Jumbo: Run-time Code Generation for Java and Its Applications

Sam Kamin*, Lars Clausen, Ava Jarvis
Computer Science Department
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{s-kamin,lrclause,ajar}@uiuc.edu

Abstract

Run-time code generation is a well-known technique for improving the efficiency of programs by exploiting dynamic information. Unfortunately, the difficulty of constructing run-time code-generators has hampered their widespread use. We describe Jumbo, a tool for easily creating run-time code generators for Java. Jumbo is a compiler for a two-level version of Java, where programs can contain quoted code fragments. The Jumbo API allows the code fragments to be combined at run-time and then executed. We illustrate Jumbo with several examples that show significant speed-ups over similar code written in plain Java, and argue further that Jumbo is a generalized software component system.

Keywords: run-time code generation, Java

1 Introduction

Jumbo is a dynamic compiler for Java capable of pulling together fragments of Java code — down to the level of single expressions or statements — into a single program and compiling the combined program at run-time. Jumbo does not simply build the program as text and invoke the Java compiler from the running program. Instead, the compiler is structured in such a way that fragments can be separately compiled to an intermediate form from which the full program can be assembled without reinvoking the compiler. In other words, Jumbo builds run-time code generators.

This structure allows for significant flexibility in the program generators Jumbo can build. Furthermore, since the program generators are binaries (that is, Java class files) and produce binaries, with no separate invocation of the compiler, Jumbo passes the "deployability" test for components. From this point of view, Jumbo can be seen as a component-constructing tool of great generality — greater than traditional component systems that operate at the level of whole methods or classes. Jumbo components can be thought of as "run-time macros."

In this paper, we introduce Jumbo and give several examples of its use. To a great extent, using Jumbo "feels" like creating strings that look like programs and then compiling them. Conceptually, the model is very simple. In some examples, complex interactions between component and client make for an inherent complexity that Jumbo cannot eliminate; however, the programmer's focus is always on the question "What should the generated program look like?", and this makes Jumbo straightforward to use.

The next section gives an overview of the Jumbo system. In Section 3, we present two examples of the use of Jumbo. References to related work and conclusions are presented in the final sections.

2 The Jumbo System

Jumbo is a "compiler API," consisting of about a dozen classes, the most important of which is `Code`. `Code` contains a number of static methods similar to these examples:

```
static Code binop (int, Code, Code)
static Code ifThenElse (Code, Code, Code)
static Code returnVal (Code)
```

These methods correspond to abstract syntax operations, acting upon and producing values of type `Code`. When using Jumbo, it is not misleading to think of `Code` as containing abstract syntax trees, or even strings; however, `Code` is neither of these things, but is instead a partially compiled value. Programmers will rarely use these AST operations directly, as a special quotation syntax is provided to produce calls to these operations. We will show examples of the quotation syntax later in this section.

In addition, the Code class provides the following two instance methods:

*Partial support for the first and second authors was received from NSF under grant CCR-9619644.

Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)
0-7695-1913-X/03 $17.00 © 2003 IEEE

```
void generate ();
Object create (String classname);
```

The `generate()` method converts a `Code` value into JVM code and writes the corresponding class files; the `Code` value must correspond to a class or list of classes. `create()` calls `generate()` and then loads the named class and returns a new object of that class. (In the current version of Jumbo, due to our reliance on the `Class` method `newInstance()`, no arguments can be passed to the constructor.)

The basic idea behind Jumbo is *compositional compilation* [6]. The crucial point is that the abstract syntax operations of the Jumbo API *are* the compiler. Unlike an ordinary compiler in which the syntax tree is a passive data structure upon which the compiler operates, the Jumbo operators are genuine functions that perform compilation. This is called a compositional compiler because the compilation of each language construct is a function only of the compilation of its sub-constructs — a very different structure from conventional compilers. The advantage is that any particular piece of syntax can be easily abstracted from and filled in at a later time.

The idea of compositional compilation is widely applicable to different languages and target architectures. In [6], we used a variant of Java and targeted SPARC machine language. Some languages and target architectures are much more difficult to compile compositionally than others, and tradeoffs in code quality must be made. The structure of Java and its implementation by an abstract machine are particularly well suited to our approach, and Jumbo produces virtually the same code as the `javac` compiler. Only inner classes, being especially difficult to handle, are not yet implemented in Jumbo.

