

Generating Optimized Residual Code in Run-Time Specialization

Hidehiko Masuhara¹ and Akinori Yonezawa²

¹ Department of Graphics and Computer Science
Graduate School of Arts and Sciences, University of Tokyo
masuhara@acm.org

² Department of Information Science, University of Tokyo
yonezawa@is.s.u-tokyo.ac.jp

Abstract. *Run-time specialization* (RTS) techniques efficiently generate specialized programs with respect to run-time values. They construct compiled native-code fragments called *templates* at compile-time, and generate a specialized program by merely copying the templates. The generated programs are less efficient than those generated by static partial evaluation techniques because the RTS techniques prevent many optimizations.

The proposed *bytecode specialization* (BCS) technique is used to generate programs in a *bytecode language* and then translate the generated bytecode into native code by using a just-in-time (JIT) compiler. Its advantages are (1) efficient specialization processes that are similar to those of RTS techniques, (2) efficient specialized programs thanks to optimizations by the JIT compilers, and (3) that it is independent of the source-to-bytecode and the bytecode-to-native compilers thanks to a binding-time analysis algorithm that directly handles bytecode programs.

Thus far, a BCS system has been implemented for a Java virtual machine subset. Micro-benchmarking showed that BCS with JIT compilation generates specialized programs that run more than 3-times faster than ones generated by traditional RTS techniques and that this specialization process takes less than one second.

1 Introduction

Given a generic program and the values of some parameters, *program specialization* techniques generate an optimized program with respect to the values of those parameters. *Partial evaluation*[10,15], which specializes programs at the source-to-source level, has been shown to be useful for optimizing various programs, such as interpreters[19–21], scientific application programs[4], and graphical application programs[13].

* To appear in Technical Report on Partial Evaluation and Program Transformation Day (PE Day'99), Nov. 1999.

Run-time specialization (RTS)¹ techniques[6–9, 16, 17, 22–24] efficiently specialize programs at run-time (1) by constructing a dedicated *specializer* for each target program at compile-time and (2) by directly manipulating native machine code—without using source-code—at specialization-time. The improved specialization speed enables programs to be specialized by using values that are computed at run-time, which means that RTS provides more specialization opportunities than (compile-time) partial evaluators. Programs written in modern programming styles, such as with a component-based architecture, tend to have generic program components and to be composed at run-time. They would thus benefit greatly from optimizations by RTS.

Because RTS is directly manipulates native machine code, it degrades the efficiency of specialized programs and the portability of implementations. In this paper, we describe a novel technique called *bytecode specialization* (BCS) that generates specialized programs as efficiently as RTS and that generates more efficient specialized programs than RTS. In BCS, a specializer generates a specialized program in bytecode, which is then translated into optimized native code. The specializer is constructed by analyzing the bytecode program; *i.e.*, its construction does not depend on the semantics of the source-level language.

We choose the Java virtual machine language (JVML)[18] as the target bytecode language; BCS is designed to construct a specializer in JVML by analyzing JVML programs. Unlike the analysis of programs in high-level programming languages, the analysis of programs in JVML may be complicated due to the operand stack and local variables in JVML. Our binding-time analysis algorithm uses typing rules that are extended from a typing system for JVML[25, 26] and uses control-flow analysis to incorporate constraints for side-effects. We have implemented a prototype BCS system for a JVML subset.

The rest of the paper is organized as follows. Section 2 overviews existing RTS techniques and their problems. BCS is described in Section 3. Section 4 presents a restricted version of JVML. Our binding-time analysis algorithm is explained in Section 5, and the construction of specializers in explained in Section 6. Section 7 presents the current status of our prototype implementation and the results of our micro-benchmarks. Section 8 discusses related studies. Section 9 concludes the paper.

2 Run-Time Specialization

2.1 Program Specialization

A program specialization system processes programs in two phases: *binding-time analysis* (BTA) and *specialization*. BTA takes a program and a list of the binding-times of arguments of a method in the program and returns an annotated program in which every sub-expression has its binding-time. The binding-time of an expression is *static* if the value can be computed at specialization time or

¹ Also known as *run-time code generation* or *dynamic code generation*.

dynamic if the value is to be computed at execution time. For example, when BTA receives

```
class Power
{ static int power(int x, int n)
  { if (n==0) return 1;
    else return x*power(x,n-1); } }
```

with list [dynamic, static] as the binding-times of `x` and `n`, it adds static annotations to the `if` and `return` statements, to the expressions `n==0` and `n-1`, and to the call to `power`. The remaining expressions, namely the constant `1` in the ‘then’ branch, the variable `x`, and the multiplication, are annotated as *dynamic*.

