reduction in executed instructions.

Another reason for pessimism with the number of executed instructions on register machines relates to register allocation. The number of virtual registers is always limited, and if there are more live values than registers, some values must be spilled to memory. Additional load and store instructions must be added to access spilled values, increasing the number of executed instructions rather than reducing them.

Our experience is that this argument is something of a red herring, at least for the Java VM. The most commonly used instructions for loading and storing local variables use a one-byte index, which specifies the number of the local variable. A comparable virtual register machine would use a one-byte index to specify each of its operands, allowing up to 256 virtual registers to be used in each method. Measurements show that no methods in the standard SPECjvm98 and Java Grande benchmarks contain anything like 256 local variables or stack values, the values that could be allocated to registers. On the contrary, most methods contain less than 25 such values [16].

The second component of the cost of executing a VM instruction is accessing the operands. This consists of two separates costs - finding the location of the operands, and accessing the operands themselves. Finding the operands' locations is expensive for a virtual register machine. The location of each operand must be fetched from the instruction stream, and used as an index into an array of virtual registers. In contrast the cost of locating operands is lower on stack machine, since most operands are found on the top of the stack. The main cost is updating the stack pointer, and even this is not always necessary.

Virtual registers and virtual stacks are usually implemented as arrays in memory, so the cost of accessing the operands themselves is similar for both types of virtual architecture. Stack caching can be used, however, to reduce the cost of accessing virtual stacks by keeping the top one or two items in a register. For example, Ertl [2] found that keeping the topmost stack item in a register reduced memory traffic for stack items by about 50%. It is very difficult to keep virtual register items in real machine registers, because real machine registers can not be accessed array-like, with an index. The final component of the cost of executing a VM instruction is actually performing the computation itself. Given that most VM instruction perform a simple computation, such as an add or a load, this is usually the smallest part of the cost of executing a VM instruction. Generally, the type of virtual machine will not make a difference to this cost. The basic computation has to be performed, regardless of the format of our intermediate representation. However, there are situations where a register architecture can allow slightly more efficient code. In particular, exploiting common subexpressions is easier on a register machine that does not destroy its operands when using them, as a stack machine normally does.

4. SOME ESTIMATES

Clearly, the difference in speed between a virtual stack machine and a corresponding register machine can depend on many factors, especially on modern out-of-order processors with branch prediction and caches that make performance difficult to predict. However, we believe that the difference between the two types of machine can be estimated by looking at four main factors. The running of a virtual register machine (VRM) might be compared to a virtual stack machine (VSM) as follows:

$$T_{VRM} \approx T_{VSM} - \#dispatches \times T_{dispatch} + \#fetches \times T_{fetch}$$

In other words, the running time of a program on a VRM will be approximately equal to the running time of the program on a corresponding stack machine, minus the reduction in dispatches times the cost of a dispatch, plus the increase in fetches of operand locations times the cost of each of those fetches.

Generally, we would expect that the increase in fetches is likely to be large, since most VM instructions need one or two extra immediate operands to specify register locations. The cost of each of these fetches is likely to be low, however, since each usually corresponds to just an additional load instruction.

The reduction in dispatches is much more difficult to estimate without looking at real programs. However the cost of each dispatch is likely to be large. Ertl and Gregg [3] found that virtual machine interpreters contain very large numbers of indirect branches (up to 13% of all executed real machine instructions). Furthermore, these branches are highly (60%-97%) unpredictable on current desktop and workstation processors. The cost of each indirect branch misprediction is high, because it requires that the entire pipeline be drained, consuming 6–20 cycles, depending on the length of the pipeline. Ertl and Gregg [3] found that more than half of the execution time of many efficient interpreters is spent on indirect branch mispredictions. Almost anything that reduces the number of dispatches has the potential to significantly improve performance.

Another complication is that the cost of all dispatch mechanisms is not the same. As outlined in section 2, threaded dispatch is about twice as fast as switch dispatch, although it cannot be implemented in ANSI C. Thus, register machines might prove more efficient where the interpreter must be written in ANSI C for maximum portability, while a stack architecture might have an edge where GNU C or assembly language is acceptable.

```
ILOAD 4     IMOVE r10, r4       ; load local 4
ILOAD 5     IMOVE r11, r5       ; load local 5
IADD        IADD r10, r10, r11  ; integer add
ISTORE 6    IMOVE r6, r10       ; store TOS to local 6
ILOAD 6     IMOVE r10, r6       ; load local 6
IFEQ 7      IFEQ r10, 7         ; branch by 7 if TOS == 0
```

Figure 3: Example of stack and corresponding register code

5. FROM STACK TO REGISTER

To compare the relative benefits of virtual stack and register machines, we constructed a system for translating stack-based Java bytecode to a similar register code. Our translation scheme is based on mapping local variables and stack locations to a single set of virtual registers. In the JVM, all local variables are numbered, and we translate local variable numbers directly to virtual register numbers (so local variable zero is mapped to register zero).

Mapping stack locations to virtual register numbers is a little more complicated. Each stack location is given a number, and those numbers start just after the position of the last local variable. Mapping stack locations to register numbers is much simplified by Java's strict stack discipline. It is not possible to write code that, for example, increases the number of items on the stack on each iteration of a loop, as can be done in Forth. At every point in the program, the height of the stack must be fixed, and can be determined by simple static analysis. At control flow join points the height of the stack must be equal on both incoming control flow edges. Thus, by tracking the value of the stack pointer at each point in the program, it is possible to map stack locations to register numbers.

Figure 3 shows an example of Java stack bytecode and the corresponding register code. Note that in the register code, the first register operand is always the destination. We assume that there are ten local variable slots in this method (`r0..r9`), so the stack pointer for the initially empty stack will point to `r10`. Thus, when we translate an `ILOAD` instruction which copies the value in local variable 4 to the top of the stack, we translate this as an integer move (`IMOVE`) instruction from register `r4` to register `r10`.

Similarly, we translate the `IADD` stack instruction to a register `IADD` instruction that takes the topmost item in the stack (`r11`) and the second from top (`r10`), adds the two and places the result in the new topmost stack item (`r10`), which will be one lower than the previous top of stack because `IADD` reduces the height of the stack by one.

Using this scheme, it is relatively easy to translate any sequence of stack Java bytecode to an equivalent register format. It is important to note that the resulting code will often contain unnecessary and redundant `MOVE` instructions. For example, the original stack code contains the sequence `ISTORE 6, ILOAD 6`, which stores the topmost stack item to local variable number 6, and then reloads the value to the top of the stack. This type of sequence is actually extremely common in code produced by the *javac* compiler. Presuming that the value is stored to the local variable to allow it to remain live after the end of the basic block, it is not possible to express this in fewer instructions. In the corresponding register code, however, it is easy to remove many of these `IMOVE` instructions.

One type of instruction that needs special handling in the translation is method invocation instructions. Invoke instructions take their parameters from the top of the stack. These n topmost stack items become the first n local variables of the invoked method. Thus, invoke instructions can consume several values, and they destroy these values in the process.

In theory, this scheme allows extremely fast parameter passing, since making the topmost stack elements into the first local variables simply involves one assignment to the frame pointer. In practice, however, the parameters are rarely already on the stack, and most invoke instructions are preceded by one or more load instructions. Furthermore, once the parameters have been passed to the invoked method, they are in local variables and must be loaded to the stack before they can be used.

The simplest way to translate the parameter passing mechanism would be a completely literal translation, where the topmost registers of the caller become the first registers of the callee. In our first implementation we used this scheme, but found that it prevented us from removing very large number of load instructions when translating to register format. The problem is that invoke instructions consume several values, each of which must be in a specific register, and so cannot be moved.

Our new scheme uses an alternative scheme where each invoke takes as an immediate argument a list of the registers that it takes as parameters. Although this increases the number of loads necessary to identify the location of operands, it allows us to eliminate the great majority of `MOVE` instructions in register code.

6. ELIMINATING MOVES

Translating the Java bytecode to register code does not automatically reduce the number of executed instructions. The translation process outlined in the previous section simply converts each instruction directly from a stack to a register format. To eliminate unnecessary `MOVE` instructions, we apply a copy propagation algorithm that rewires the source and destination registers of instructions to bypass `MOVE`s. Once the sources and destinations have been changed, many of the `MOVE` instructions become dead code and can be eliminated.

We implemented two copy propagation algorithms, the first operating only on basic blocks and the second operating on the entire Java method. The basic block algorithm is both simple and efficient, and allows copy operations to be bypassed within a basic block. It is important to note that most values that are loaded to the stack are used very soon afterward, so a basic block algorithm can be very effective.

One complication that normally arises with single-basic-block copy propagation is that it is difficult to eliminate dead copies, because it is not clear which values are still alive at the end of the basic block. In Java bytecode, the problem is very much easier, since most destinations of `MOVE` instructions are on the stack. We can easily identify when most values on the stack become dead because the stack pointer moves below them (anything above the stack pointer is dead). Furthermore, the standard idiom used by the *javac* compiler is that the stack should be empty at the start and end of each statement. Thus, in the great majority of cases, the stack is empty at the end of each basic block, and all the items on the stack are dead.

Our second copy propagation algorithm operates on an entire method at a time. It uses classic dataflow analysis to compute liveness sets and propagate copies across basic block boundaries. Thus, it is much slower and more complicated that our basic block algorithm, and is probably not suitable for using in a scheme that translates from stack code to register code at load time. However, when we compared the two algorithms we found that there is a difference of less than 1% in the results. Clearly, a simple, efficient basic-block approach with liveness based on stack position is sufficient in most cases.

7. EXPERIMENTAL EVALUATION

Our basic thesis is that virtual register machines have the potential to be interpreted more efficiently than stack machines by reducing the number of executed instructions. To test this thesis, we implemented a system for translating Java bytecode to a corresponding register format, and measured the differences using the SPECjvm98 [14] and Java Grande [1] benchmarks. These benchmarks consists of several large programs with real data, which are intended to be representative of a wide range of Java applications.

Our translation system was built into CVM, a small implementation of the Java 2 Micro Edition (J2ME) standard. It supports the full JVM instruction set, as well as full system-level threads. We made a number of small additions to CVM to enable it to run the SPEC benchmarks and to allow us to safely compile it at a higher level of optimization than the standard distribution. All quoted figures are for the basic block implementation of copy propagation and dead copy elimination, as we believe this to be the most practical scheme. The whole-method copy propagation gives only slightly (less than 1%) better results.

The first time each method is invoked we translate it to register format. Thus, we present measurements only for methods that are executed at least once. Table 1 and figure 4 show the breakdown of instructions after translation to register format, based on statically appearing code. Overall, an average of 34.11% of statically appearing instructions are `MOVE` instructions. 28.56% of total instructions are `MOVE`s that can be eliminated with copy propagation and dead code elimination.

Table 2 and figure 5 show the breakdown of dynamically executed instructions. Interestingly, 41.28% of dynamically executed instructions are `MOVE`s. Clearly, local loads and stores are not distributed evenly throughout the programs, and code with larger numbers of such instructions tends to be executed more frequently. An average of 34.88% of executed instructions can be eliminated by translating to a register format. At more than one third of executed instructions, this is a very large number and strongly suggests that virtual register machines could be interpreted more efficiently than stack machines. This is especially likely to be true where the interpreter uses `switch` dispatch (see section 2), such as where the interpreter must be written in ANSI C.

In the last two columns of figure 2 and in figure 6 we examine the net increase in bytecode loads in the interpreter caused by the register format. These figures assume that each opcode and each register operand occupies one byte, each of which must be loaded separately. There are two effects at work here. First, translating to a register format increase the number of operands in the bytecode. Secondly, applying copy propagation and dead copy elimination allows us to eliminate a large number of instructions, thus reducing both the number of opcodes and operands. Overall, the register format requires an average of 44.81% extra bytecode loads. Clearly this is a large number, but loads are usually very much less costly than the indirect branches in instruction dispatches.

We also examined the ratio of the increase in the number of loads to the reduction in dispatches. We found that for most of the SPEC benchmarks the register format increased the number of loads required by an average of about two extra bytecode loads for every dispatch eliminated. The Grande benchmarks required quite a few more loads for each dispatch removed, such as a ratio of more than five to one for the *Euler* benchmark. We believe that this is probably caused by the Grande benchmarks being more mathematically oriented, and thus using more complicated expressions which can be expressed more elegantly in stack code. Over all the SPECjvm98 and Grande benchmarks, translating to bytecode increased the number of bytecode loads by an average of 2.32 for every dispatch eliminated. We believe that for `switch` based interpreters running on modern pipelined processors where the cost of branch mispredictions is very high, even 2.32 extra loads for each dispatch removed will still result in a significant benefit to the virtual register machine interpreter.

8. RELATED WORK

Recent important developments in interpreters include the following. Interpreter generators simplify construction and maintenance of interpreters and can allow automatic VM instruction combining [12] and stack optimizations [4]. Stack caching [2] is a general technique for storing the topmost elements of the stack in registers. Ertl and Gregg [3] showed that interpreters (especially those using switch dispatch) spend most of their time in branch mispredictions on modern desktop architectures. Interpreter software pipelining [9] is a valuable technique for architectures with delayed branches (e.g. Philips Trimedia) or prepare to branch instructions (e.g. PowerPC), which makes the target of the dispatch branch available earlier by moving much of the dispatch code into the previous VM instruction. Costa [13] discusses various smaller optimizations.

The Sable VM [6] is an interpreter-based research JVM. This interpreter uses a run-time code generation system [11], not dissimilar from a just-in-time compiler. Sable uses a novel system of *preparation sequences* [7, 5] to deal with bytecode instructions that perform initialisations the first time they are executed, which make code generation difficult.