2.1 Using code generators

In discussing Jumbo programs, we often refer to the supplier of the code generator as the *server*, and the user of the supplied code generator as the *client*. Following this metaphor, we also sometimes refer to Jumbo programs as *components*.

The following code is a typical Jumbo client which uses a vector dot-product generator supplied by a server. `Dot` is an interface containing the method `double dot(double[])`.

```
double[] vec1 = getInputVector();
Code dotprodcode = codegenDot(vec1);
Dot dotprod =
   (Dot)dotprodcode.create("DotProd");
while (true) {
  double[] vec2 = getInputVector();
  double p = dotprod.dot(vec2);
  outputResult(p);
}
```

The server supplies `codegenDot`, a Jumbo method whose function is to produce a class that implements the `Dot` interface. We will show the definition of `codegenDot` shortly. As with any other library code, the client knows only the type, `Code codegenDot (double[])`, and purpose of the method. `create` can be applied to the value returned by `codegenDot` to compile the class (named `DotProd`) and return an instance of that class. Since that class does not exist at the time the client code is compiled, we must use the `Dot` interface.

In summary, the client uses the method `codegenDot` to produce a method that is specialized to multiply other vectors by the fixed vector `vec1`. Fixing one of the vectors in a dot product calculation offers the potential to speed up the calculation by using dynamic information. That information includes the length of the vector, which allows the dot product loop to be unrolled, and its values, which may be exploited to gain efficiency — for example, by omitting multiplications by zero or one.

The pure Java alternative to the above would use a non-generating dot product routine, say `dotprod`:

```
double[] vec1 = getInputVector();
while (true) {
  double[] vec2 = getInputVector();
  double p = dotprod(vec1, vec2);
  outputResult(p);
}
```

where `dotprod` is the straightforward library method (omitted for brevity).

The advantage of the dynamically compiled version is that, in some cases, it may be much faster; the disadvantage is that, in other cases, it may be much slower due the cost of run-time compilation. Like any other programming tool, Jumbo must be used judiciously. When appropriately applied, Jumbo gives speed-ups that are hard to obtain by conventional means.

2.2 Speed-ups

A use of Jumbo involves three time-consuming stages:

1. Loading the Jumbo API.
2. Generating and loading the class file.
3. Executing the generated code.

In the `DotProduct` example, step 2 is performed when variable `dotprodcode` is assigned and then sent the `create` message. If the class file were created during one session (by sending the `generate` message) and then used

Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)
0-7695-1913-X/03 $17.00 © 2003 IEEE

in a separate session, we would break step 2 into two parts, but for this example that is not necessary.

Each of these timings is potentially of interest to the Jumbo user. A single use of Jumbo, as in the current example, entails all three steps. Once the Jumbo API is loaded (or, speaking more speculatively, if it were built into the virtual machine) step 2 represents the incremental cost of generating new code dynamically. Step 3 gives the "bottom line", the speed of the generated code — which should be, and is, substantially faster than plain Java code. Whether or not using Jumbo saves time overall depends on how it is used.

Note that there are two variables in the dot product computation that affect the overall computational cost: the size of the vectors and the number of dot product computations that are performed. The cost of step 1 is independent of these numbers, and the cost of step 2 depends only on the vector `vec1`. For the timings presented here, we used randomly-generated 100-element vectors of two kinds: no bias, and bias towards generating zeroes (sparse vectors). The results are given in Table 1. The dot product computation was repeated a varying number of times (n), from 10 to 1,000,000. For the Jumbo version, the code generation process (step 2) occurred just once in each run.

All timings were produced on an unloaded 1.47GHz Athlon PC with 1GB of memory, running Linux kernel 2.4.18. We used Sun's Java SDK, version 1.3.1 (Standard Edition, Blackdown build). Times are in milliseconds, obtained with a microsecond timer routine.