In the specialization step, the annotated program is executed with the values of the static parameters, and a specialized program is returned as a result. The execution rules are the same as the ordinary ones except for the dynamic expressions. The result of the execution of a dynamic expression is the expressions itself. For example, execution of annotated `power` with argument `3` for static parameter `n` proceeds as follows: it tests “`n==0`”, then selects the ‘else’ branch, computes `n-1`, and recursively executes `power` with `2` (*i.e.*, the current value of `n-1`) as an argument. It eventually receives the result of the recursive call, which should be “`x*x*1`”, and finally returns “`x*x*x*1`” by appending the received result to “`x*`”.

2.2 Description

Both partial evaluation (PE) and run-time specialization (RTS) are program specialization techniques, but efficiency of specialization becomes vital when there are parameter values that can be obtained only at run-time. The efficiency of specialization is measured by the time spent for *online processing*, which begins with static parameters, and finishes with the generation of an executable specialized program.

RTS improves the specialization speed by changing how the static expressions are evaluated and when the dynamic expressions are compiled.

PE interprets static expressions because the annotated source program includes both static and dynamic expressions and compiles dynamic expressions because it is source-to-source program transformation (Figure 1 (a)).

RTS, on the other hand, constructs a specializer that executes the static expressions and native-code templates before the online processing. A specializer is a program consisting of static expressions in the annotated program. A native-code template is a compiled code fragment of a dynamic expression. During online processing, the compiled specializer generates an executable specialized program by merely *copying* the templates (Figure 1 (b)).

Comparing PE and RTS, expensive operations including interpretation and compilation are moved to the earlier stage (*i.e.*, offline processing) from online processing (shown as shadowed steps in Figure 1) in RTS. Previous studies showed that RTS systems need to execute dramatically fewer instructions for

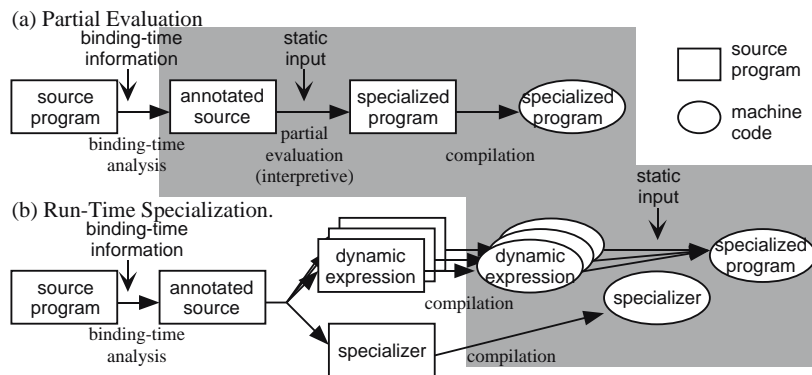


Fig. 1. Overviews of PE and RTS.

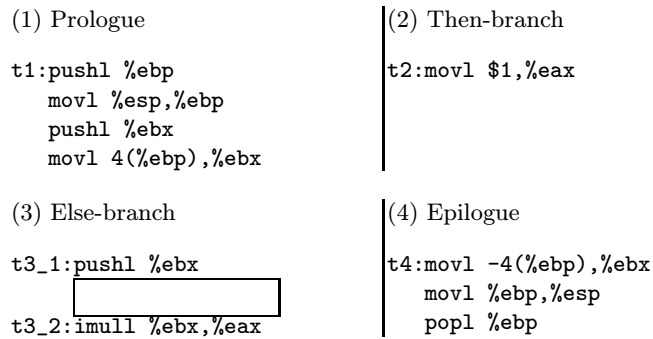


Fig. 2. Constructed templates for power.

```

code *power_gen(int n, code *p)
{ p=generate(p,t1);
  if (n==0) p=generate(p,t2);
  else { p=generate(p,t3_1);
        p=power_gen(n-1,p);
        p=generate(p,t3_2); }
  return generate(p,t4); }

```

Fig. 3. Constructed code generator for `power`.

having specialized programs during online processing than PE with traditional compilers do[7–9, 16, 17, 22, 23].

Templates can be constructed by extracting appropriate fragments from the output of traditional compilers. Figure 2 shows the templates for `power` in Intel x86 mnemonic; they were extracted from the output of a C compiler. In addition to the dynamic expressions in both conditional branches ((2) and (3)), templates corresponding to the prologue (1) and the epilogue (4) of the method are defined for passing parameters among templates. The rectangle in the third template shows the position where the code generated by the recursive call to `power` is inserted.