Myers [10] attempts to refute the idea that stack machines will necessarily result in smaller code, with lower cost to access operands. The argument is based on measurements of real programs which show that the expression in most assignment statements is extremely simple. Thus, in most cases operands must be loaded to the stack for use, rather than already being there as part of the evaluation of a complex expression. However, beyond measurements of the complexity of expressions, Myers presents only a handful of small examples showing situations where register code is superior to stack code.

The controversy between stack and register code has arisen again recently because of the decision to make the the Parrot

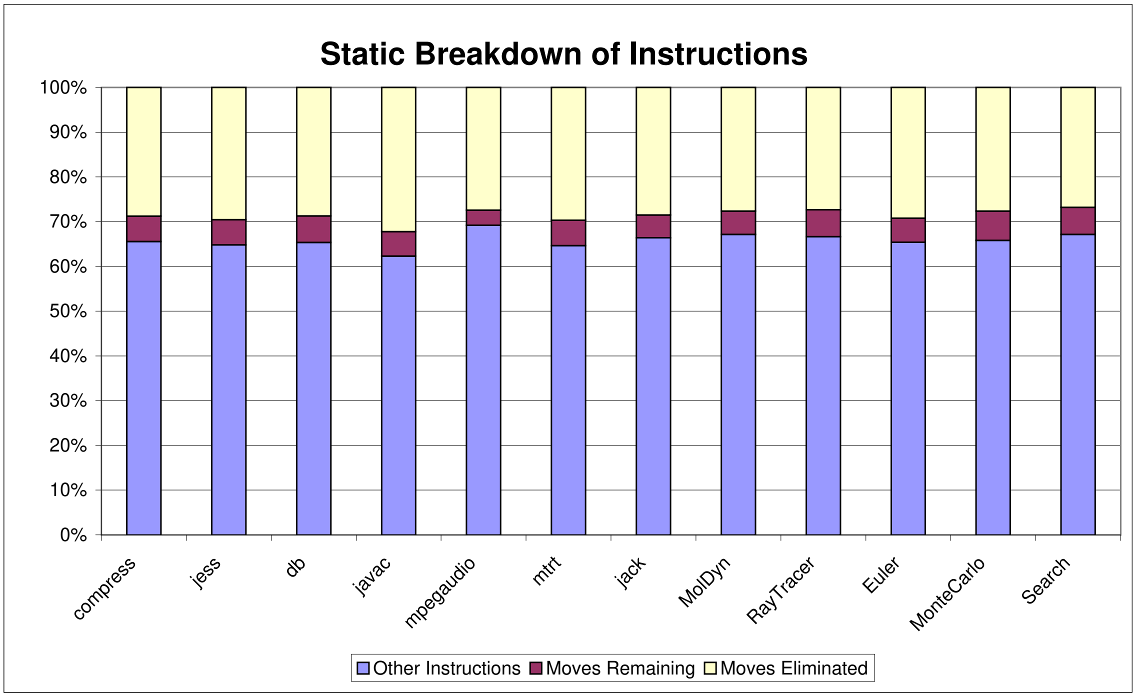


Figure 4: Breakdown of statically appearing register code instructions into moves eliminated, moves that could not be eliminated and other instructions.

Benchmark	Instructions	Moves	%	Eliminated	%
compress	28,612	9,852	34.43%	8,227	28.75%
jess	38,537	13,557	35.18%	11,392	29.56%
db	29,365	10,167	34.62%	8,434	28.72%
javac	59,545	22,442	37.69%	19,179	32.21%
mpegaudio	58,823	18,126	30.81%	16,135	27.43%
mtrt	33,969	12,004	35.34%	10,079	29.67%
jack	44,709	15,027	33.61%	12,737	28.49%
MolDyn	31,873	10,465	32.83%	8,811	27.64%
RayTracer	20,999	7,006	33.36%	5,739	27.33%
Euler	28,003	9,684	34.58%	8,178	29.20%
MonteCarlo	23,442	8,010	34.17%	6,477	27.63%
AlphaBetaSearch	21,328	7,005	32.84%	5,717	26.81%
Average	34,636	11,838.85	34.11%	9,982	28.56%

Table 1: The number of static instructions that can be potentially removed and the the number that are actually removed, compared with the total number of instructions.

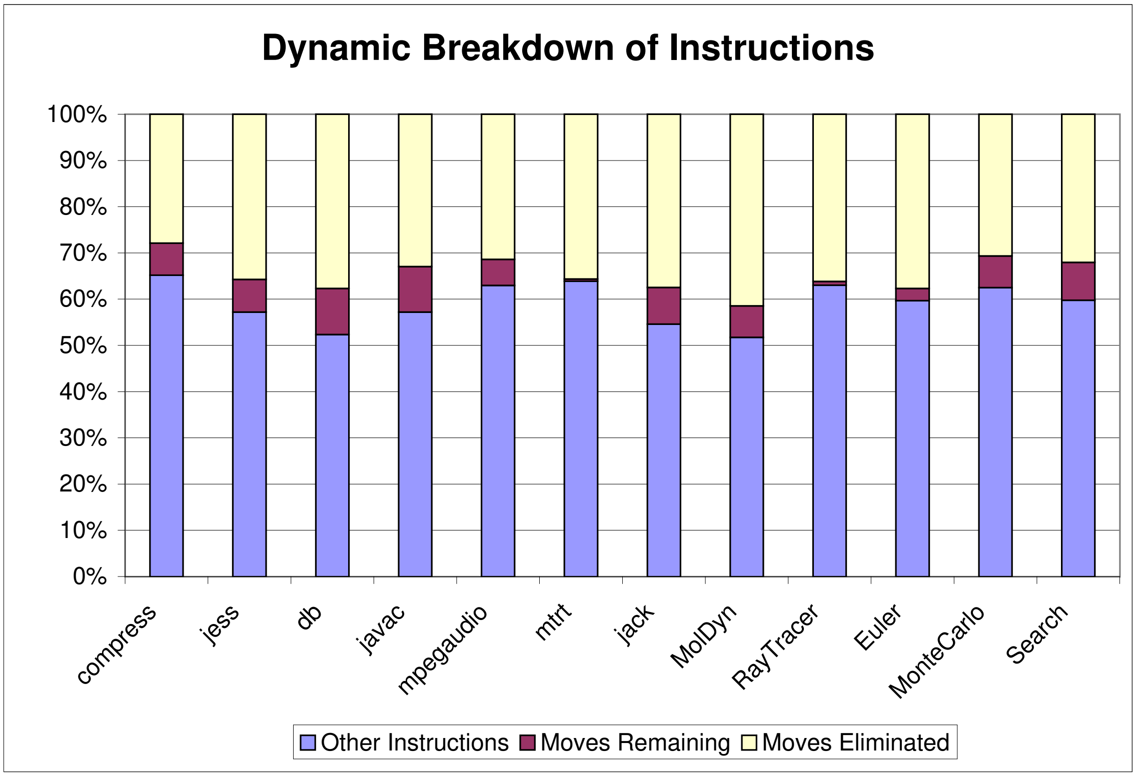


Figure 5: Breakdown of dynamic register code instructions into moves eliminated, moves that could not be eliminated and other instructions.

Benchmark	Instructions	Moves	%	Eliminated	%	extra loads	loads/dispatch
compress	4,917	1,713	34.84%	1372	27.91%	5280	3.85
jess	979	419	42.82%	349	35.73%	657	1.88
db	1,135	541	47.68%	428	37.69%	750	1.75
javac	1,335	571	42.82%	440	32.95%	952	2.17
mpegaudio	4,805	1,779	37.04%	1509	31.40%	4387	2.91
mtrt	970	350	36.13%	346	35.63%	667	1.93
jack	611	277	45.40%	229	37.48%	347	1.52
MolDyn	7,589	3,663	48.26%	3147	41.48%	2883	0.92
RayTracer	7,177	2,654	36.98%	2596	36.18%	7594	2.92
Euler	10,162	4,100	40.35%	3830	37.69%	9082	2.37
MonteCarlo	1,625	609	37.52%	498	30.67%	1717	3.45
AlphaBetaSearch	4,780	1,924	40.26%	1531	32.04%	4563	2.98
Average	3,545	1,431	41.28%	1252	34.88%	2991	2.32

Table 2: The number of executed instructions (in millions) that can be potentially removed and that are actually removed, compared with the original total. The net increase in bytecode fetches (in millions) is shown in second rightmost column. The rightmost column shows the number of extra bytecode loads for each VM instruction dispatch eliminated.

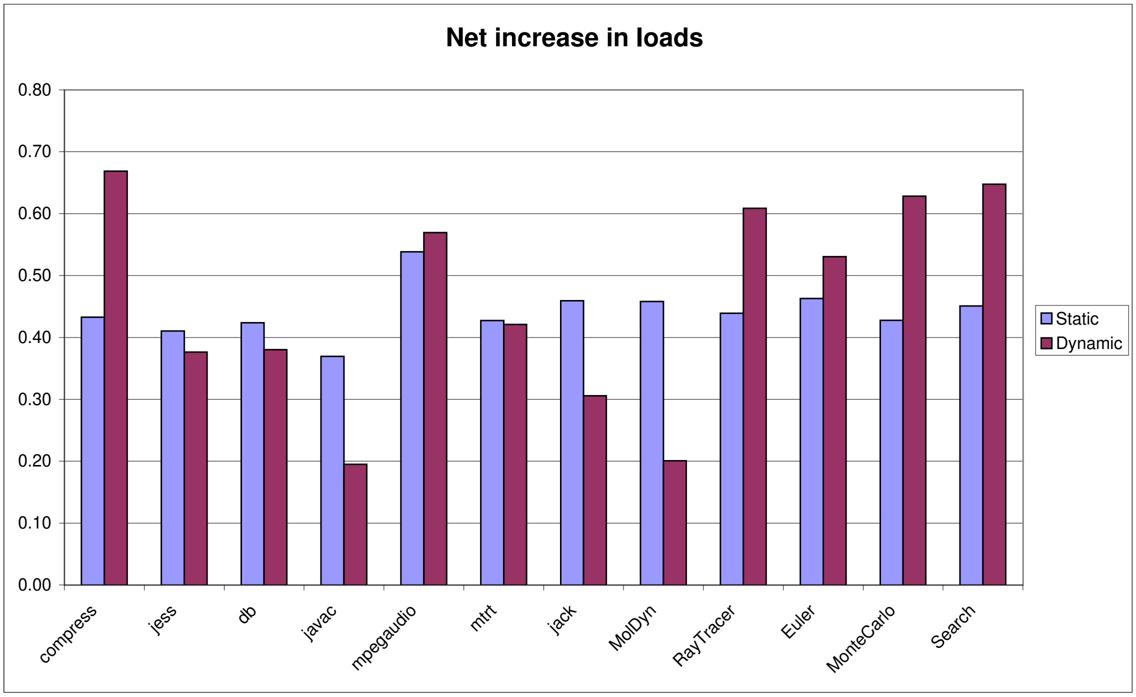


Figure 6: Net increase in bytecode loads from using a register rather than stack architecture.

VM, the intermediate representation for the Perl 6 language, a register rather than stack machine. Again, arguments for this design decision [15] have been based on just a couple of small examples, rather than any study of real programs.

9. CONCLUSION

Virtual register machines are an attractive alternative to virtual stack architectures because they allow the number of executed instructions to be reduced by eliminating large number of loads to and stores from the stack. This is especially important for interpreters modern pipelined processors, where the cost of instruction dispatch is very high.

We have described a system for translating Java bytecode to a corresponding register format. We have implemented this system in a real JVM and used it to collect data on the the effect of translating the SPECjvm98 and Java Grande benchmarks to register format. We believe that ours is the first quantitative data that measures hard numbers in real programs, rather than basing arguments on small examples. We found that translating to a register format decreases the number of executed instructions by an average of 34.88%, while increasing the number of bytecode loads by an average of 44.81%. Overall, this corresponds to an increase of 2.32 loads for each dispatch removed. We believe that the high cost of dispatches makes register machines attractive even at the cost of increased loads.

10. REFERENCES

[1] M. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. Benchmarking Java Grande applications. In *Second International Conference and Exhibition on the Practical Application of Java*, Manchester, UK, April 2000.

[2] M. A. Ertl. Stack caching for interpreters. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315–327, 1995.

[3] M. A. Ertl and D. Gregg. The behaviour of efficient virtual machine interpreters on modern architectures. In *Euro-Par 2001*, pages 403–412. Springer LNCS 2150, 2001.

[4] M. A. Ertl, D. Gregg, A. Krall, and B. Paysan. vmgen — A generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.

[5] E. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, Mc Gill University, December 2002.

[6] E. Gagnon and L. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *First USENIX Java Virtual Machine Research and Technology Symposium*, Monterey, California, April 2001.

[7] E. Gagnon and L. Hendren. Effective inline-threaded interpretation of java bytecode using preparation sequences. In *Proceedings of the 12th International Conference on Compiler Construction*, LNCS 2622, pages 170–184, April 2003.

[8] D. Gregg, A. Ertl, and A. Krall. A fast java interpreter. In *Proceedings of the Workshop on Java optimisaton strategies for embedded systems (JOSES)*, Genoa, April 2001.

[9] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Wiel. A code compression system based on pipelined interpreters. *Software—Practice and Experience*, 29(11):1005–1023, Sept. 1999.

[10] G. J. Myers. The case against stack-oriented

instruction sets. *Computer Architecture News*, 6(3):7–10, 1977.

[11] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[12] T. A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Principles of Programming Languages (POPL '95)*, pages 322–332, 1995.