The results are easily summarized:

- Step 1, the loading of the Jumbo API, took about 0.2 seconds. For small numbers of iterations, this time dominates. Step 1 takes about 15 times longer than step 2 (in the unbiased case).
- Step 2, code generation, took about 14 milliseconds for the unbiased vector and 6 milliseconds for the biased vector. This is, in turn, about 300–700 times slower than a single dot product calculation. (See Table 1 for more accurate numbers.) Note that the cost per dot product declines significantly as the number of iterations rises because of the effect of run-time optimizations in the Java run-time implementation. Thus, for 10 iterations, the cost is about 19 μsec per iteration; for 1,000,000 iterations, the cost goes down to about .3 μsec per iteration. Since the cost of code generation remains constant, the ratio of code generation to execution time rises sharply.
- Step 3, execution time of the generated code for the unbiased case took from 19 μsec per dot product computation down to about .3 μsec, as discussed above. For the biased case, these numbers were cut approximately in half, as would be expected.

Random data	**Jumbo**	**Java**
Load API	226.0	N/A
Code gen.	14.3	N/A
Run-time (n)		
10	0.19	0.16
100	0.7	1.5
1000	5.6	19.6
10000	15.5	48.9
100000	43.2	127.4
1000000	290.7	929.1

50% 0's	**Jumbo**	**Java**
Load API	228.0	N/A
Code gen.	5.57	N/A
Run-time (n)		
10	0.1	0.16
100	0.4	1.5
1000	3.0	15.1
10000	8.3	39.8
100000	24.3	120.2
1000000	151.0	925.8

Table 1. Effect of dynamically compiling dot product routine. (Times in milliseconds.)

- By comparison, the Java version does not pay the costs of steps 1 and 2, but the dot product computation took from 16 μsec to .9 μsec in both the unbiased and unbiased cases, per iteration. (The biased case is slightly faster because floating-point multiplication by zero is faster on this platform than multiplication by non-zero values.) Over time, the generated code in the unbiased case ran about three times faster per iteration than the Java code, and in the biased case, about five times faster.

The "cross-over point" depends upon which parts of the computation are included. For a complete run, including all three stages of Jumbo computation, the cross-over is at about 400,000 iterations in the unbiased case, 300,000 in the biased case. If the API is assumed to be preloaded — which is to say, for all uses of Jumbo except the first — then the cross-over points are at about 1000 and 500 iterations for the two cases.

The results of this run show the potential for significant speed-ups when the cost of loading and dynamic code generation is amortized over a sufficient number of uses. For the unbiased vectors, where the only benefits of Jumbo come from loop unrolling, the Jumbo version is eventually over three times as fast as Java; for the biased vector, the improvement factor is about five.

Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)
0-7695-1913-X/03 $17.00 © 2003 IEEE

We are currently investigating methods of decreasing run-time compilation time, and we expect great improvements by applying partial evaluation techniques. Furthermore, Jumbo can be used statically as well as dynamically, and in some cases (as in the example in Section 3.1) this is quite appropriate. In that case, the run-time seen by the user would be that shown in the last column.

2.3 Writing code generators

Someone has to write the code generators, and Jumbo is easy to use in this respect. Component writers must use the Jumbo compiler (unlike in the previous section, where the client only needed the Jumbo API and could have used any Java compiler). The Jumbo compiler processes a "two-level" version of Java to produce code generators, using a quote/anti-quote syntax similar to that of MetaML [10].

A fragment of Java code within brackets `$<` and `>$` represents a value of type `Code`:

```
Code C = $< class CodeExample
             implements InterfaceType {
           ... } >$;
InterfaceType obj =
  (InterfaceType)C.create("CodeExample");
/* ... use obj ... */
```

The quotation syntax works much like ordinary string quotation, except that the result is a value of type `Code` rather than `String`. Java type-checking requires that we use an interface `InterfaceType`, since the class `CodeExample` does not exist at the time this code fragment is compiled. (The argument to `create` is redundant in this case, but required because the `Code` value may, in general, contain more than once class definition.)

The Jumbo compiler is used to compile the code within quotes. The only restriction on such quoted code is that inner classes, not yet implemented in Jumbo, cannot be used. In our examples, we sometimes use inner classes, but never inside quotes.