A *specializer* is a function that uses static parameters and the pointer at which the specialized code is to be generated. The body of a specializer has the same expressions as the original method, except for the dynamic ones. A dynamic expression in the original method is replaced with an expression that copies a corresponding template to memory. Figure 3 shows the specializer for `power`, where “`generate(p,t1)`” copies the binary representation of template `t1` (which corresponds to the label `t1` in Figure 2) to the memory block beginning from `p` and returns the address for the subsequent code generation.

2.3 Problems

Efficiency One of problems with RTS is that the programs it generates are less efficient than those generated by PE with traditional compiler. RTS generates native machine instructions in templates before specialization, so optimizations across templates, such as register allocation and instruction scheduling, cannot be applied. There is thus implicit overhead when the program invokes many methods. Although RTS can generate inline methods by concatenating caller and callee templates, instructions that save/restore local values shuffle registers are inserted between the templates. Those instructions impose nearly the same overheads as with method invocation. Such overhead is not found in programs that are specialized by source-level partial evaluation techniques because optimization can be performed online.

In the example illustrated in Figures 2 and 3, overhead occurs for (i) `pushl %ebx` (third template), which passes the value of `x` to the subsequent template and (ii) the instructions in the first and fourth templates that save/recover

registers. Our micro benchmarking (Section 7) showed that the RTS version is approximately 3-times slower than the PE version. In addition, our previous work showed that RTS versions have approximately 40% more overhead in more practical applications[27, 28].

Portability Another problem with RTS is its strong dependency on both the target language and target machine architecture. To the best of the authors' knowledge, most RTS systems perform BTA in high-level languages. As a result, in order to develop an RTS system for a new high-level language, one need to invent BTA system for that language.

RTS systems also have customized compiler for constructing specializers, which may require deep knowledge of internals of compilers and, at least, knowledge of the native machine-language. For example, Fabius[16, 17] and DyC[11, 12] have special compilers that generate native machine instructions. Note that there are several studies that alleviates this problems. Tempo constructs specializers by editing outputs of a standard C compiler[6]. ICODE is a retargetable intermediate language that can be used as a back-end of RTS[22, 23]. It thus would be beneficial to develop RTS techniques for highly-portable intermediate languages, like Java virtual machine language.

3 ByteCode Specialization

Our proposed *bytecode specialization* (BCS) technique overcome the two problems described above. It performs run-time specialization in a virtual machine language and efficiently translates specialized code into optimized native code by using just-in-time (JIT) compilation techniques. The characteristics of BCS are

efficient specialization: Similar to RTS, BCS constructs specializers that directly generate specialized programs in compiled code.

efficient specialized programs: Thanks to optimization after code generation, specialized programs have less overhead than RTS-generated ones.

portability: Thanks to our BTA algorithm for a virtual machine language, BCS uses the virtual machine language both as an input language and as an output language. In other words, compilers from high-level source language to virtual machine language, and from virtual machine language to native-machine languages can be developed independently from BCS.

As a virtual machine language, we choose the Java virtual machine language (JVML)[18], which has the following advantages:

- Since it is basically a stack-machine language, composing code fragments, which is required for unfolding method calls at specialization time, is easier.
- Compilers from various high-level languages to JVML and JIT compilers for various machine architectures can be exploited[3, 5, 14, 29].

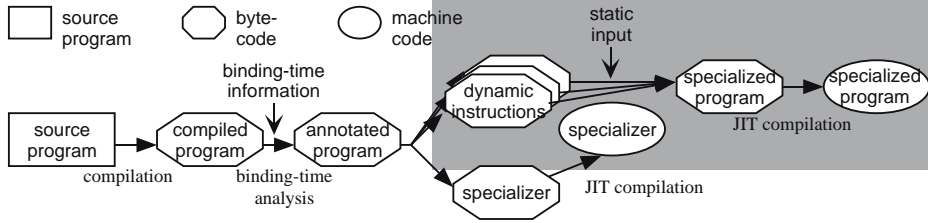


Fig. 4. Overview of BCS.

As shown in Figure 4, a compiler first translates a source program written in a high-level language (*e.g.*, Java) into JVMML bytecode. The compiled program is annotated by using our BTA algorithm. From the annotated program, a *specializer* for generating the *dynamic instructions* is constructed. During online processing, the *specializer* takes the values for the *static parameters* and generates a specialized program in bytecode by writing the dynamic instructions in an array. Finally, the JVM’s class loader and the JIT compiler translate the bytecode specialized program into machine code, which can be executed as a method in the Java language.

