Harpoon Project Compiler Intermediate Representation

C. Scott Ananian

October 12, 1998
Revision: 1.16

1 Conceptual Overview

The Harpoon project compiler front-end translates Java bytecode files into a class-oriented intermediate representation which is intended to be easier to analyze and manipulate than bytecode assembly language. The intermediate representation is control-flow-graph structured, with all control flow explicit. It is also maximally factored and in static single assignment (SSA) form. Internally the intermediate representation is known as "QuadSSA," referring to its derivation from quadruple-style IRs and its SSA form.

1.1 Quadruples

Unlike expression-tree structured intermediate representations, where every operand can optionally be the root of an expression, quadruple representations are flat. The "typical" statement is of the form $a \leftarrow b \oplus c$; the name *quadruple* comes from the fact that there are four components (a, b, c, $\oplus$) [App98].[1] Obviously an IR expressive enough to represent the entire Java language needs more than a four component operation statement, but we have attempted to retain the atomic simplicity of the quadruple form.

[1] We use $\oplus$ to stand for an arbitrary binary operator.

1.2 Static Single-Assignment Form

Quoting from Appel in [App98]:

> Many dataflow analyses need to find the use-sites of each defined variable, or the definition-sites of each variable used in an expression. The *def-use chain* is a data structure that makes this efficient: for each statement in the flow graph, the compiler can keep a list of pointers to all the *use* sites of variables defined there, and a list of pointers to all *definition* sites of the variables used there. . . .
>
> An improvement on the idea of def-use chains is *static single-assignment form*, or *SSA form*, an intermediate representation in which each variable has only one definition in the program text. The one (static) definition-site may be in a loop that is executed many (dynamic) times, thus the name *static* single-assignment form.

An example of the use of SSA form is shown in figure 1. Simple variable renaming suffices to transform straight-line code into SSA form. Subscripts are used to emphasize the relationship of the renamed variables to the original variables. A benefit of SSA form which is obvious from the example is that unrelated uses of the same variable in the source

Conventional	Static Single Assignment
...	...
$\mathtt{i} = 0$	$\mathtt{i}_0 = 0$
$\mathtt{i} = \mathtt{i} + 1$	$\mathtt{i}_1 = \mathtt{i}_0 + 1$
$\mathtt{j} = \mathtt{func}(\mathtt{i})$	$\mathtt{j}_0 = \mathtt{func}(\mathtt{i}_1)$
$\mathtt{i} = 2$	$\mathtt{i}_2 = 2$
...	...

Figure 1: SSA transformation of straight-line code.

Conventional	Static Single Assignment
...	...
$\mathtt{i} = 0$	$\mathtt{i}_0 = 0$
if x **then**	**if** x **then**
$\mathtt{i} = 1$	$\mathtt{i}_1 = 1$
	$\mathtt{i}_2 = \phi(\mathtt{i}_0, \mathtt{i}_1)$
$\mathtt{j} = \mathtt{func}(\mathtt{i})$	$\mathtt{j}_0 = \mathtt{func}(\mathtt{i}_2)$
...	...

Figure 2: SSA transformation of branching code.

program ($\mathtt{i}_1$, $\mathtt{i}_2$) become different variables in the SSA form, eliminating false dependencies.

SSA form becomes more complex when we introduce branches and loops. Figure 2 shows the necessary transformation. You will notice the introduction of *phi functions* at locations where control flow merges. The ϕ-function "magically" chooses a value from among its arguments based on the control flow path used to reach it. Note that, although ϕ-functions are impossible to implement directly in an instruction set (due to their magical properties), they can be replaced by `move` instructions along each control flow edge leading to the ϕ-function. Doing so violates the static single assignment constraints, but leads to code executable by real processors.

Unless you are implementing code generator backends, it is unlikely you will need to so replace ϕ-functions or view them as anything but magical n-ary operators. However, it is important to observe and maintain the ordering relationship between control-flow edges and ϕ-function arguments during transformation and analysis.