Within a quoted Java fragment, values of type `Code` can be "spliced," using the anti-quotation syntax

‘*syntax-category* (*Java code fragment*)

The entire anti-quoted section is replaced, for parsing purposes, by a generic fragment of syntactic type *syntax-category*; the only reason we require the syntax category to be present is to allow for parsing of the surrounding code.

For example, we can write

```
Code addxy = $< x+y >$;
Code assignz = $< z = x*5; >$;
Code example =
  $< class CodeExample
       implements InterfaceType {
          ... ‘Expr(addxy) ... ‘Stmt(assignz) ...
       } >$;
  InterfaceType obj =
    (InterfaceType)C.create("CodeExample");
  /* ... use obj ... */
```

The anti-quoted expressions `‘Expr(addxy)` and `‘Stmt(assignz)` are "holes" filled by the code in `addxy` and `assignz`. As mentioned earlier, the casual user can think of this as splicing strings into the string `example`; however, these values are not strings but rather of type `Code`.[1]

The anti-quoted parts can be any Java expression of type `Code`. If the method `f` has signature `Code f(int)`, we can write

```
$< ... ‘Expr( f(3) ) ... >$
```

At run-time, `f(3)` would be evaluated and the resulting `Code` value spliced into the surrounding `Code` value.

Table 2 shows the syntactic categories available for anti-quotations. For each case, the table shows the type of the expression that must be given as an argument to the syntax category. The examples above use syntax categories that take arguments of type `Code`, `Expr` and `Stmt`. As the table shows, there are some syntax categories that require different types of arguments:

Category	Expression value expected (Type)
`Expr`	Expressions (Code)
`Stmt`	Statements (Code)
`Name`	Identifiers (String) (Default category)
`Type`	Types (Type)
`Case`	List of case branches (MonoList of Code values)
`Method`	Method declaration (Code)
`Field`	Field declaration (Code)
`Body`	List of class members (MonoList of Code values)
`Char`	Character constant (char)
`Int`	Integer constant (int)
`Float`	Float constant (float)
`Long`	Long constant (long)
`Double`	Double constant (double)
`Bool`	Boolean constant (boolean)
`String`	String constant (String)

Table 2. Syntactic categories for anti-quotation

[1] We insist on this perhaps seemingly pedantic distinction. In our view, it is a virtue of Jumbo that the programmer can *think* of these fragments as strings, because that is a simple model to understand; it is also a virtue of Jumbo that they are *not* strings, because this allows for a more efficient, non-source based implementation.

Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)
0-7695-1913-X/03 $17.00 © 2003 IEEE

- `Name` is a category used when the only syntactically valid item is an identifier. `Name` would be used, for example, to abstract the name of a class. This is the default category; when we write `'ident` with no syntax category and no parentheses, `ident` must be a variable of type `String`.
- `Type` is used wherever a type is required. In order for `$< 'Type(t) x, y; >$` to be a valid declaration, `t` must be a Java expression of type `Type`. A value of type `Type` can be created by using `Type.parseType(String jtype)`, where `jtype` is a Java-style type. `Type` also defines constant type values `int_type`, `float_type` etc. for the primitive types.
- The `Case` and `Body` categories require `MonoLists` of `Code` values. `MonoList` is a collection class interface included in the Jumbo API and differs from Java's `List` class primarily in that the `add` operation has type `MonoList add(Object)` instead of `boolean add(Object)`; the functional style `add` operator turns out to be more convenient for our purposes. A quote of switch cases (`$<case 0: ... case 1: ...>$`) returns a `MonoList` of `Code` values; all other quoted fragments return `Code`.
- The last categories (`Char`, ..., `String`) must have arguments of the corresponding types. (That is, expressions of those exact types, not `Code`.) These allow Java values to be "reified," i.e. turned into syntax that can be included in generated programs.

There is only one `Code` type, but it covers a variety of syntactic types. Intuitively, `$<x+y>$` should denote "a `Code` value of type `Expr`", and `$<x=y;>$` should denote "a `Code` value of type `Stmt`." When we write, for instance, `$<...'Expr(e)...>$`, we should insist that `e` evaluates to a `Code` value of type `Expr`. However, we make no such distinctions and have only a single type, `Code`. In discourse, we often refer to a `Code` value "corresponding to" an expression or statement, etc. We do no type-checking of generated code when we compile the program generator (as is done in MetaML); such type-checking is done when `generate` is called. `generate` and `create` may only be applied to a `Code` value corresponding to a class definition or list of same.