4 Target Language (JVMLi)

4.1 Instruction Set

For the sake of simplicity, we present our analysis and *specializer* construction in JVMLi, a restricted version of JVMML. JVMLi is a stack-machine language with local variables and methods. It is different from the full version of JVMML in the following respects:

- All values have type `int` (*i.e.*, objects, arrays, etc. are not used).
- All methods are *class methods* (*i.e.*, methods are declared `static`).
- Subroutines (`jsr` and `ret`), exceptions, and multi-threading are not used.

A JVMLi program is a map that can be addressed by a program counter, which is a pair of a method name and an instruction address in the method:

$$P : method \times \mathcal{N} \rightarrow instruction.$$

JVMLi has seven instructions:

$$\begin{aligned}
 instruction ::= & \text{iconst } n \mid \text{iadd} \mid \text{iload } x \mid \text{istore } x \\
 & \mid \text{ifne } L \mid \text{invoke } m \ n \mid \text{ireturn}.
 \end{aligned}$$

Roughly, an instruction first pops zero or more values off the stack, performs computation, and pushes zero or one value onto the stack. A frame is a set of

```

Method int Power.power(int,int)
0 iload 1          // push n
1 ifne 4           // go to if n ≠ 0
2 iconst 1         // (case n = 0) push 1
3 ireturn          // return 1
4 iload 0          // (case n ≠ 0) push x
5 iload 0          // push x (1st arg.)
6 iload 1          // push n
7 iconst 1         // push 1
8 isub             // compute n - 1 (2nd arg.)
9 invoke Power.power 2 // call method power
10 imul            // compute x × (return value)
11 ireturn         // return x × (return value)

```

Fig. 5. Method `power` in `JVMLi`.

variables local to each method invocation. The `iconst n` instruction pushes a constant n onto the stack. The `iadd` instruction pops two values off the stack and pushes the sum of them onto the stack. Other arithmetic instructions, such as subtraction (`isub`), multiplication (`imul`), and division (`idiv`), have the similar semantics. The `iload x` instruction pushes the current value of local variable x onto the stack. The `istore x` instruction pops a value off the stack and assigns it to local variable x . The `ifne L` instruction pops a value off the stack and jumps to address L in the current method if the value is not zero. The `invoke m n` instruction invokes method m with the first n values on the stack as arguments². The instruction (1) saves the current frame and program counter, (2) pops n values off the stack and assigns them into variables $0, \dots, (n - 1)$ in a newly allocated frame, and (3) jumps to the first address of method m . The `ireturn` instruction disposes of the current frame and restores the saved one, then jumps to the next address of the saved program counter. The caller can use the value at the top of the stack as a return value.

Figure 5 shows the result of compiling method `power` (Section 2.1) into `JVMLi`.

5 Binding-Time Analysis

5.1 Strategy

Our BTA algorithm takes a `JVMLi` program and the binding-time (*i.e.*, *static* or *dynamic*) for each argument of a method and returns the binding-time of each instruction in the program. The returned binding-times must be *consistent*; *i.e.*, static instructions use only static values, and dynamic instructions never generate static values. Local variables complicate the analysis because (1) `JVML` allows polymorphic usage of local variables in a method, *i.e.*, a local variable can

² It corresponds to `invokestatic` in `JVML`.

have different binding times in a method from the viewpoint of BTA, and (2) they are mutable, *i.e.*, the side-effects of assignments must be treated properly.

Our solution is to define constraints among the binding-times of instructions, stacks, and local variables as typing rules and to incorporate the constraints on the side-effects of assignments by using the result of control-flow analysis. The result of BTA is a minimal solution of the constraints.

The typing rules of our BTA algorithm are based on the type system of JVMML proposed by Stata and Abadi[25, 26], but also includes instructions for method invocation.

5.2 Typing Rules

The typing rules use S and D , respectively, for static and dynamic values. The instruction types are also S and D ; the sub-typing relation between them is $S \leq D$. A stack type is a string of value types, either ϵ or $\alpha \cdot \sigma$, where ϵ is the empty stack type, α is a value type, and σ is a stack type. A frame type is the map from a local variable to a value type. An extension of a value f of frame type is defined as

$$(f[x \mapsto \alpha])[y] \equiv \text{if } x = y \text{ then } \alpha, \text{ else } f[y],$$

where x and y are local variables and α is a value type. An empty map is written as ϕ , and we abbreviate $\phi[x_1 \mapsto \alpha_1][x_2 \mapsto \alpha_2] \cdots$ as $[x_1 \mapsto \alpha_1, x_2 \mapsto \alpha_2, \dots]$. Program P is a vector of instructions indexed by program counters. Program counter pc is a pair of method name m and instruction address i in a method; it can be written as $\langle m, i \rangle$. Initial binding-time assignment I is given as a map from a method name to a frame type.