[13] V. Santos Costa. Optimising bytecode emulation for Prolog. In *LNCS 1702, Proceedings of PPDP'99*, pages 261–267. Springer-Verlag, September 1999.

[14] SPEC. SPEC releases SPEC JVM98, first industry-standard benchmark for measuring Java virtual machine performance. Press Release, August 19 1998. http://www.specbench.org/osg/jvm98/press.html.

[15] D. Sugalski. *http://www.parrotcode.org/*.

[16] J. Waldron. Dynamic bytecode usage by object oriented java programs. In *Proceedings of the Technology of Object-Oriented Languages and Systems 29th International Conference and Exhibition*, Nancy, France, June 7-10 1999.

Dynamic Native Optimization of Interpreters

Gregory T. Sullivan
gregs@ai.mit.edu
Artifical Intelligence Lab

Derek L. Bruening
iye@lcs.mit.edu
Laboratory for Computer Science

Iris Baron
iris@lcs.mit.edu
Laboratory for Computer Science

Timothy Garnett
timothyg@lcs.mit.edu
Laboratory for Computer Science

Saman Amarasinghe
saman@lcs.mit.edu
Laboratory for Computer Science

Massachusetts Institute of Technology
Cambridge, MA

Abstract

For domain specific languages, "scripting languages", dynamic languages, and for virtual machine-based languages, the most straightforward implementation strategy is to write an interpreter. A simple interpreter consists of a loop that fetches the next bytecode, dispatches to the routine handling that bytecode, then loops. There are many ways to improve upon this simple mechanism, but as long as the execution of the program is driven by a representation of the program other than as a stream of native instructions, there will be some "interpretive overhead".

There is a long history of approaches to removing interpretive overhead from programming language implementations. In practice, what often happens is that, once an interpreted language becomes popular, pressure builds to improve performance until eventually a project is undertaken to implement a native *Just In Time* (JIT) compiler for the language. Implementing a JIT is usually a large effort, affects a significant part of the existing language implementation, and adds a significant amount of code and complexity to the overall code base.

In this paper, we present an innovative approach that dynamically removes much of the interpreted overhead from language implementations, with minimal instrumentation of the original interpreter. While it does not give the performance improvements of hand-crafted native compilers, our system provides an appealing point on the language implementation spectrum.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.4 [**Programming Languages**]: Processors—*Interpreters*

IVME'03, June 12, 2003, San Diego, California, USA.

1 Introduction

Certain kinds of programming languages lend themselves to an interpreted implementation: languages with a high degree of dynamism (e.g., dynamic OO languages), or with a premium on fast startup and smaller programs (e.g., scripting languages), or where simplicity of implementation is important. Indeed, it can be argued that nearly all sufficiently complex applications have elements of interpretation throughout[1].

Typically, the language front end will transform the surface syntax of a program to an intermediate representation of the program (e.g., bytecodes) that is then interpreted. A simple interpreter consists of a loop that fetches the next bytecode, dispatches to the routine handling that bytecode, then loops. There are many ways to improve upon this simple mechanism, but as long as the execution of the program is driven by a representation of the program other than as a stream of native instructions, there will be some "interpretive overhead".

We take an approach that is a combination of native *Just In Time* (JIT) compiler and *partial evaluation* techniques. We start with a dynamic optimization system called *DynamoRIO*, jointly developed at HP Labs and MIT. We give a quick overview of the DynamoRIO system in Section 2.

DynamoRIO records sequences of native instructions, which are subsequently partially evaluated with respect to the in-memory representation of the program being interpreted.

Suppose the application program is represented as an immutable vector of bytecodes, and we have a long trace of native instructions. If we look carefully at the trace, there will typically be a value (we call it the "Logical PC") used as an index into the bytecode vector, and there will be sequences of instructions as follows:

[1] Greenspun's Tenth Rule of Programming: "Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp."

1. Fetch the next bytecode, using the logical PC as an index into the vector;
2. Possibly dereference other nearby bytecodes, to be used as arguments to the handler for this bytecode;
3. A certain amount of conditional control flow may take place based on the value of the bytecode (e.g., a `switch` statement);
4. Jump to bytecode-specific instructions;
5. Increment the logical PC by a fixed amount;
6. Repeat.

The key insight of dynamic native partial evaluation is that if we know the starting logical PC at the start of the above sequence, then:

1. Dereferences from the bytecode array, indexed by the logical PC, can be statically folded to constants,
2. Conditional branches based on now constant values can be removed entirely, and
3. Direct increments to the logical PC can be identified and tracked, thus enabling continued partial evaluation.

2 The DynamoRIO Dynamic Optimization Framework

Our optimization infrastructure is built on a dynamic optimizer called DynamoRIO. DynamoRIO is based on the IA-32 version [5] of Dynamo [4]. It is implemented for both IA-32 Windows and Linux, and is capable of running large desktop applications.

The goal of DynamoRIO is to observe and potentially manipulate every single application instruction prior to its execution. The simplest way to do this is with an interpretation engine. However, interpretation via emulation is slow, especially on an architecture like IA-32 with a complex instruction set. DynamoRIO uses a typical trick to avoid emulation overhead: it caches translations of frequently executed code so they can be directly executed in the future.

DynamoRIO copies *basic blocks* (sequences of instructions ending with a single control transfer instruction) into a code cache and executes them natively. At the end of each block the application's machine state must be saved and control returned to DynamoRIO (a *context switch*) to copy the next basic block. If a target basic block is already present in the code cache, and is targeted via a direct branch, DynamoRIO *links* the two blocks together with a direct jump. This avoids the cost of a subsequent context switch.

Indirect branches cannot be linked in the same way because their targets may vary. To maintain transparency, original program addresses must be used wherever the application stores indirect branch targets (for example, return addresses for function calls). These addresses must be translated into their corresponding code cache addresses in order to jump to the target code. This translation is performed as a fast hashtable lookup.

To improve the efficiency of indirect branches, and to achieve better code layout, basic blocks that are frequently executed in sequence are stitched together into a unit called a *trace*. When connecting beyond a basic block that ends in an indirect branch, a check is inserted to ensure that the actual target of the branch will keep execution on the trace. This check is much faster than the hashtable lookup, but if the check fails the full lookup must be performed. The superior code layout of traces goes a long way toward amortizing the overhead of creating them and often speeds up the program [4, 31].

A flow chart showing the operation of DynamoRIO is presented in Figure 1. The figure concentrates on the flow of control in and out of the code cache, which is the bottom portion of the figure. The copied application code looks just like the original code with the exception of its control transfer instructions, which are shown with arrows in the figure.

Note that in this paper we will use the term *fragment* to mean either a basic block or a trace in the code cache.

3 Interpreters Confound DynamoRIO

An important heuristic upon which DynamoRIO's trace identification and collection strategy is based is that popular targets of jumps are good candidates for *trace heads*. Furthermore, it is assumed that execution will proceed most or all of the way through a trace most of the time. In other words, recording long instruction streams starting at common branch targets will result in execution spending most of its time in recorded traces.

Unfortunately, typical interpreter implementations foil DynamoRIO's trace collection heuristic. In the case of a simple read-switch-jump loop, the interpreter's hottest loop head is in fact a poor candidate for a trace head, as its body follows a different path for each byte code type.

Threaded interpreters pose a related problem for DynamoRIO. The pervasive use of indirect jumps for control transfer foils DynamoRIO's trace head identification heuristics, and, even if a trace head were to be identified, the address of, e.g., the `CALL` bytecode does not uniquely identify a commonly occurring long sequence of native instructions.

3.1 Logical PC's

The goal is to record a long series of native instructions that:

1. will often be invoked, and
2. will usually run most or all of the way to the end once started.

An important realization for applying dynamic optimization techniques to interpreters is that we need a pair, (*native PC*, *logical PC*) to uniquely identify the current overall computation point. By *logical PC*, we mean some unique identifier into the control flow structure of the interpreted application. Neither the logical nor native PC is sufficient. For example, the native PC corresponding to the start of the handler for a `CALL` bytecode would be executed for each call site encountered in an interpreted application. The logical PC, on the other hand, might stay constant over a significant amount of interpreter execution (consider the interpretation of the `invokevirtual` JVM instruction). We call the (*logical PC*, *native PC*) pair the *abstract PC*.

4 Instrumenting the Interpreter

There are two types of information the interpreter writer must supply to DynamoRIO: the identification of logical control flow actions and the location of the immutable program representation used to drive interpretation.

4.1 Logical Control Flow API

Every time that either the native PC changes (either by sequential execution or a branch or jump) or the logical PC changes (by a

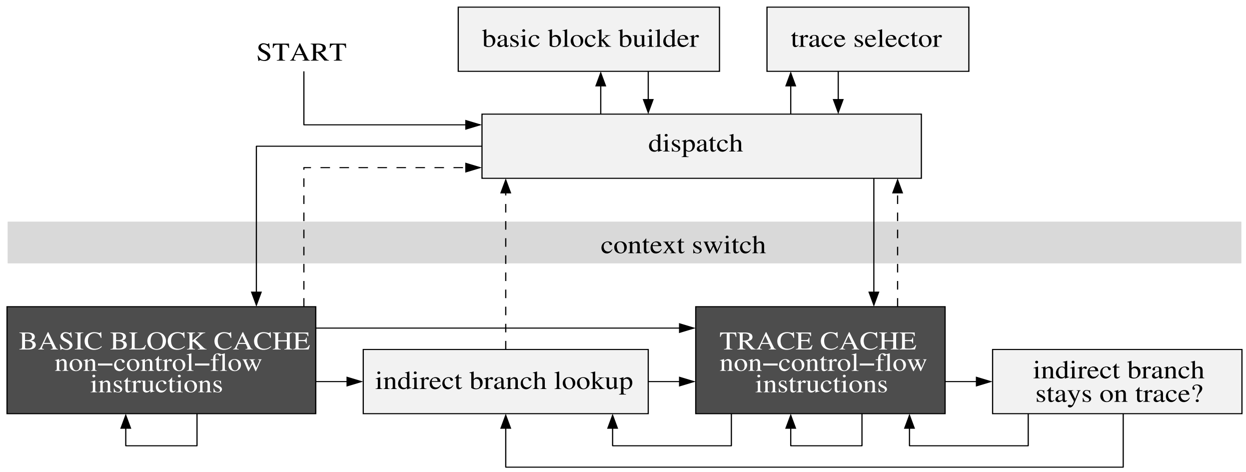


Figure 1. Flow chart of the DynamoRIO System. Dark shading indicates application code.

change in the controlling state of the interpreter), the abstract PC has changed. DynamoRIO provides the interpreter writer with a simple API for identifying relevant changes in the control state of the interpreter. Calls to these API functions enable DynamoRIO to identify potential *trace heads* and also to link traces together. Each of the three API functions correspond to typical native control transfer operations:

`logical_direct_jump(new_logpc)` corresponds to a direct native jump. By calling logical_direct_jump(x) at a particular point (native PC) in the interpreter, call it *n*, the interpreter writer promises that if the interpreter ever reaches *n* with the logical PC equal to its current value, say *y*, it will always make the logical control transfer to the next interpreter instruction (call it $n+1$) and logical PC *x*.

`logical_indirect_jump(new_logpc)` corresponds to an indirect native jump. As you might guess, this means that the interpreter cannot make the guarantee required for `logical_direct_jump`. A classic case of an indirect logical jump is the implementation of a `RETURN` bytecode, where the actual target is based on runtime data (e.g., a value on the stack) rather than a compile time constant.

`logical_relative_jump(offset)` corresponds to sequential native execution. Calling this function has the same guarantee as `logical_direct_jump` (that is, the current (native PC, logical PC) will always advance to (native PC + 1, logical PC + offset)) but also tells DynamoRIO that this transfer is "uninteresting". This corresponds to the fact that sequential execution through native code also involves regular changes to the native PC, but these are uninteresting control transfers.

Each of the three `logical_*_jump` functions identifies, when called, an abstract PC. The interpreter writer provides the logical PC value, and DynamoRIO provides the native PC value, for any given call. The native PC value for a logical control transfer is the address of the instruction following the call to the logical PC API function.

4.2 Identifying Immutable Program Data

In order for constant propagation to be able to fold dereferences of the program representation to constant values, the interpreter writer must identify regions of memory that hold immutable representations of the application, as well as identifying other memory locations that will be constant for given *(logical PC, abstract PC)* pairs. There are three API functions for providing this information:

`set_region_immutable(start, end)` identifies the region of memory delimited by `start` and `end` (inclusive) as immutable. `set_region_immutable` may be called for any number of regions. Once called for a given *start* - *end* region, that region of memory must not be written to for the duration of the program run.

`add_trace_constant_address(addr)` identifies an address whose value (i.e. when dereferenced) is guaranteed to be the same whenever control reaches the abstract PC of the call. For example, suppose a call `add_trace_constant_address(&pc)` is made when the logical PC is currently *lpc*, the native PC of the call site is *npc*, and the value of `pc` is 42. If DynamoRIO makes a trace headed by abstract PC *(lpc, npc)*, then any dereferences of `pc` can be folded to the constant 42. Furthermore, if this constant is then used to dereference values in immutable memory (e.g. `bc = byte_codes[pc]`) that dereference can also be folded to a constant.

`set_trace_constant_stack_address(addr, val)` identifies a stack-allocated variable (at address `addr` on the stack) that currently has the value `val`. The meaning of a call `set_trace_constant_stack_address(&pc, pc)` is just like a call to `add_trace_constant_address(&pc)` described above, except that the current value (`pc`) is made explicit in the call so that Dynamo can note the stack offset calculation used to dereference the stack-allocated variable. DynamoRIO will only fold dereferences of `addr` to a constant when control is within the stack frame of the call to `set_trace_constant_stack`.