We have now completed our presentation of Jumbo and are able to present the definition of `codegenDot`. We gain speed-ups by unrolling the loop and using the fixed values of the static vector as constants, removing the iteration and array lookup overhead. Additionally, we take advantage of the frequent presence of 0's and 1's in vectors by applying simple arithmetic equivalences of multiplication.

The code used for Table 1 follows:

```
public static Dot codegenDot(double[] V1) {
  Code c =
    $<public class DotProd
      implements Dot {
        public double dot(double[] V2) {
          return 'Expr(makeSumCode(V1, $<V2>$));
        }
      }>$;
  return (Dot)c.create("DotProd");
}

public static Code makeSumCode(double[] V1,
                               String V2) {
  Code sumcode = $<0.0>$;
  for (int i = 0; i < V1.length; i++) {
    if (V1[i] == 1.0)
      sumcode = $<'Expr(sumcode)+'V2['Int(i)]>$;
    else if (V1[i] != 0.0)
      sumcode = $<'Expr(sumcode)
                  +'Double(V1[i])*'V2['Int(i)]>$;
  }
  return sumcode;
}
```

If `v1` were the array `{5.2, 0.0, 2.4, 1.0}`, the generated class would contain a method equivalent to

```
return 0.0 + 5.2*vec2[0]
       + 2.4*vec2[2] + vec2[3];
```

Other static characteristics of the vector `v1`, such as repetitions or symmetries, could be exploited as well.

We have explained essentially all of the features of Jumbo: a "compilation API" with a parser that adds a quotation/anti-quotation feature. These conceptually simple facilities are remarkably powerful, as illustrated in the remainder of this paper. Further examples can be found in [5, 6], which describe a predecessor of Jumbo.

3 Examples

In this section, we give two examples that show the performance benefit obtained using RTCG. The first example is a *first-order* method, which doesn't require the client to use the Jumbo compiler, only the Jumbo API. The second example involves *higher-order* methods, where both clients and component writers create code fragments using the Jumbo compiler.

3.1 Generic data types

In Java, most collection classes are generic in a trivial way: they contain only `Objects`. One disadvantage is that primitive values must be boxed in wrapper classes like like

Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)
0-7695-1913-X/03 $17.00 © 2003 IEEE

`Integer` and `Character` before being used in a collection. Boxing has a significant run-time cost due to extra object creation. Yet these generic collection classes can be easily transformed into non-generic classes: just replace the type `Object` by the desired type of element, and recompile — in essence, the effect of using templates in C++.

We can get the same benefit in Jumbo by supplying the type name as a parameter to a program generator. For example, the `Vector` class in the Java library is a heavily used collection class, with operations like `add` and `iterator`. We created a non-generic collection class generator using Jumbo, which generates classes similar to the standard `Vector` class. The generator provides three operations:

```
static Type vector (String type);
static Type iterator (String type);
static Code newVector (String type);
```

The `vector` and `iterator` methods are used to create the names of the vector type corresponding to a particular non-generic instance, and the type of its iterator. The argument is the name of the component type. `newVector` creates an expression whose value is an object of the desired collection type. The non-generic class is created implicitly whenever one of these operations is called, but is only created once.

The implementation consists largely of a Java class placed within quotations and abstracted on the names of the element type and the collection type:

```
Code C =
  $<public class `vectorname {
     `Type(elttype)[] elements;
     int numelements;

     public `vectorname() {
       elements = new `Type(elttype)[10];
     }

     public void add(`Type(elttype) o) {
       ...
     }

     ...
  >$;
```

The only additional code, which is not shown, is the code to insure that the desired collection class is generated exactly once.

The timings shown in Table 3 were obtained by creating vectors of various lengths (`vlen`) and then summing them:

```
`Type(vector("int")) v =
     `Expr(newVector("int"));
for (int i = 0; i < vlen; i++) {
    v.add(i);
}
int sum1 = 0;
`Type(iterator("int")) i;
for (i = v.iterator();
     i.hasNext(); ) {
  sum1 += i.next();
}
```

vlen	vector(”int”)	vector(”Object”)
100	1	1
1000	2	2
10000	7	22
100000	23	132
1000000	186	976

Table 3. Effect of dynamically compiled non-generic collection class. (Times in milliseconds.)