Analysis, transformation, and optimization of the IR is simplified by its SSA form. In addition, the QuadSSA form is *maximally factored.* Constants are not allowed as quadruple operands (except for a special `const` operation); which creates a unique mapping from variable names to values in the computation. This simplifies value analysis.

Appel describes several other benefits of the SSA form in [App98]:

> —If a variable has N uses and M definitions (which occupy about $N + M$ instructions in a program), it takes space (and time) proportional to $N \cdot M$ to represent def-use chains—a quadratic blowup. For almost all realistic programs, the size of the SSA form is linear in the size of the original program.
>
> —Uses and defs of variables in SSA form relate in a useful way to the dominator structure of the control flow graph, which simplifies algorithms such as interference-graph construction.

1.3 Exception handling

Exception handling in the Java language complicates control-flow. Operations (such as division, possibly by zero) may implicitly throw exceptions that radically redirect the flow of control. To facilitate analysis, exception handling and its associated control-flow is made explicit in the intermediate representation. For example, null pointer and array bounds checks are inserted before object and array references, and division by zero is explicitly

checked before every division operation. These explicit and comprehensive checks are intended to be targeted by aggressive optimizations designed to removed those cases with are redundant, impossible, or otherwise unnecessary. The design goal of the IR is that *no* statement should throw an implicit exception under any circumstance. Pursuing that goal involves changing the semantics of method invocation slightly: so that the `call` statement not throw an implicit exception, the IR `call` has been defined to return *two* values. In addition to the conventional (and optional) method return value, an "exception return value" is defined to hold the exception thrown by a method, or `null` if the method completed without throwing an exception. Explicit tests of the exception return value can then be added after the `call` statement, and control-flow made explicit as with the other IR operations. The `throw` statement in the IR is thus stripped of its special meaning and becomes simply an alternate `return` statement for the exception return value.[2]

2 Implementation Details

The IR described in these notes is defined in the Java package `harpoon.IR.QuadSSA`. Source code and binaries are available at `http://www.magic.lcs.mit.edu/Harpoon`.

The QuadSSA statements are called `Quad`s and are subclasses of `harpoon.IR.QuadSSA.Quad`; they are graph-structured and doubly-linked to enable both forward and reverse traversal of the control-flow graph. Edges are represented by Objects to facilitate associating analysis data with these control-flow graph edges. The parent class `Quad` contains the graph-oriented methods of the objects. Its superinterface `harpoon.ClassFile.HCodeElement` defines standard methods to get object ID numbers and source file information which are valid for elements of any intermediate representation.

An enumeration of `Quad` types and their uses is provided in figure 3. It may be observed that the representation uses both header and footer nodes, in the `HEADER` and `FOOTER` classes. `HEADER` nodes contain a special link to the `FOOTER` to allow this node to be easily identified, and a special subclass of `HEADER`, `METHODHEADER`, provides information on the assignment of method arguments to compiler temporary variables at the start of method code.

With the exception of `CJMP`, `SWITCH`, `PHI`, `HEADER`, `METHODHEADER`, and `FOOTER`, all `Quad`s have exactly one predecessor and one successor in the control flow graph.

2.1 Quads

Here are more details on each Quad statement. First the header and footer nodes:

HEADER(f) Start node in the control flow graph with end node f. Performs no operation. Zero predecessors, one successor.

METHODHEADER(f, p) Start node in the control flow graph for a method with parameters p and end node f. A subclass of `HEADER`. The method arguments are loaded into $p_0 \ldots p_n$ before execution starts. Zero predecessors, one successor.

FOOTER( ) Final node in control-flow graph. Performs no operation. All `RETURN` and `THROW` statements must have the graph's `FOOTER` as their only successor. `FOOTER` nodes may have any positive number of predecessors. They have no successors.

[2] Try, catch, and finally blocks are, of course, taken into consideration when a bytecode `athrow` is translated to an IR `THROW`.