If an address is identified as containing a trace constant, that address must not be *aliased*. That is, if `pc` is identified as a trace constant, changes to the value stored in `pc` must be apparent in the instruction sequence, rather than indirectly through another memory location.

For example,

```
ByteCode* pc = ...;
ByteCode* alias = pc;
add_trace_constant_address(&pc);
*alias = some_bytecode;
```

would violate the anti-aliasing requirement.

5 Example: TinyVM

In this section, we present a simple VM called `TinyVM`, written in C, to demonstrate the instrumentation by a programmer and the run-time optimization performed by DynamoRIO[2].

We start by adding calls to `dynamorio_app_init` and `dynamorio_app_start` to the startup code of TinyVM. Similarly, we arrange to call `dynamorio_app_stop` and `dynamorio_app_exit` when TinyVM exits. This is standard procedure for using DynamoRIO, as explained in the DynamoRIO documentation.

The main dispatch loop of TinyVM deconstructs the element of the bytecode vector `instrs` pointed to the logical PC `pc`, then finds the matching case in a `switch` statment:

```
loop:
  op = instrs[pc].op;
  arg = instrs[pc].arg;
  switch (op) {
    ...
```

Calls to functions are statically dispatched, and the target (bytecode offset) of the call is embedded in the bytecode stream. Thus, a call is a direct jump (i.e., from a given callsite, control will always transfer to the same target), and we instrument the handling of the `CALL` opcode as follows:

```
case CALLOP:
  ... setup new call frame ...
  pc = arg;     /* go to start of function body */
  dynamorio_logical_direct_jump(pc);
  goto loop;
```

As mentioned earlier, a `RETURN` bytecode is a case of an indirect logical control transfer. We instrument the `RETURN` case as follows:

```
case RETOP:
  ... clean up stack ...
  pc = pop_raw(); /* pop the return PC */
  dynamorio_logical_indirect_jump(pc);
  *(++sp) = val;   /* put return value back on stack */
  goto loop;
```

The case for bytecodes implementing conditional branches is more interesting. You might think that the logical branches would be indirect, but in fact they are often direct. In the following code, implementing the `BEQ` bytecode, the true branch is a direct logical jump (recall that the `arg` value came from the bytecode vector), and the false branch is a direct relative branch:

```
case BEQOP:
  n = pop_int();     /* top must be an int */
  if (n) {           /* not equal 0 */
    pc++;            /* just continue */
    dynamorio_logical_relative_jump(1);
  } else {           /* equal 0 */
    pc = arg;        /* do the branch */
    dynamorio_logical_direct_jump(pc);
  }
  goto loop;
```

The reason the logical transfers are direct is that the native conditional branch has already distinguished between the true and false cases. For each call to `dynamorio_logical_*`, the combination of the current native PC and the current logical PC uniquely identifies the target (native PC, target PC) abstract PC.

Finally, we inform DynamoRIO of immutable regions of memory and of any logical PC pointers:

```
set_region_immutable(instrs,
    (instrs + num_instrs*sizeof(ByteCode)−1));
dynamorio_add_trace_constant_address(&pc);
```

6 Collecting Logical Traces

The interpreter writer supplies the target logical PC (or offset) to the `dynamorio_logical_*` calls. When first processing the basic block, DynamoRIO adds the global DynamoRIO context, the native PC, and some "from context" data to the call also, so the actual functions receive four arguments. The translation of a call `logical_direct_jump(x)` at native PC *n* is given in Figure 2.

```
  push x              ; logical pc
  call set_log_pc     ; saves logpc (x) in a global
  pop 1
  jmp exit_stub
...
exit_stub:
  push dcontext       ; add arg. for logical_jump call
  push n+1            ; add native PC arg.
  push from_data      ; for linking traces
  call logical_direct_jump
  test next_logical_trace   ; if logical jmp to trace
  jne *next_logical_trace   ; go there
  next_addr = n+1     ; otherwise
  jmp dynamo_dispatch ; go to dynamorio with native PC = n+1
```

Figure 2. Translation of `logical_direct_jump(x)` at instruction *n*

Given the target logical and native PC's, DynamoRIO proceeds as

[2] TinyVM will be available in the released version of DynamoRIO as an example.

outlined by the following pseudo-code:

```
logical_jump(dcontext, native_pc, logical_pc, from) {
  if (currently building a trace) {
    set flag telling DynamoRIO to finish trace;
  } else { /* if not currently building a trace */
    /* get logical basic block for (logical, native) abstract PC */
    lbb = lookup(native_pc, logical_pc),
        adding new entry if necessary;
    lbb->count++;   /* count hits */
    if ((native PC, logical PC) entry corresponds
        to a trace) {
      if (we are coming from a trace &&
          this is a direct logical jump) {
        link the two traces together;
      }
      next_logical_trace = lbb->trace;
    } else { /* lbb not a trace */
      /* should we build a trace? */
      if (lbb->count > hot_threshold) {
        set flags to start DynamoRIO tracing;
    }
  }
}
```

When `logical_jump` is called during trace building mode with a direct or indirect logical jump, we signal DynamoRIO to finish building the trace. Thus each trace is headed by the target of a logical control transfer and ends with another logical control transfer. When DynamoRIO installs the trace, the trace is associated with the (*logical PC*, *native PC*) pair in the logical basic block table.

The first time in `logical_jump` for a given (*logical PC*, *native PC*) abstract PC, the pair will not be found in the "logical basic block table", and an entry will be added.

After the same abstract PC has been seen `hot_threshold` times, from direct or indirect logical jumps, we set flags to signal DynamoRIO's top level loop that we should start recording a trace, keyed by the (*logical PC*, *native PC*) pair. While trace building is active, DynamoRIO collects each basic block encountered into a contiguous list of instructions (this is regular DynamoRIO trace collection).

The next time `logical_jump` encounters an abstract PC for which a trace exists, it sets `next_logical_trace` (a field in the thread-local `dcontext` structure), and when control returns to the exit stub, the exit stub jumps directly to that target logical trace, without returning to DynamoRIO's dispatch loop.

Finally, if a logical basic block entry has a corresponding trace, *and* the `from-data` supplied by DynamoRIO indicates that we are following a direct logical control transfer from another trace, we *link* the two traces.

Linking two traces involves replacing the `jmp exit_stub` instruction in Figure 2 with a jump directly to the target logical trace fragment.

7 Trace Optimization

By organizing traces by the logical and native PC's, instead of the native PC's that regular DynamoRIO uses, we improve performance so that the DynamoRIO system has little overhead compared to running without DynamoRIO. In some cases, we even improve performance over native (not under DynamoRIO) execution.

We then apply relatively simple but aggressive constant propagation to the recorded traces.

7.1 Constant Propagation

We apply standard constant propagation, made more challenging by the fact that we are doing it for the X86 instruction set. Crucial to the success of constant propagation is being able to fold references into immutable memory to constants. As mentioned earlier, we use `set_region_immutable` calls from the interpreter to establish constant memory regions. Furthermore, the `*trace_constant*` calls register addresses whose contents we can rely on at the start of any given trace.

7.2 Call-Return Matching

DynamoRIO performs expensive checking when a return is encountered, because a return is basically an indirect branch. If static analysis of the trace reveals that the return matches a call site in the trace, then the return overhead can be elimintated. By removing some jumps, call-return matching enables other optimizations.

7.3 Dead Code Elimination

Constant propagation produces dead code, such as from storing no longer needed temporary values or from statically dispatching conditionals, and DynamoRIO collects dead code during its optimization.

8 Experimental Results

Figure 3 shows the running times of six benchmarks, normalized to the execution time on TinyVM without DynamoRIO.

We see that running under regular Dynamo slows down the application significantly (by almost a factor of two for some of the benchmarks). While adding logical tracing recovers some performance (fib-iter, sieve), it is clear that we need both logical tracing and optimization.

Fragment	Size	Time	Exits
1	1201	30.5%	S1: 66.2% S7: 6.8% S10: 27.0%
2	1849	6.1%	S15: 100.0%
3	1909	6.1%	S7: 2.4% S16: 97.6%
4	366	5.9%	S1: 18.2% S2: 14.9% S3: 66.9%

Figure 4. Fragment Size and Exit Statistices, Sieve Benchmark, regular DynamoRIO

The primary reason that regular DynamoRIO loses so much performance can be gleaned from the results of DynamoRIO's profiling tools. Figure 4 shows the top four traces for a run of a Sieve of Eratosthenes finding the first 30,000 prime numbers. Not only are the traces relatively short (about 1KB), but the hottest trace exits from its first exit (a conditional, taking the non-trace branch) 66% of the time.

Figure 5 shows the same profiling information for the application running under DynamoRIO with logical PC tracking. We see much longer traces, and the hottest trace exits at the end 97% of the time.

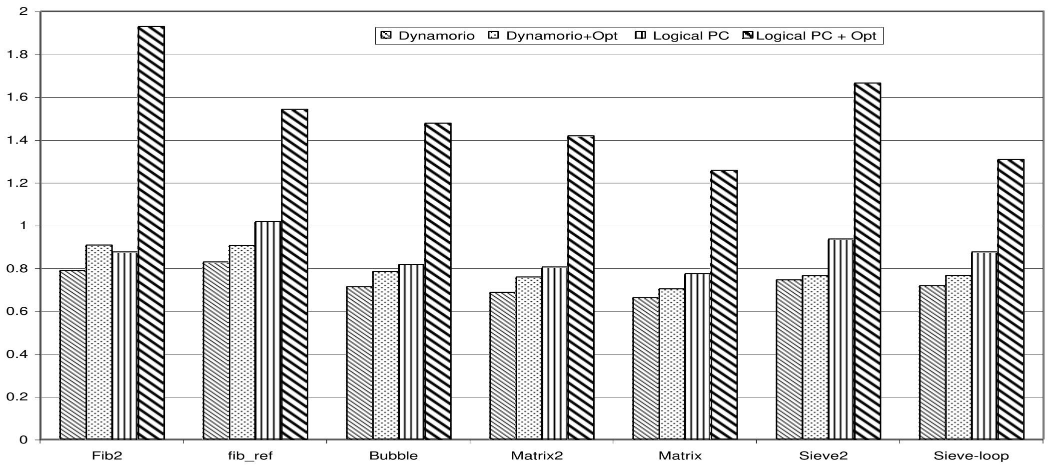


Figure 3. Running Times in Four Configurations, Normalized against Native (Without DynamoRIO)

Fragment	Size	Time	Exits
1	25255	48.1%	S34: 2.4% S242: 97.6%
2	36301	29.0%	S347: 100.0%
3	18834	15.0%	S161: 10.8% S178: 89.2%
4	10119	7.1%	S97: 100.0%

Figure 5. Fragment Size and Exit Statistices, Sieve Benchmark, DynamoRIO and Logical PC Tracing

In the fib-recurse benchmark, we actually see a slowdown when running with logical tracing instead of regular DynamoRIO. This is because of a large number of returns, which cause a lookup in our logical basic block table.

Figure 6 compares the contributions of the different optimizations we apply. It is surprising that constant propagation by itself does not contribute much. We get the single biggest improvement from call-return matching.

9 Related Work

Moore and Leach [27] describe writing threaded interpreters. Piumarta and Riccardi [29] go further with dynamically generated bytecode sequences.

Dynamic compilation has proven essential for efficient implementation of high-level languages [13, 1]. Some just-in-time compilers perform profiling to identify which methods to spend more optimization time on [20]. The Jalapeño Java virtual machine [3, 23] utilizes idle processors in an SMP system to optimize code at runtime. Jalapeño optimizes all code at an initial low level of optimization, embedding profiling information that is used to trigger reoptimization of frequently executed code at higher levels. Self [19] uses a similar adaptive optimization scheme.

Staged dynamic compilers postpone a portion of compilation until runtime, when code can be specialized based on runtime values [11, 17, 24, 25, 16]. These systems usually focus on spending as little time as possible in the dynamic compiler, performing extensive offline pre-computations to avoid needing any intermediate representation at runtime.

API-less dynamic optimization systems include Dynamo [4] for PA-RISC; Wiggins/Redstone [12], which employs program counter sampling to form traces that are specialized for a particular Alpha machine; and Mojo [7], which targets Windows NT running on IA-32, but has no available information beyond the basic infrastructure of the system. Kistler [21] proposes “continuous program optimization” that involves operating system re-design to support adaptive dynamic optimization.

Hardware dynamic optimization of the instruction stream is performed in superscalar processors. The Trace Cache [31] allows such optimizations to be performed off of the critical path.

Dynamic translation systems resemble dynamic optimizers in that they cache native translations of frequently executed code. Domains include instruction set emulation [9, 15] and binary compatibility [8, 22]. Recent dynamic translation systems such as UQDBT [33] and Dynamite [30] separate the source and target architectures to create extensible systems that can be re-targeted.

Dynamic instrumentation can be used to build runtime code analyzers and, to some degree, runtime code modifiers. Both Dyninst [6] and Vulcan [32] can insert code into running processes. It is based on dynamic instrumentation technology [18] developed as part of the Paradyn Parallel Performance Tools project [26].

Other related fields include link-time optimization [28, 10] and low-overhead profiling [2, 14].