For the Java column, we mimicked Java by creating a generic class `vector("Object")` and using it as one would an ordinary generic collection in Java.

Table 3 shows only the execution times, since we assume the various vector classes are created and compiled first, using `generate`. Afterwards, they are ordinary class files, and can be used without further compilation. It will come as no surprise to Java programmers that a vector of `int`s is much more efficient than a vector of `Integer`s.

3.2 Loop unrolling

The full power of Jumbo can be employed only when the client has the Jumbo compiler and can write his own code-producing methods. This provides for communication between the component and client. Loop unrolling is one such situation: a loop unrolling component allows the client to supply the body of a loop along with the bounds of the loop, and the component unrolls the loop as needed.

Intuitively, loop unrolling is *second order*; the client supplies the body of the loop and the unrolling method repeats it as necessary. However, supplying the body as a `Code` value does not allow the body to access the loop variable. The solution is for the body of the loop, supplied by the client, to itself be a `Code`-producing function (more precisely, a function object). A component whose argument is a function is called a *higher order* component. Here, the client passes an instance of the `LoopIteration` interface, in which the `iteration` method takes a code fragment for the iteration value and returns a code fragment for the body, incorporating the iteration value:

```
interface LoopIteration {
```


Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)
0-7695-1913-X/03 $17.00 © 2003 IEEE

```
  Code iteration (Code i);
}
```

A loop unroller takes as arguments not only the number of iterations and the loop body, but also the initial value of the iteration variable, a loop increment, and the iteration variable itself. Of these, only the iteration count and the increment are constants; the initial value and the loop variable are expressions, and the loop body is a `LoopIteration` value.

The simplest unroller is a complete unroller:

```
public static Code unroll_all (
    Code i,    // iteration variable
    Code init, // initial value
    int incr,  // increment (fixed value)
    int iterations, //iterations (fixed)
    LoopIteration F // loop body
  ) {
   Code C = $< ; >$ ;
   for (int x=0; x<iterations; x++)
     C = $< `Stmt(C)
             `Stmt(F.iteration(
                $<`Expr(init)+`Int(x*incr)>$))
         >$;
   C = $< `Stmt(C)
           `Expr(i) = `Expr(init)
                    +`Int(iterations*incr);
       >$ ;
   return C;
 }
```

The call

```
unroll_all(
  $<i>$, $<0>$, 1, 3,
  new LoopIteration () {
    public Code iteration (Code x) {
      return $<System.out.print(`Expr(x));>$;
    }
  });
```

produces code equivalent to

```
System.out.print(0);
System.out.print(1);
System.out.print(2);
i=3;
```

Part of the unroller's contract is to increment the index variable by the appropriate amount.

This unrolling component can be applied to our initial example, `dotgen`. However, full unrolling of the loop in `dot` can be highly disadvantageous. With an array of size 1000, the fully unrolled version runs approximately eight times slower than the original version. The reason for this difference in speed comes from one of two causes, or a combination thereof. The HotSpot run-time system for Java performs optimizations at run time selectively, with longer methods optimized much less aggressively then shorter ones. Secondly, longer methods can result in costly misses in the instruction cache. In any case, it has been frequently observed that long methods, though they execute fewer instructions, can be slower than short ones.

This suggests a compromise: partially unrolling the loops. In fact, this is the approach often used by traditional optimizing compilers. The following unroller has an additional argument, the "block size," i.e. the number of iterations that should be unfolded in each loop iteration.

```
public static Code unroll_part (
    Code i, Code init, int incr,
    int iterations, LoopIteration F,
    int BlockSize   // max loop size
  ) {
  int loops = iterations/BlockSize,
      leftover = iterations%BlockSize;
  if (loops < 2)   // 0 or 1 loops - unroll
    return unroll_all(i, init, incr,
                      iterations, F);
  else
    return
      $< for (`Expr(i)=`Expr(init);
              `Expr(i) < `Expr(init)
                 + `Int(loops*BlockSize*incr); )
         {
           `Stmt(unroll_all(i, i, incr,
                            BlockSize, F))
         }
         `Stmt(unroll_all(i, i, incr,
                          leftover, F))
      >$;
}
```

When run on a vector of 1000 elements, the most efficient dot product computation (counting only run time, not code generation time, and repeating the dot product computation 100,000 times) was achieved using a blocking factor of 24. The execution speed was about 14% faster than no unrolling, and about ten times faster than complete unrolling.