The following `Quad`s modify the control-flow:

PHI(t**,** l**)** Control flow merges at `PHI` nodes. A phi node represents a list of ϕ-functions of the form

$$t_i = \phi(l_{i0}, l_{i1} \ldots l_{ij})$$

where j is the *arity* of the ϕ-function; that is, the number of predecessors to the node. Any non-negative number of predecessors, one successor.

CJMP(t**)** Conditional jump based on the boolean t. If t is false (0), control flows to the first successor (`next[0]`). If t is true (1), control flows to the second successor (`next[1]`). One predecessor, two successors.

SWITCH(t**,** k**)** Indexed jump. Depending on the value of index variable t and a key list k, control is transferred to $target_n$ where $t = k_n$. If t does not match any key in k, control is transferred to the default $target_{n+1}$ where k_n is the last key in the key list. One predecessor, multiple successors.

RETURN(t**)** Return an optional value t from this method. One predecessor, one successor. The single successor should be a `FOOTER` node.

THROW(t**)** Throws an exception t as the result of this method. One predecessor, one successor. The single successor should be a `FOOTER` node.

The remaining `Quad`s have no effect on control flow, and have exactly one predecessor and one successor. No exceptions are thrown.

AGET(t_1**,** t_2**,** t_3**)** Fetches the element at index t_3 from array t_2 and stores the value in t_1.

ALENGTH(t_1**,** t_2**)** Puts the length of array t_2 into variable t_1.

ANEW(t_1**,** c**,** l**)** Creates a new uninitialized array with type c and dimensions $l_0 \ldots l_n$, storing a reference in t_1. The number of dimensions supplied in list l may be smaller than the number of dimensions of array class type c, in which case only the n dimensions specified in l will be created.

ASET(t_1**,** t_2**,** t_3**)** Sets the element at index t_2 of array t_1 to the value in t_3.

CALL(m**,** t**,** p**,** r**,** e**)** Calls method m of optional class reference t with parameter list $p_0 \ldots p_n$, putting the return value in r if no exception is thrown, or setting e to the thrown exception. Either r or e will be `null` on completion of the `CALL`. Exception e is not automatically thrown from the method containing the `CALL`: e must be explicitly tested and its exception rethrown if that behavior is desired. t is not needed for static methods.

COMPONENTOF(t_1**,** t_2**,** t_3**)** Puts the boolean value `true` (1) in t_1 if object t_3 is an instance of the component type of array t_2, or `false` (0) otherwise.

CONST(t**,** c**,** y**)** Assigns numeric or string constant c of type y to variable t.

GET(t_1**,** f**,** t_2**)** Puts the value of field f of optional object t_2 in variable t_1. t_2 is not necessary for static fields.

INSTANCEOF(t_1**,** t_2**,** c**)** Puts the boolean value `true` (1) in t_1 if object t_2 is an instance of class c, or `false` (0) otherwise.

MOVE(t_1**,** t_2**)** Copies the value in t_2 into t_1.

NEW(t**,** c**)** Create a new uninitialized instance of class c, storing a reference in t. A class con-

structor must be invoked using `CALL` in order to initialize the instance.

NOP( ) Performs no operation.

OPER(*o*, *t*, *l*) Performs operation o on the variables in list l, storing the result in t. The operation is represented as a string; figure 3 lists all valid operation strings. The operations performed by the strings are identical to the operations performed by the Java bytecode operation of the same name, except that no exceptions are ever thrown. See [LY96] for details.

SET(*f*, t_1, t_2) Sets field f of optional object t_1 to the value of t_2. t_1 is not necessary for static fields.

The `harpoon.IR.QuadSSA.Code` class provides a means to access the QuadSSA form of a given method; see the definition of superclass `harpoon.ClassFile.HCode` and the example code in `harpoon.Main.Main` for details.