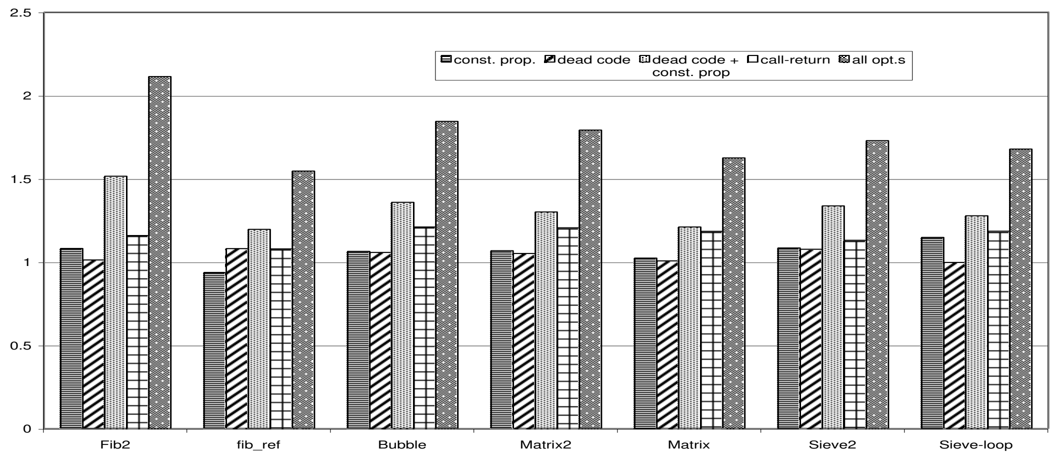


Figure 6. Contributions of Separate Optimizations, Normalized to DynamoRIO with Logical Tracing

10 Future Work

The research presented in this paper is a proof-of-concept for the idea of applying general purpose dynamic optimization techniques and native dynamic partial evaluation to interpreters.

We are currently working on applying this version of DynamoRIO to "real" interpreters, including OCAML and Kaffe. OCAML is a well-implemented threaded interpreter for a variant of the ML language. Kaffe is a very slow implementation, using recursive tree walking, of Java; Kaffe's interpreter can afford to be extremely slow, because the implementation also includes a reasonable JIT compiler. We expect very good results from applying our technology to OCAML. For Kaffe, we are generalizing the DynamoRIO-interpreter API to allow for multiple "logical PC" like values and to allow those values to be on the stack (to handle recursive interpretation).

There are also a number of straightforward engineering improvements that should improve the performance of the system.

We do not handle long loops within single bytecode implementations – they are unrolled in place. Currently we simply specify a maximum number of blocks per trace, and truncate the trace if we hit the maximum. We would like to avoid this sort of loop unrolling in the first place, and translate back jumps as jumps.

We are investigating adding more sophisticated alias analysis in order to enable constant propagation to track constant values through memory.

11 Conclusion

There will always be interpreted languages, and only some implementation efforts can afford the time and money to produce a native compiler.

We present a novel approach to improving the performance of interpreters. The interpreters do not have to be written in a particular style; they need only annotatations identifying a logical PC and regions of immutable memory (particularly the program representation). DynamoRIO then performs trace recording and aggressive partial evaluation of the X86 code to produce substantially optimized traces of the running interpreter. We have applied these techniques to simple interpreters and shown that this approach can provide substantial speedups with minimal instrumentation from the interpreter writer. We are currently working on applying these techniques to more sophisticated interpreters.

12 References

[1] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *Proceedings of the SIGPLAN'98 Conference on Programming Language Design and Implementation*, June 1998.

[2] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *16th ACM Symposium on Operating System Principles (SOSP '97)*, Oct. 1997.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, Oct. 2000.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent runtime optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, June 2000.

[5] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for

Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, Dec. 2001.

[6] B. R. Buck and J. Hollingsworth. An API for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[7] W.-K. Chen, S. Lerner, R. Chaiken, and D. M. Gillies. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, Dec. 2000.

[8] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2), Mar. 1998.

[9] R. F. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *SIGMETRICS*, 1994.

[10] R. Cohn and P. G. Lowney. Hot cold optimization of large Windows/NT applications. In *29th Annual International Symposium on Microarchitecture (MICRO '96)*, Dec. 1996.

[11] C. Consel and F. Nöel. A general approach for run-time specialization and its application to C. In *ACM Symposium on Principles of Programming Languages (POPL '96)*, Jan. 1996.

[12] D. Deaver, R. Gorton, and N. Rubin. Wiggins/Restone: An on-line program specializer. In *Proceedings of Hot Chips 11*, Aug. 1999.

[13] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *ACM Symposium on Principles of Programming Languages (POPL '84)*, Jan. 1984.

[14] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00)*, Oct. 2000.

[15] K. Ebcioglu and E. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *24th Annual International Symposium on Microarchitecture (ISCA '97)*, June 1997.

[16] E. Feigin. *A Case for Automatic Run-Time Code Optimization*. Senior thesis, Harvard College, Division of Engineering and Applied Sciences, Apr. 1999.

[17] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*, May 1999.

[18] J. Hollingsworth, B. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Proceedings of the 1994 Scalable High-Performance Computing Conference*, May 1994.

[19] U. Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University, 1994.

[20] The Java HotSpot performance engine architecture.

[21] T. Kistler and M. Franz. Continuous program optimization: Design and evaluation. *IEEE Transactions on Computers*, 50(6), June 2001.

[22] A. Klaiber. The technology behind Crusoe processors. Transmeta Corporation, Jan. 2000.

[23] C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8), Mar. 2001.

[24] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, May 1996.

[25] M. Leone and R. K. Dybvig. Dynamo: A staged compiler architecture for dynamic program optimization. Technical Report 490, Department of Computer Science, Indiana University, Sept. 1997.

[26] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 28(11):37–46, Nov. 1995.

[27] C. H. Moore and G. C. Leach. Forth – a language for interactive computing. Technical report, Mohasco Industries, Inc., Amsterdam, NY, 1970.

[28] R. Muth, S. Debray, S. Watterson, and K. D. Bosschere. alto : A link-time optimizer for the Compaq Alpha. *Software Practice and Experience*, 31:67–101, Jan. 2001.

[29] I. Piumarta and F. Riccardi. Optimizing direct-threaded code by selective inlining. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[30] A. Robinson. Why dynamic translation? Transitive Technologies Ltd., May 2001.

[31] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *29th Annual International Symposium on Microarchitecture (MICRO '96)*, Dec. 1996.

[32] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, Apr. 2001.

[33] D. Ung and C. Cifuentes. Machine-adaptable dynamic binary translation. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, Jan. 2000.

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.

IVME'03, June 12, 2003, San Diego, California, USA.

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.

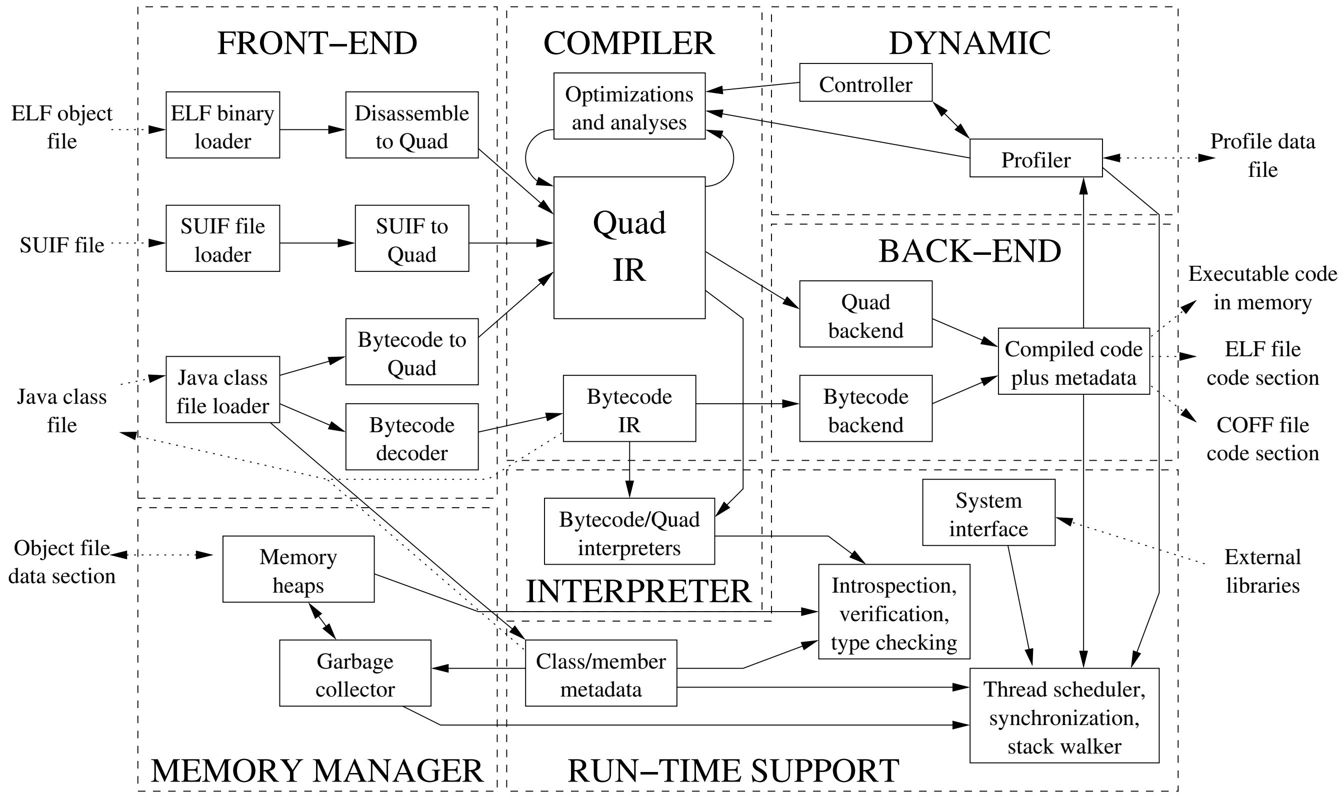


Figure 1. Overview of the Joeq system. Arrows between blocks signify either the flow of data between components, or the fact that one component uses another component.

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 object files, libraries, and executable images in the popular ELF format[27]. The front-end also includes an intelligent 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 as high-level as Java to as low-level as assembly 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.

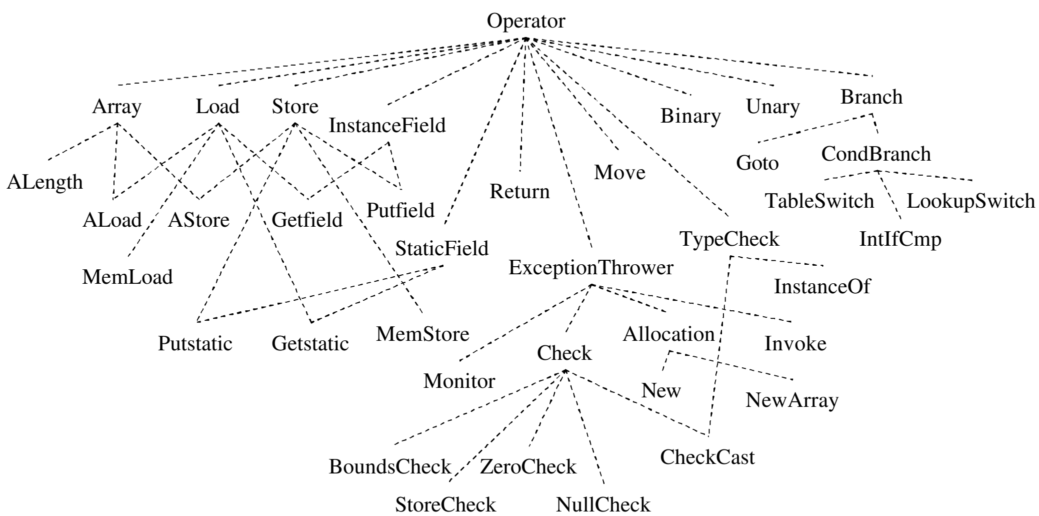


Figure 2. Overview of the operator hierarchy. Visitors visit in most-specific to least-specific order.

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 without synchronization[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 references into a heap.

[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.

Joeq supports the simultaneous usage 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 the 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 a 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 to 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 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 you to "append" 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 in Joeq, 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 and 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.

Engineering a Customizable Intermediate Representation

K. Palacz J. Baker C. Flack C. Grothoff H. Yamauchi J. Vitek
S[3] lab, Department of Computer Sciences, Purdue University

ABSTRACT

The Ovm framework is a set of tools and components for building language runtimes. We present the intermediate representation and software design patterns used throughout the framework. One of the main themes in this work has been to support experimentation with new linguistic constructs and implementation techniques. To this end, framework components were designed to be parametric with respect to the instruction set on which they operate. We argue that our approach eases the task of writing new components without sacrificing efficiency.

1. INTRODUCTION

The goal of the Ovm project is to deliver an open source framework for the purpose of building language runtimes. Ovm (pronounced *ovum*) is a toolkit that provides the basic components of a virtual machine. These components can be specialized and assembled into an *Ovm configuration* customized for a particular problem domain. The framework is designed so as to be able to support different object models, while our first emphasis is on supporting Java, we have plans to provide a C# personality. In Ovm, a configuration defines a new runtime environment, for example we have developped Real-Time Java configuration which yields a virtual machine compliant with the Real-time Specification for Java. The Ovm code base is written in a combination of Java and AspectJ [12]. Even though the framework is not complete, it already contains some 2000 classes and interfaces, accounting for more than 150K LOC.

The framework is composed of a number of tools and subsystems (interpreter, verifier, compilers) which must be kept operational whenever the instruction set on which they operate is modified or extended. One of the earliest design decisions in the project was to attempt to parameterize framework components by their instruction set using an intermediate representation, OvmIR, which can be easily adapted to a particular configuration. We consider components to be parametric if they can be applied to different instruction sets with minimal changes to their implementation. In Ovm this form of parameterization is achieved by a combination of a reflective instruction set specification and consistent use of software design patterns. Instructions in the OvmIR are represented by data structures that detail their semantics and that can be inspected introspectively by the framework components. The OvmIR specification uses a type hierarchy to classify instructions according to their behavior. This classification is used by the tools to reduce the amount of redundant code written for processing OvmIR.