4 Related work

Several researchers have noted that program generators might be useful in ways that conventional components are not. An example is Batory's notion of components as *layers* [1]. In this view — greatly simplified — a component is a collection of classes whose *superclasses* are undetermined. From our point of view, then, a "layer" is a function from a list of superclass names to a list of classes. This is easy to implement in Jumbo (we simplify here by assuming the component contains just one class):

Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)
0-7695-1913-X/03 $17.00 © 2003 IEEE

```
Code myComponent (String superclass) {
  return $< class myClass
            extends `superclass {
              ...
            } >$;
}
```

Instantiating a layer means applying it to a particular class name:

```
myComponent("MySuperClass").generate();
```

Another example is [7], in which simple additions to class files are made automatically at load time (mainly to solve version integration problems); again, this facility could be provided by using Jumbo. Yet another is the work of Franz and Kistler [3], in which a compact representation of abstract syntax trees is used as the medium of communication of programs, instead of conventional binaries; compilation to machine code is performed at load time.

Undoubtedly, these systems require support that goes beyond what Jumbo can offer and therefore need to be built into Java compilers and run-time systems in some ways. However, to a first order of approximation, these are just RTCG systems, and Jumbo provides a simple way to experiment with such ideas.

On a technical level, Jumbo's closest relatives are various two-level language systems, such as MetaML [10] and 'C [2]. As compared with these, Jumbo is distinguished by the simplicity of the programming model it provides. To a great extent, Jumbo programmers can think of their programs simply as strings. The systems cited above impose various restrictions that limit the programmer's ability to abstract on some parts of their programs (most notably, on type names). In Jumbo, context-sensitive checks are not done until the entire program is put together at code-generation time.

MetaML [10] is a partial evaluation system with explicit staging. Though superficially similar, MetaML and Jumbo are quite different at the semantic level. MetaML is designed to be transparent to the programmer, in the sense that the only result of omitting the staging information will be slower execution. This transparency has an obvious advantage: the programmer is really programming in only one language — the two-level annotations are just "pragmas" — so that writing correct programs is no more difficult than in the base language. Its disadvantage is that, as mentioned above, the available abstractions are limited to those already available in the base language; programs cannot be parameterized on type names or exception names, for instance. There is also a serious question of *control* of the specialization process which arises in all partial evaluation-based systems: the difficulty in arranging to have exactly the right code generated at run time. For example, in the sorting example in [5], a certain code fragment is to be inlined only if a particular variable has a value within a given range; the MetaML programmer cannot make this decision explicit.

JSpec [8] is a partial evaluation system for Java programs; the programmer indicates where and when specialization should occur in a "specialization class" separate from the Java program itself. The comments just made with respect to MetaML apply here as well.

'C [2] and Cyclone [4] are systems that are similar to ours in that they include a two-level notation to specify run-time code generation. As mentioned above, these systems impose restrictions to insure that type-checking can be performed (mostly) by the compiler, that control flows into and out of generated code are known to the compiler, and so on. In Jumbo, such checks are performed when run-time code generation is performed; of course, earlier checking is preferable, but the Jumbo approach gives the programmer great freedom to modularize her program, requiring only that the end result be correct. (Manifest type errors — those that are visible in the quoted code — can, in principle, be caught early by the Jumbo compiler; this is one goal of our current development.)

We should mention that run-time code generation can also be performed by more primitive methods. Sestoft [9] describes the use of various APIs for creating JVM class files at run time, in which the user provides the byte codes for each method. Reflection can also be used to perform a certain amount of customization of programs [11]. The advantage of Jumbo — or any of the other systems mentioned in this section — over such approaches is that little more is required of the programmer than the knowledge of Java which he already possesses.