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, September 1996. Online at `http://www.javasoft.com/docs/books/vmspec`.

```
package harpoon.IR.QuadSSA;
```

Quadruple statements

```
abstract class Quad implements HCodeElement
AGET(HCodeElement source, Temp dst, Temp objectref, Temp index)
ALENGTH(HCodeElement source, Temp dst, Temp objectref)
ANEW(HCodeElement source, Temp dst, HClass hclass, Temp dims[])
ASET(HCodeElement source, Temp objectref, Temp index, Temp src)
CALL(HCodeElement source, HMethod method, Temp objectref, Temp params[],
    Temp retval, Temp retex) // objectref, retval may be null
CJMP(HCodeElement source, Temp test)
COMPONENTOF(HCodeElement source, Temp dst, Temp arrayref, Temp objectref)
CONST(HCodeElement source, Temp dst, Object value, HClass type)
FOOTER(HCodeElement source)
GET(HCodeElement source, Temp dst, HField field,
    Temp objectref) // objectref may be null
HEADER(HCodeElement source, FOOTER footer)
INSTANCEOF(HCodeElement source, Temp dst, Temp src, HClass hclass)
METHODHEADER(HCodeElement source, FOOTER footer, Temp params[])
MOVE(HCodeElement source, Temp dst, Temp src)
NEW(HCodeElement source, Temp dst, HClass hclass)
NOP(HCodeElement source)
OPER(HCodeElement source, String opcode, Temp dst, Temp operands[])
PHI(HCodeElement source, Temp dst[], int arity)
PHI(HCodeElement source, Temp dst[], Temp src[][], int arity)
RETURN(HCodeElement source, Temp retval) // retval may be null
SET(HCodeElement source, HField field, Temp objectref,
    Temp src) // objectref may be null
SWITCH(HCodeElement source, Temp index, int keys[])
THROW(HCodeElement source, Temp throwable)
```

String constants for `opcode` *field of* **OPER**

```
{ "acmpeq", "d2f", "d2i", "d2l", "dadd", "dcmpg", "dcmpl", "ddiv", "dmul",
    "dneg", "drem", "dsub", "f2d", "f2i", "f2l", "fadd", "fcmpg", "fcmpl",
    "fdiv", "fmul", "fneg", "frem", "fsub", "i2b", "i2c", "i2d", "i2f", "i2l",
    "i2s", "iadd", "iand", "icmpeq", "icmpge", "icmpgt", "idiv", "imul",
    "ineg", "ior", "irem", "ishl", "ishr", "isub", "iushr", "ixor", "l2d",
    "l2f", "l2i", "ladd", "land", "lcmpeq", "lcmpge", "lcmpgt", "ldiv",
    "lmul", "lneg", "lor", "lrem", "lshl", "lshr", "lsub", "lushr", "lxor"
};
```

Figure 3: Classes comprising the QuadSSA intermediate representation. Only the constructors are shown; the object field variables correspond exactly to the names of the constructor arguments.

A Quick reference

Class Name	Description
HEADER	Start node for control-flow graph. Performs no operation.
METHODHEADER	Subclass of HEADER with additional method-argument information.
FOOTER	End node for control-flow graph. Performs no operation.
AGET	Fetch from an indexed array element.
ALENGTH	Access the length of an array.
ANEW	Create a new array object (uninitialized).
ASET	Assign a value to an indexed array element.
CALL	Invoke an object method.
CJMP	Conditional jump based on a boolean value.
COMPONENTOF	Determine whether an object reference is an instance of the component type of an array reference; result is a boolean value.
CONST	Assign numeric or string constants to compiler temporary variables.
GET	Fetch the value of an object field.
INSTANCEOF	Determine whether an object reference is an instance of a given class; result is a boolean value.
MOVE	Assign one compiler temporary to another.
NEW	Create a new uninitialized class object.
NOP	Do nothing.
OPER	Perform a n-ary operation on set of compiler temporaries.
PHI	ϕ-function representation.
RETURN	Return a value for this method invocation.
SET	Assign a value to an object field.
SWITCH	Jump to one of multiple targets, depending on a key value.
THROW	Return an exception for this method invocation.