In this paper we describe the design of the Ovm intermediate representation and architecture of software components that manipulate it. The OvmIR is an extended version of Java bytecode. Extensions include operations to express unsafe operations and other Ovm specific primitives. This representation is flexible in the sense that it is easy to define new operations or modify the semantics of existing ones. In order to write tools that may adjust to changes in the instruction set we use a single specification for the entire Ovm tool chain, from static analysis tools to interpreter generation and run-time code generation We hasten to emphasize that the current IR is but one point in the design space. Even with a completely different IR, many of the fundamental design choices and decisions that we present in this paper are likely to still be valid. While the OvmIR is instrumental in generating an interpreter and native code, it is not unique in that respect [11]. We have not investigated optimizations of the generated logic. Tools such as vmgen [3] are complementary, and it would be interesting to see if they could be used in our setting.

The paper is structured as follows. Sec. 2 describes how the OvmIR is specified. Sec. 3 introduces the design patterns that are used to operate on the IR. Sec. 4 describes components that have been build using the OvmIR. Sec. 5 describes the use of OvmIR-to-OvmIR transformations in allowing necessary non-Java semantics to be expressed in Java source. Sec. 6 gives a simple example of extending the IR. We conclude with related and future work.

IVME'03, June 12, 2003, San Diego, California, USA.

2. A REFLECTIVE IR SPECIFICATION

The intermediate representation currently in use in the Ovm framework is a stack-based high-level intermediate language that closely mirrors Java bytecode [13]. The choice to base OvmIR on bytecode was made for pragmatic reasons. Java bytecode is a compact and executable representation with a well-specified semantics. This entails a shorter learning curve for working with the framework components and a natural path towards an interpreter. Of course, there are also drawbacks. Bytecode is not fully typed; thus, it is necessary to perform static analysis to recover type information. Other drawbacks include relative addressing and the complexity of dealing with the stack when performing code transformations. For the set of applications and tools that we have been targeting so far, these drawbacks have turned out to be minor.

The OvmIR specification is expressed as a set of classes, one class per instruction. This has the advantage of a close correspondence between the textual form of the specification and the internal data structures used within the VM and tools operating on this IR. Furthermore, this permits writing code that operates on the IR by reflection. For example, the abstract interpreter handles many instructions using introspection on their definition. It is possible to programatically alter the instruction set to add new operations or modify the semantics of existing ones. In many cases, such changes require minimal changes on tools operating on the IR.

2.1 Abstract machine model

The semantics of instructions are defined with respect to an implicit abstract machine. Currently, we use a stack-based machine similar to the one expected by Java bytecode. Thus, every operation may read and write values from local variables, push and pop variables from the stack, consume values from the instruction stream, jump to a set of offsets, throw exceptions, have evaluation side effects or access compile-time constant values contained in a constant pool. The OvmIR specification consists of the definition of a number of *instructions*, *values* and so-called *value sources*. While instructions are self-explanatory, it is worth discussing values and value sources. A value object is the representation of a concrete value that will be manipulated at runtime by the virtual machine. The `Value` class is used to represent runtime values and their types. Typing constraints on the values consumed or produced by an instruction are expressed by using subclasses of the `Value` class (*e.g.* integer values are represented by instances of `IntValue`). Since it is not always possible or practical to model all the constraints on values using inheritance, further typing constraints can be expressed as annotations on the value type. For example, `CPIndexValue` is a subclass of `IntValue` used to denote constant pool indices and declares a field indicating the type of constant pool entry, such as field reference, type name reference etc.

2.1.1 Instructions

Each IR instruction corresponds to a subclass of the `Instruction` class. Analyses written for the framework use the flyweight pattern [6] to avoid the need for multiple instances of the same instruction class. The semantic specification of every instruction is provided in the parameterless constructor of the respective class. Each instruction must at least implement the methods `size()`, which returns the size of the instruction in bytes, and `getOpcode()`, which returns a numeric instruction identifier. Further behavior is added only by subclasses for which the behavior is meaningful. Thus, code manipulating the OvmIR can rely on the type checker to prevent errors such as trying to use a constant pool index as a jump target.

The semantics of instructions are characterized by the following arrays of abstract values that are members of `Instruction`:

`streamIns`	values consumed from instruction stream
`stackIns`	values consumed from the stack
`stackOuts`	values produced on the stack
`evals`	expressions evaluated for side effects
`jumpTarget`	target program counter value (subtypes of `FlowChange`)
`controlValue`	value used to control branching (subtpyes of `ConditionalJump`)

Table 1: Fields that specify the input-output behavior of each instruction.

Furthermore, instructions that implement the `Throwing` interface define an array of exceptions that may be thrown as a side effect of executing the instruction.

As an example of one of simple instruction specifications, consider the definition of the `SWAP` instruction which exchanges the two topmost stack values. The code for `SWAP` is shown in Fig. 2. `SWAP` is a *concrete instruction* in that it corresponds to an actual operation in the instruction set. It is defined as a subclass of the abstract class `StackManipulation` which in turn extends `Instruction`. The hierarchy permits us to reuse code between related instructions by inheritance. Extensive examples for this are given in sections 3 and 4. The instruction hierarchy can also be used to classify instructions into various categories.

```
class SWAP extends StackManipulation {{
    stackIns = new Value[]{ new Value(), new Value()};
    stackOuts = new Value[]{ stackIns[1], stackIns[0]};
}}
```

Figure 2: Specification of the `SWAP` instruction.

The specification of `SWAP` describes that the instruction consumes two values from the stack and pushes two values back onto the stack. The fact that the output values are identical to the input values, with the exception of their order, is represented by pointer equality of the `Value` objects.

The core of the instruction hierarchy used within Ovm is shown in Fig. 1. The given hierarchy reflects properties of the Java bytecode instruction set and is based on pragmatic considerations, not on any systematic analysis of features of all conceivable instructions of stack machines. It turns out to be impossible to model all the features of bytecode using

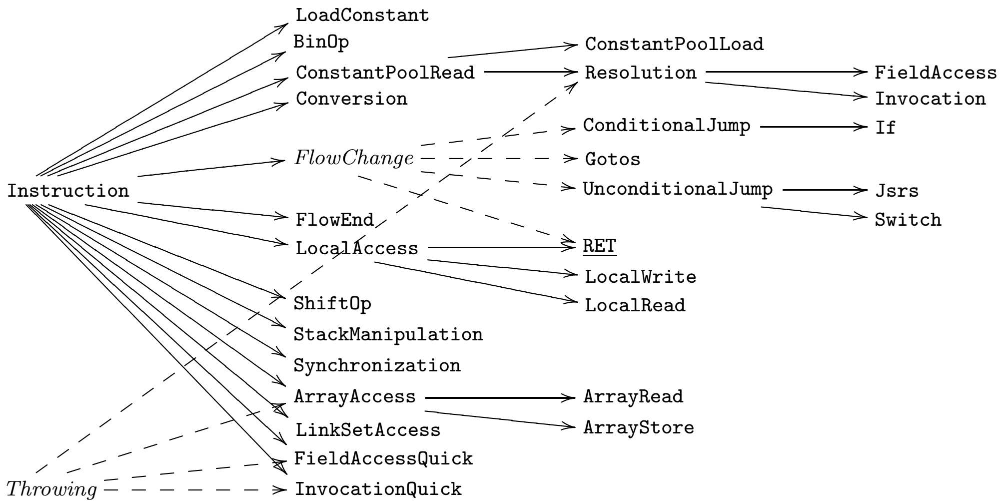


Figure 1: The instruction hierarchy. The OvmIR consists of 237 concrete instructions (RET is the only one shown here) that account for 3400 lines of code.

single inheritance. For example, `FlowChange` is an interface and is implemented by the concrete instruction `RET` (return) that inherits from `LocalAccess`. Another example is the `Throwing` interface which is implemented by all instructions that can throw exceptions.

Fig. 3 gives a simple example of reflective programming. The method initializes a lookup table, the static variable `instructionSet`, with instances of every concrete instruction. The code assumes that all instructions are inner classes of the `Instruction` class and that they have a constructor with no arguments. Only concrete instructions are instantiated.

```
void makeInstructionSet() {
   Class[] inner =
           Instruction.class.getDeclaredClasses();
   for ( int i = 0; i < inner.length; i++)
      if ( Instruction.class.isAssignableFrom( inner[i])
      && !Modifier.isAbstract( inner[i].getModifiers())){
        Instruction inst =
            (Instruction) inner[i].newInstance();
        instructionSet[inst.getOpcode()] = inst; } }
```

Figure 3: Reflective instruction set initialization.

2.1.2 Values and Value Sources

Another way to associate output values with input values in the specification is to add *value sources* to values. Each `Value` object contains a reference to an instance of the `ValueSource` interface. A variety of value source implementations are provided, such as binary arithmetic and logical expressions, and routine invocation expressions. Since value sources are also represented as Java data structures, they can be analyzed introspectively. Value sources can reference existing values, and hence the instruction specification forms an expression tree consisting of a sequence of expressions represented by the `evals` array followed by a sequence of expressions in the `stackOuts` array. The evaluation of both sequences can introduce side effects. For example, `IADD`, which has no side effects and pushes the sum of two integers on the stack could be defined as follows:[1]

```
class IADD extends BinOp {{
    super(IADD);
    stackIns = new Value[] {
        new IntValue(), new IntValue()};
    evals = new Value[] {};
    stackOuts = new Value[] {
        new IntValue(new BinExp(stackIns[0],
            "+",
            stackIns[1])) }; }}
```

Individual instruction definitions are assumed to be fragments of code with single input and single output *modulo exceptions.*

The IR supports flow control private to an instruction definition, for example:

```
class FCMP extends Instruction {{
    stackIns = new Value[] {new FloatValue(),
            new FloatValue()};
    IfExp e = new IfExp(new CondExp
                (stackIns[1], ">", stackIns[0]),
            ONE,
            new IntValue(new IfExp
                (new CondExp(stackIns[1],
                "==", stackIns[0]),
            ZERO, MINUSONE)));
    stackOuts = new Value[]  new IntValue(e) ; }}
```

The IR can also specify control flow instructions that affect the program counter. In this case instructions declare their control value and jump targets, for example:

```
class IFNE extends ConditionalJump {{
    stackIns = new Value[] { new IntValue() };
    streamIns = new IntValue[]{ new PCValue() };
    jumpTarget = streamIns[0];
    controlValue = new IntValue
        (new CondExp(stackIns[0], "!=", stackIns[1])); }}
```

[1] In Ovm the specification of IADD is further abstracted to:
`class IADD extends BinOp{IADD(){super(IADD,"+",intFactory);}}`

Note that instead of using pointer identity to bind stack output values to input values, one can use a separate kind of value source to denote that a value comes from a stack slot. However, the presented approach results in more concise definitions.

The `InstructionBuffer` class maintains a notion of the current program counter as well as the definitions of constants referenced from the bytecode stream. Because of this design, instruction objects can retrieve and interpret their immediate operands without any state of their own. For example, the concrete instruction `GETFIELD` subclasses the abstract class `FieldAccess`. `FieldAccess` provides a method `getSelector(InstructionBuffer)` to return information about the name and type of the field being accessed. The state required by the method is encapsulated in the instruction buffer argument. This design follows the flyweight pattern [6], allowing the `InstructionSet` class to hold a single instance of each concrete instruction.

The OvmIR specification contains approximately 3400 lines of code and 237 concrete instruction definitions which results in about 14 lines of code per definition. The line count includes abstract classes and supporting methods. Easily recognizable syntactic conventions (class and constructor declarations) account for approximately 4 lines per instruction hence it requires about 10 lines of nontrivial code to specify an instruction.

Example. To demonstrate the expressiveness of a reflective specification, consider the implementation of ovmp[2], the Ovm counterpart to the javap class file disassambler. Ovmp prints the mnemonic of each opcode and, for instructions such as branch or loads that have immediate arguments, the value of those argument in the format specified by the instruction semantics. The main loop is written as follows.

```
while (ib.remaining() > 0) {
    Instruction i = ib.get();
    print(i.getName());
    if (isStreamReader(i))
        iprinter.visitAppropriate(ib.currentOpcode()); }
```

A runabout dispatches to the proper print method. Because the instruction hierarchy lacks a type to characterize all iStream readers, the implementation takes advantage of the reflective IR specification. It filters calls to the runabout based on the characteristics of instructions. This is done by the method `iStreamReader()` which return true if an instruction has immediate arguments. It is written as follows.

```
    iStreamReader(Instruction i) {
        return i.istreamIns.length != 0; }
```

The argument printer runabout has sixteen methods and is a total of 120 lines long. The entire ovmp has 520 lines (out of which 60 are comments).

[2]This example is due to Jacques Thomas who implemented ovmp.

3. DESIGN PATTERNS

The software architecture of the tools and components of the Ovm framework has evolved over time. This section describes this evolution and the motivations behind the changes. Originally, the instruction objects used dedicated methods to perform abstract execution of bytecode. The dedicated methods accessed the state information of the current method via a helper object which was provided to the constructor of every instruction object and stored in a field of the instruction. This created problems in that the instruction objects could not be used concurrently by multiple threads operating on different methods.