Jumbo uses the same language as metalanguage and object language, but this is mostly a convenience. In our earlier work [6], the metalanguage and object language were distinct — the former a functional language and the latter an imperative language. The only additional feature made possible by having the same meta- and object language is the possibility of unbounded levels of quotation. For parsing reasons, unbounded levels of quotation are not allowed in Jumbo at the moment.

5 Conclusions

The Jumbo system correctly compiles most of Java. All the examples shown in this paper compile and run as advertised. More information about Jumbo — including, in the near future, the system itself — can be obtained from our web site, `http://fuji.cs.uiuc.edu/Jumbo`.

Besides working on specific applications, we have two immediate goals with respect to the Jumbo system. The first is to complete Jumbo by implementing inner classes. At present, Jumbo compiles the rest of Java, but omits certain static checks (though they are discovered by the verifier); in

Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)
0-7695-1913-X/03 $17.00 © 2003 IEEE

essence, Jumbo produces the same code as `javac`. It will be especially useful to have a complete implementation of Java, because it will then be possible to quote — and thus abstract parts of — *any* Java code.

The other goal is to optimize the code generation process. The structure of Jumbo should permit this to be done very effectively: even if there is a hole in a piece of quoted Java code, most of the work involved in compiling that code can normally be done statically. In principle, the cost of code generation could be dramatically reduced in most cases. The resulting code-generating components could be used more routinely, since the need to amortizem the compilation cost would be that much less.

The general idea of compositional compilation has broad applicability. In [6], we described an earlier implementation targetting SPARC machine code instead of JVM code. Targeting machine code might be useful for producing code for small computers that cannot support a JVM. The concept can also be applied to other languages, although for many languages the implementation would be more complex than Jumbo.

Security and correctness are major issues in this context. The behaviors of components can be hard to describe, since their "holes" can be filled by arbitrary Java fragments. The same problem appears in functional languages in which functions can have side effects, and in object-oriented languages in which a method can be overridden by another with no enforceable constraints on the latter's behavior. To what extent can the Jumbo compiler verify safety of the generated code? For that matter, to what extent can it be verified at code-generation time? These questions are the subjects of current research.

In the meantime, the system offers a good deal of power and is, we believe, very encouraging for the use of the two-level approach to code-generating components.

References

[1] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Fifth International Conference on Software Reuse*, pages 143–153. IEEE Computer society, June 1998.

[2] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynaic code generation. In *Conference Record of POPL 1996*, pages 131–144, January 1996.

[3] Michael Franz , Thomas Kistler . Slim binaries. Communications of the ACM 40:12, December 1997, 87–94

[4] S. Frederick, G. Dan, M. Greg, H. Luke, and J. Trevor. Compiling for run-time code generation. Technical report, Cornell University, October 2000.

[5] Sam Kamin, Miranda Callahan, and Lars Clausen. Lightweight and generative components I: Source-level components. In *Generative and Component-Based Software Engineering (GCSE'99)*, volume 1799 of *LNCS*, pages 49–62, September 1999.

[6] Sam Kamin, Miranda Callahan, and Lars Clausen. Lightweight and generative components ii: Binary-level components. 1999.

[7] Ralph Keller and Urs Hölzle. Binary Component Adaptation. Proc. ECOOP 98, Springer-Verlag LNCS 1445, pages 107–329, 1998.

[8] Ulrik Pagh Schultz, Julia L. Lawall, Charles Consel, and Gilles Muller. Towards automatic specialization of Java programs. *Lecture Notes in Computer Science*, 1628:367–390, 1999.

[9] Peter Sestoft. Runtime Code Generation with JVM and CLR. Unpublished. Available at http://www.dina.dk/ sestoft/publications.html. 2002.

[10] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of PEPM-97*, volume 32 of *ACM SIGPLAN Notices*, pages 203–217, June 1997.

[11] S. Yamazaki, E. Shibayama. Runtime Code Generation for Bytecode Specialization of Reflective Java Programs. Position paper for ECOOP'2002 Workshop on Generative Programming, University of Mlaga, Spain. June 10-14, 2002.

Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)
0-7695-1913-X/03 $17.00 © 2003 **IEEE**