The use of dedicated methods also had the disadvantage that every additional analysis or processing required changes to each instruction class. Thus, each of the instructions was extended with an accept method and various analyses were written as visitors operating on the instructions. In order to make code-factoring easier, the instructions were arranged in a hierarchy (see Fig. 1) that reflects commonalities between the instructions. Convenience methods implemented by the instructions were factored into common superclasses. An example of such a method is `getCPIndex`, a method that returns the index into the constant pool for every instruction that accesses the constant pool.

Runabout	# visit methods
CloneInstructionVisitor	202
ClassCleaner	2
LinearPassController	1
AbstractInterpreter	27
ControlFlowInterpreter	8
BasicBlockJ2CTranslator	11
TypeNameClosure	10
SimpleJitVisitor	134
JamitConstraintGenerator	8
ZeroCFA	18
ConstraintGenerator	9
MaxStackHeightInference	30

Table 2: List of the Runabouts working on the instructions.

The visitors that implement the various analyses are also able to take advantage of the instruction hierarchy; the visit methods can be refactored using the hierarchical visitor pattern [10]. For example, our access modifier inference tool does not need to distinguish between Java's four field access operations (`GETFIELD`, `PUTFIELD`, `GETSTATIC`, `PUTSTATIC`). Using the hierarchical visitor pattern, Jamit only needs to implement the visit method for the abstract `FieldAccess` instruction class. In order to make the hierarchical visitor pattern work, a helper method that indirects calls from `visit(PUTSTATIC i)` to `visit(FieldAccess i)` is required (see code on the left in figure 4).

Writing this indirection code, while conceptually trivial, turns out to be cumbersome over time. Each time the instruction hierarchy evolved, the base-classes of the visitors needed to be rewritten. With over 200 instruction classes it became difficult to track changes in the hierarchy. Probably worse,

```
class FieldAccess {
    void accept(Visitor v) {
        v.visit(this); } }

class GETFIELD extends FieldAccess {
    void accept(Visitor v) {
        v.visit(this); } }

class HierarchicalVisitor extends Visitor {
    void visit(FieldAccess i) {
        ...
    }
    void visit(GETFIELD i) {
        visit((FieldAccess)i); } }

class Main {
    static void main() {
        Visitor v = new HierarchicalVisitor();
        Instruction i = new GETFIELD();
        inst.accept(v); } }
```

```
class FieldAccess { }

class GETFIELD extends FieldAccess { }

class MyRunabout extends Runabout {
    void visit(FieldAccess i) {
        ...  } }

class Main {
    static void main() {
        Runabout r = new MyRunabout();
        Instruction i = new GETFIELD();
        r.visitAppropriate(inst); } }
```

Figure 4: Hierarchical Visitor vs. Runabout

the instruction set needed to be expanded to support operations that are not part of the JVM standard. The use of the visitor pattern required that every analysis supplied visit methods for *all* instructions. Thus, every change in the hierarchy of the instruction set required updates to several visitors. Considering that one of the requirements for OvmIR is that the instruction set is customizable, this was not practical.

The problem was solved by replacing the use of visitors with the Runabout pattern [7]. Runabouts declare visit methods just like visitors, but instead of doing double-dispatch with `accept` methods in the instruction objects, the appropriate visit methods are found by reflection and invoked by dynamically generated and loaded helper classes. The result of this final refactoring was that hundreds of `accept` methods were removed from the instruction objects and hundreds of `visit` methods that were either abstract (visitor interface), empty (default base class) or indirecting to other visit methods (hierarchical visitor) became obsolete. The Runabout code on the right side of figure 4 is the equivalent to the visitor code

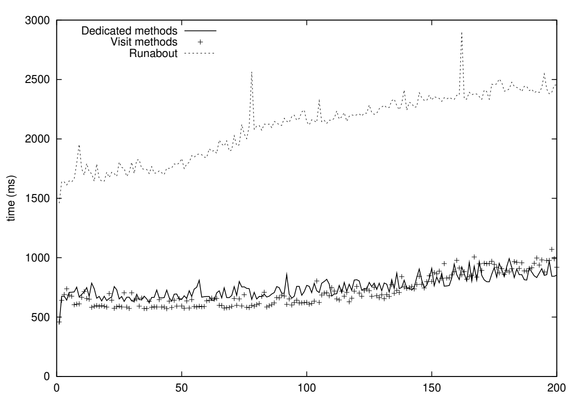


Figure 5: Runtime comparison of dedicated methods, visitors and the Runabout in Micro-benchmark for 10.000.000 iterations on SUN JDK 1.4.1, P-III 1 GHZ, Linux 2.4.18. The time shown is for the dispatch only.

on the left side. As the example shows, using the Runabout eliminates the need for the `accept` methods and the code for the code for the hierarchical indirection. The performance impact of the dispatch with the Runabout on the framework was minor (a few percent depending on the analysis), confirming expectations from the microbenchmarks (Fig. 5). Table 2 shows the various classes in the Ovm framework that are Runabouts visiting the instructions. The table also lists the number of visit methods that each of these classes implements. The total number of instruction classes in Ovm is 273 (237 of these are concrete instructions, but some tools only support the 201 instructions of the Java VM specification [13]).

4. COMPONENTS BUILT AROUND OVM

Various tools have been built around the OvmIR. Tools that are used by the VM itself include a bytecode verifier, a simple JIT compiler, an interpreter generator, a bytecode-to-C++ translator, and the OvmIR transformation infrastructure. The analysis framework has been used to implement Kacheck/J [8], a tool to infer confined types, Jamit, an access modifier inference tool and Hitsuji, a tool that performs control-flow analysis (for example, 0-CFA [16] or type-safe method inlining [15]).

4.1 Interpreter Generation

The Ovm framework includes a Java bytecode interpreter written in C. The interpreter is automatically generated from the instruction specification which ensures that the part of the system written in C is synchronized with the rest of the codebase. In particular, this ensures that the C interpreter actually operates on the same IR that is produced by tools written in Java without forcing the interpreter to consult `Instruction` objects at runtime.

We chose to implement only a subset of Java bytecode instructions in C code. The more involved instructions that require cooperation with the JVM runtime such as, for example, the resolution of symbolic member references or memory allocation, are reduced to invocations of methods of the runtime. Because instruction specifications can be viewed as an abstract syntax tree format for a subset of C expressions, the interpreter generator is fairly simple. Conceptually, each instruction is compiled to a `case` clause that copies immediate and stack operands into local variables, computes the result, and pushes it on the stack. For instance, `IADD` would be compiled to the following code:

```
case INSTR_IADD: {
    jint stack_in_0 = POP().jint;
    jint stack_in_1 = POP().jint;
    jint stack_out_0 = (stack_in_1 + stack_in_0);
    PUSH_P(stack_out_0);
    INCPC(1);
    NEXT_INSTRUCTION;
}
```

Currently, our interpreter loop is not using a switch statement; instead, we generate a threaded interpreter [3] using language extensions of the GCC compiler. We had to change very few lines of code in the interpreter generator to switch between these two techniques.

4.2 Ahead of Time Compilation

We are currently using the same instruction specification to implement a bytecode to C++ compiler, J2c. This compiler does not use high-level C++ features such as member functions and run-time type identification, but treats C++ as a portable assembly language with exception-handling support. The compiler may be configured for either conservative or precise garbage collection.

Although the instruction specification forms the basis of J2c's intermediate representation, J2c extends the specification in a few ways. For instance, the OvmIR differentiates between explicit method invocations and traps into the runtime. J2c translates both types of method calls into a single form so that they can be devirtualized and inlined uniformly. J2c also defines a `ValueSource` type corresponding to `MULTIANEWARRAY`'s operands. Such a type makes sense for occurrences of the `MULTIANEWARRAY` instruction, but does not fit into the singleton framework, since `MULTIANEWARRAY` takes a variable number of arguments.

4.3 Just in Time Compilation

We have implemented a just-in-time compiler, SimpleJIT, which converts the OvmIR into Intel x86 native code in one linear pass. SimpleJIT is not intended to be an optimizing compiler, but rather a basic fast compiler like the Jikes RVM baseline compiler [2]. SimpleJIT emulates the operand stack in the native stack frame. Each concrete instruction is covered by a visit method in the compiler. Fig. 6 shows the visit method for the family of instructions pushing an integer on the stack. The code exploits the instruction hierarchy to cover nine concrete instructions.

```
public void visit( IConstantLoad instruction) {
    asm.pushI32( instruction.getValue( this));
}
```

Figure 6: The visit method for `ICONST_0`, `ICONST_1`, `ICONST_2`, `ICONST_3`, `ICONST_4`, `ICONST_5`, `ICONST_M1`, `BIPUSH`, and `SIPUSH`.

4.4 Static Analysis

Ovm includes a number of classes for performing static program analysis. Implementing a static analysis requires the implementation of a set of visit methods for the instructions that are relevant for the analysis. Furthermore, the analysis needs to select an iterator that specifies the traversal over the code. Ovm provides two basic templates. The first one performs a linear pass and that will visit every instruction once. The second runs a fixpoint iteration that is coupled with a customizable abstract interpreter. For customization, the analysis defines the level of abstraction by providing the set of abstract values. Defining the abstract values requires code that provides tests for value equality and merging of abstract values at join points. The default set of abstract values in Ovm corresponds to the set used by a Java bytecode verifier and distinguishes between four basic primitive types (int, float, double, long), the null reference, jump targets (JSR), initialized objects and uninitialized objects. The default execution model also corresponds to the abstract execution performed by a bytecode verifier.

A typical analysis uses this basic form of abstract execution and interposes calls to analysis specific visitors that inspect the state of the abstract interpreter for information relevant to the particular analysis. If the value abstractions are extended to better match the different abstract domain of a given analysis, visit methods of the abstract interpreter must be overriden to ensure proper handling of the new values. Examples of existing extensions of the abstract value set in Ovm include the addition of a special value for the `this` reference in Kacheck/J [8] and the use of a flow sensitive type sets for the implementation of 0-CFA [16].

The instruction specification isolates the analysis code from irrelevant changes in the OvmIR. Often a single visit method covers the behavior of multiple instructions that are equivalent from the point of view of the analysis. For example, the default abstract interpreter has generic code for instructions that merely perform basic operations such as moves or arithmetic. Thus, instructions that fall into these categories can be added trivially without changing the abstract interpreter.

Example. Fig. 7 gives a simple example of a `visit` method that overrides the default behavior of the abstract interpreter for the `NEW` instruction. In the context of 0-CFA, a `NEW` instruction pushes a flow set on the stack that contains the type of the object being constructed. The visit method is located within the body of a runabout called within the fixpoint iteration. The method starts by querying the `NEW` instruction for the name of the class under construction. Since the instruction is a flyweight object, the runabout passes the current instruction buffer to the instruction so that it can retrieve the type name.

```
void visit( NEW instruction) {
    TypeName type = instruction.getClassName( ibuf);
    getFrame().push( valueFactory.makeSet( type));
}
```

Figure 7: The visit method for the `NEW` instruction.

The `valueFactory` object is an abstract value factory that creates the flow set which is then pushed onto the operand stack. The implementation of `getClassName` in the `NEW` instruction class is shown in Fig. 8. The method is written in terms of operation on the state of the instruction buffer and an auxiliary `getCPIndex` method which returns the constant pool index immediately following the opcode of the current instruction.

```
TypeName getClassName( InstructionBuffer ibuf) {
    int index = getCPIndex( ibuf));
    return ibuf.getConstantPool().getTypeNameAt( index);
}

int getCPIndex( InstructionBuffer ibuf) {
    return ibuf.getCode().getChar( ibuf.getPC() + 1);
}
```

Figure 8: The `NEW` flyweight instruction object extracts context dependent information from the instruction buffer.

Performance. The code needed to extract information from the abstract execution is typically small. For instance, the addition of the `this` pointer to the abstract value set for our confinement checker Kacheck/J [8] is specified in 160 lines of code. Flow sensitive types for the 0-CFA algorithm are implemented in 370 lines. The code for constraint generation in Kacheck/J is merely 660 lines; obtaining type set information for the 0-CFA takes 530 lines.

The use of the flyweight pattern for instructions is key for achieving high throughput. Kacheck/J, which can be run as a standalone application and performs what amounts to a slightly extended variant of bytecode verification, has competitive running times. Fig. 9 shows the running time of the analysis on a set of large benchmark programs. The graph plots the number of MB of bytecode (size of class files inclusive of constant pools) in the benchmark against the time required for the analysis proper (we did not include the time it takes to load the bytecode and parse the constant pools). This amounts to roughly 10MB per second, which appears competitive with tools written in C.

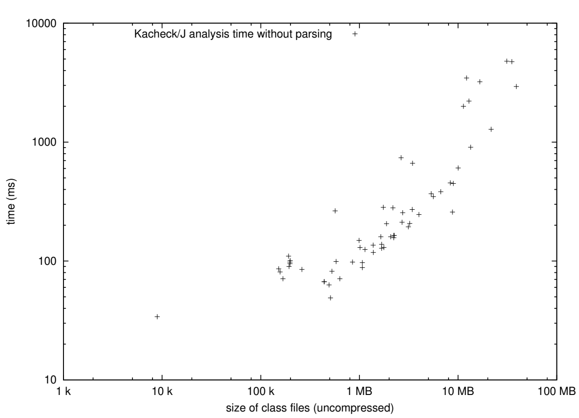


Figure 9: Performance of the analysis framework for Kacheck/J on a PIII-800 running Sun JDK 1.4.1.

4.5 Code Manipulation

Code manipulation of OvmIR is performed by `Editor` objects. Editors operate on instruction buffers and provide two abstractions, `Cursor`s and `Marker`s. Cursors are used for inserting instructions. Markers act as symbolic jump targets. Like the analyses, editing is typically performed by a visitor that iterates over the code in some application specific order. A transformation consists of a sequence of edit operations followed by a commit. The original code remains visible until the commit is performed.

Cursors have methods to create all of the concrete instructions. These methods often provide slightly higher-level abstractions than the actual instructions of the IR. For example, the cursor will emit appropriate code for a branch instruction, *i.e.* either a short branch or a sequence of instructions that use a combination of short branch and long jump. Similarly, an insertion operation like load constant will automatically choose the best instruction for the given value (such as `ICONST3` for 3, or `BIPUSH(42)` for 42). The required constant pool entries are also automatically generated.

The editor uses stateful instruction objects. The reason for this is that code generation requires context information, such as the location of a jump target, that is not always available before a commit. Correctness of the resulting code is not enforced. In fact, the cursor API allows the insertion of arbitrary sequences of bytes into the instruction stream.

Complex modifications such as code motion are possible within the framework, but can quickly become complicated to express since the client code is not provided with any automated mechanisms to deal with data-flow. Moving a single instruction such as `IADD` will always be difficult because the surrounding instructions that first prepare the operand stack and later process the result must also be moved. An SSA-based IR would make these dependencies more explicit. We are planning to provide a higher-level abstraction of the IR that will give client code a more SSA-like view.

5. IDIOM RECOGNITION

Because OvmIR is the input accepted by the interpreter, ahead-of-time compiler, and just-in-time compiler, a uniform approach is possible to the various unsafe or "magic" operations (*e.g.* memory dereferencing) that must be available to a VM implementation but cannot be expressed in pure Java. Most such operations are very short sequences of code, often heavily executed, so that it would not be practical to implement them with a heavyweight mechanism such as the Java Native Interface; in any case a single native library implementing the operations would not suffice, as the interpreter and compilers will use different techniques to realize the same operations.

We include in OvmIR a small set of opcodes for non-Java primitive operations that can, either singly or in combination, express the special operations Ovm needs. The interpreter and compilers simply need to support these opcodes as well. To make the extended semantics available at the level of Ovm's Java source, we do *idiom recognition*[9]: we

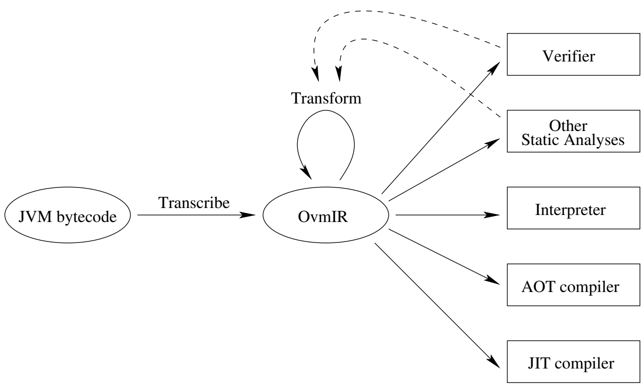


Figure 10: OvmIR is the common input to later processing and execution. Idiom recognition is performed iteratively on the OvmIR.

attach special semantics to certain idiomatic uses of standard Java constructs and, in an OvmIR-to-OvmIR transformation pass, replace those idioms with appropriate IR sequences. The transformation pass can be iterated to a fixed point, allowing new idioms to be defined in terms of existing ones. A report[4] more fully describes this aspect of Ovm; this section provides a brief overview. Fig. 10 illustrates the architecture.

By *tag idiom*, we denote an idiom that can be recognized unequivocally, such as implementing a specific marker interface or including a marker exception type in a `throws` clause. By *heuristic idiom*, we mean one that is detectable with some finite risk of false negatives or false positives, such as a sequence of related operations in some order (perhaps altered by javac before the recognizer sees it). We use *guarded idiom* to name the combination of those techniques, a heuristic idiom that is associated with a tag and is only recognized where the tag is used. Guarded idioms can be elaborate enough to express special operations at a usefully high level, while eliminating the risk of false positives and mitigating false negatives: if the tag is seen and the rest of the idiom is not recognized a warning can be given.

Ovm provides a small set of primitive tag idioms, based on marker interfaces and exceptions, with which other tag and guarded idioms can be defined. We define a new idiom by subclassing an existing tag and writing its IR transformation in terms of the `Editor` and `Cursor` interfaces. No other code—in particular, no compiler or interpreter code—needs to be touched to implement a new idiom as long as its semantics can be expressed with some sequence of existing

```
public static ObjectModel getObjectModel()
    throws InvisibleStitcher.PragmaStitchSingleton {
    return (ObjectModel)InvisibleStitcher
        .singletonFor( "ovm.core.domain.ObjectModel"); }
```

Figure 11: A stitched Abstract Factory method. It returnsa singleton of an abstract implementing class. The stitcher idiom transforms all call sites to constant-loads of the singleton.

OvmIR operations. The idiom in the following example has a definition that occupies 32 lines in a single source file.

Fig. 11 illustrates an idiom we use throughout to achieve configurability using the familiar Abstract Factory design pattern[6] without paying the indirection cost that might otherwise weigh against heavy use of this pattern in a VM.

6. CUSTOMIZING THE INSTRUCTION SET

The Ovm framework allows us to experiment with many aspects of the virtual machine design space, some of which require changes to the instruction set. Consider, for instance, a change to the VM internal data structures that merges per-class constant pools into a global shared data structure (as done in Jikes [2]). The current limit on constant pool indices (65535) must be overcome by extending the OvmIR with new instructions to load values from 32 bit constant pool indices.

To define the new instructions `LDC_DW` and `LDC_2D`, one must define compile-time constants for the new opcodes, define the new instructions, and add these new instructions to the instruction set. This amounts to adding roughly 20 lines of code. One must also teach the `Cursor` class used in bytecode rewriting how to generate the new instructions.

As shown in Fig. 12, the specification for `LDC_DW` is similar to that of `LDC_W`. The only difference between the two is the size of the integer constant in `streamIns`. This value is used by `ConstantPoolRead.getCPIndex` to decode the immediate operand statically and by the interpreter generator to choose the decoding macro to invoke at runtime.

Finally, one must check that the new instruction is supported by all visitors that may encounter the new opcode. For `LDC_DW`, the only place that requires a change is the `CloneInstructionVisitor` that needs to use the new method in the `Cursor` to clone the `LDC_DW` instruction. More elaborate additions of new instructions, especially if they can

```
public static class LDC_W extends ConstantPoolLoad { {
        CPIndexValue val = new CPIndexValue
            (CPIndexValue.CONSTANT_Any, TypeCodes.USHORT);
        streamIns = new IntValue[] val;
        stackOuts = new Value[] {new Value
            (new CPAccessExp(val)) };
        exceptions_ = new TypeName.Scalar[] {
            VIRTUALMACHINE_ERROR};
    }
    private final TypeName.Scalar[] exceptions_;
    public TypeName.Scalar[] getThrowables() {
        return exceptions_;
    } }
public static class LDC_DW extends LDC_W { {
        CPIndexValue val = new CPIndexValue
            (CPIndexValue.CONSTANT_Any, TypeCodes.UINT);
        streamIns = new IntValue[] val;
        stackOuts = new Value[] {
            new Value(new CPAccessExp(val)) } }
```

Figure 12: The specification of Java's `LDC_W` instruction, which loads a constant using a 16 bit index, and `LDC_DW`, which loads a constant using a 32 bit index.

not simply be modelled as subtypes of existing instructions, typically require slightly more extensive additions to visitors that need to handle the new instruction.

7. RELATED WORK

Interpreter generation has been employed by many systems, among them Java Virtual Machines (*e.g.* Hotspot) and Scheme interpreters ([11]). Recent refinements in interpreter implementation techniques [5] suggest that a sophisticated interpreter may be a valid alternative to an unsophisticated dynamic compiler. The Virtual Virtual Machine project aims to build a *dynamically* reconfigurable virtual machine capable of running a variety of bytecode based languages [1]. vmgen [3] is a tool capable of generating interpreters for a variety of virtual machines. vmgen accepts instruction definitions in form of textual description in a special purpose programming language. We decided to express the instruction definitions directly as Java data structures and hence removed the necessity of a parser for the special purpose language. In effect, we have sped up the development process (since no parser is needed) as well as the development cycle (since the textual definition does not have to be processed to generate a representation the Java tools can work on).

The joeq virtual machine [17] is a compiler infrastructure that provides an intermediate representation modeling Java bytecode. The latest source distribution of joeq uses two variations of the visitor pattern to access the bytecode IR. The first variant is a nonhierarchical visitor that uses a switch statement on the opcode to dispatch to visit methods for the concrete instruction classes (e.g., GETFIELD). The visit methods take opcode-specific arguments that specify the parameterization of the instruction (for example, the selector of a field). These arguments are retrieved from the instruction stream by the code in the switch statement. The other style of visitor is a hierarchical visitor that dispatches over an instruction hierarchy with multiple inheritance, similar to the one used in Ovm. The major difference is that joeq uses accept methods that always sequentially call all applicable visit methods in abstract-to-concrete order (e.g., ExceptionThrower, StackConsumer, StackProducer, TypedInstruction, LoadClass, CPInstruction, FieldOrMethod, FieldInstruction, and GETFIELD for GETFIELD.accept()). For this visitor, information from the instruction stream is passed to the visitor using state in the instruction objects, preventing the use of singleton instruction objects for all parametrized opcodes. In Ovm, *all* instructions are flyweight objects and the required state is passed as an argument to the methods of the instructions that decode the bytecode stream. Notably, joeq's bytecode-related analyses use the switch-based nonhierarchical visitor almost exclusively in favor of the more object-oriented hierarchical visitor. Joeq also has a hierarchical visitor for its quad-based IR, which is similar to the hierarchical visitor for the bytecode IR.

8. FUTURE WORK

The instruction hierarchy is mostly ad hoc and tied to the Java bytecode instruction set. However, instead of static categorization in terms of the type system, one can envision dynamic inference of bytecode specification properties since most of the necessary information is present in the IR definition of instructions. For example, it is possible to infer that `IINC` both reads and writes local variables because of the way that `LocalAccess` code sources are used in its definition. Instructions could be dynamically annotated with appropriate properties in the inference phase. This would remove the burden of manual categorization from the programmer and ensure that categorization is consistent with the semantics given by the IR. This new approach assumes a static set of properties according to which the inference would proceed. As an extension, one could provide a way to specify new properties outside of the standard set supplied by the framework. Tools interested in such properties would have a way to specify them, and the inference engine would be able to discover these properties. For example, a tool performing stack height inference might be interested in identifying all the instructions that do not change the stack height. The tool would register a predicate, for example.

```
boolean hasInvariantStackHeight(Instruction instr) {
    return instr.stackIns.length == instr.stackOut.length;
}
```

The instruction property inference engine would annotate the appropriate instructions with the `InvariantStackHeight` property. The stack height inference tool would then ignore all instructions that have this property.

Currently, expression trees are given for each individual instruction. Optimizing compilers, however, typically operate on intermediate representations of entire functions. It is conceivable to convert the expression trees forming instruction definitions to standard three address code form and obtain an intermediate representation of any function by combining the expression trees resulting from the sequence of bytecodes constituting the definition of this function.

9. CONCLUSION

We have presented how certain design patterns can be used to build an extensive set of VM components around the specification of an intermediate representation. The resulting IR is customizable and the components using the IR can be easily adapted to changes in the configuration. Static analysis tools that use our implementation of a bytecode-based IR show competitive performance.

Acknowledgments. This work is supported by grants from DARPA (PCES), and NSF (CCR–9734265). The authors thank Tony Hosking, Doug Lea, David Holmes and Krista Bennett for their comments; Jacques Thomas for implementing ovmp.

10. REFERENCES

[1] B. Baillarguet and I. Piumarta. A highly-configurable, modular system for mobility, interoperability, specialization, and reuse. In *2nd ECOOP Workshop on Object-Orientation and Operating Systems*, June 1999.

[2] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing

compiler for Java. In *Proceedings ACM 1999 Java Grande Conference*, pages 129–141. ACM, June 1999.

[3] M. Anton Ertl and David Gregg. Building an interpreter with vmgen. In *Compiler Construction (CC'02)*, pages 5–8. Springer LNCS 2304, 2002.

[4] C. Flack, T. Hosking, and J. Vitek. Idioms in Ovm. Technical Report CSD-TR-03-017, Department of Computer Sciences, Purdue University, 2003.

[5] E. Gagnon and L. Hendren. Effective inline-threaded interpretation of java bytecode using preparation sequences. In *Compiler Construction, 12th International Conference*, Jan 2003.

[6] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[7] Christian Grothoff. Walkabout revisited: The runabout. In *ECOOP 2003 - Object-Oriented Programming*, Berlin, Heidelberg, New York, 2003. Springer-Verlag.

[8] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 241–253. ACM SIGPLAN, 2001.

[9] David Hovemeyer, William Pugh, and Jaime Spacco. Atomic instructions in java. In Magnusson [14], pages 133–154.

[10] Martin E. Nordberg III. Variations of the Visitor Pattern. 1996.

[11] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1994.

[12] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersen, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Object-Oriented Programming: 15th European Conference, Budapest, Hungary: proceedings*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Berlin, Heidelberg, New York, June 2001. Springer-Verlag.

[13] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[14] Boris Magnusson, editor. *ECOOP 2002 — Object-Oriented Programming: 16th European Conference, Málaga, Spain: proceedings*, volume 2374 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, New York, June 2002. Springer-Verlag.

[15] Jens Palsberg and Neal Glew. Type-safe method inlining. In Magnusson [14], pages 525–544.

[16] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.

[17] John Whaley. Joeq: A Virtual Machine and Compiler Infrastructure. In *ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME 2003)*. ACM SIGPLAN, 2003.