Program P has valid binding-times with respect to initial binding-time assignment I if there are vectors A, R, B, F , and T that satisfy the following judgment:

$$A, R, B, F, T \vdash P, I \quad ,$$

where A, R, B, F , and T are the types of method arguments, return values, instructions, frames, and stacks, respectively.

The rule to prove the judgment is defined as the judgment for all methods, and constraints of I :

$$\frac{\forall m \in \text{Methods}(P). A, R, B, F, T, m \vdash P, \quad \forall m' \in \text{Dom}(I). I[m'] \subseteq F[\langle m', 0 \rangle]}{A, R, B, F, T \vdash P, I} \quad ,$$

where $\text{Dom}(I)$ is the domain of map I , $\text{Methods}(P)$ is a set of method names in P , and the inclusion between maps is defined as

$$\frac{\forall x \in \text{Dom}(f). f[x] = f'[x]}{f \subseteq f'} \quad .$$

The judgment of a method is defined by the rule

$$\frac{\forall i \in \text{Addresses}(P, m). A, R, B, F, T, \langle m, i \rangle \vdash P \quad T_{\langle m, 0 \rangle} = \epsilon \quad F_{\langle m, 0 \rangle} = A_m}{A, R, B, F, T, m \vdash P} \quad ,$$

$$\begin{array}{c}
\begin{array}{ccc}
P[pc] = \mathbf{iconst} \ n & P[pc] = \mathbf{iadd} & P[pc] = \mathbf{iload} \ x \\
F_{pc+1} \subseteq F_{pc} & F_{pc+1} \subseteq F_{pc} & F_{pc+1} \subseteq F_{pc} \\
T_{pc+1} = \alpha \cdot T_{pc} & T_{pc} = \alpha \cdot B[pc] \cdot \sigma & T_{pc+1} = \alpha \cdot T_{pc} \\
B[pc] \leq \alpha & \alpha \leq B[pc] \leq \beta & F_{pc}[x] = B[pc] \leq \alpha
\end{array} \\
\hline
A, R, B, F, T, pc \vdash P & A, R, B, F, T, pc \vdash P & A, R, B, F, T, pc \vdash P
\end{array}$$

$$\begin{array}{ccc}
\begin{array}{c}
P[pc] = \mathbf{istore} \ x \\
F_{pc+1} \subseteq F_{pc}[x \mapsto B[pc]] \\
\alpha \cdot T_{pc+1} = T_{pc} \\
\alpha \leq B[pc]
\end{array} &
\begin{array}{c}
P[pc] = \mathbf{ifne} \ L \\
F_{pc+1} \subseteq F_{pc}, \quad F_L \subseteq F_{pc} \\
\alpha \cdot T_L = \alpha \cdot T_{pc+1} = T_{pc} \\
\alpha \leq B[pc]
\end{array} &
\begin{array}{c}
P[pc] = \mathbf{invoke} \ m \ n \\
F_{pc+1} \subseteq F_{pc} \\
T_{pc} = A_m[n-1] \cdots A_m[0] \cdot \sigma \\
T_{pc+1} = \alpha \cdot \sigma \\
R[m] \leq \alpha
\end{array} \\
\hline
A, R, B, F, T, pc \vdash P & A, R, B, F, T, pc \vdash P & A, R, B, F, T, pc \vdash P
\end{array}$$

$$\begin{array}{c}
P[pc] = \mathbf{ireturn} \\
T_{pc} = \alpha \cdot \sigma \\
\alpha \leq R[m] \\
pc = \langle m, i \rangle \\
\hline
A, R, B, F, T, pc \vdash P
\end{array}$$

Fig. 6. Typing rules for BTA

where $Addresses(P, m)$ is a set of all instruction addresses of method m in program P . The first hypothesis is a local judgment applied to each instruction address in a method. The second and third are the initial conditions on the stack and the frame.

Figure 6 shows rules for a judgment $A, R, B, F, T, pc \vdash P$, which are explained as follows.

- When the instruction at pc pushes a value onto the stack, the type of the top value of the stack at the next address is larger than or equal to the instruction's type. Those instructions include **iconst**, **iadd**, **iload**, and **invoke**.
- When the instruction at pc pops values off the stack, the type of the instruction is larger than or equal to the types of values on the stack at pc . Those instructions include **iadd**, **istore**, and **ifne**.
- When the control moves from pc to pc' , types of live local variables are propagated—*i.e.*, a type of a local variable at pc' must be the same as that of the variable after executing the instruction at pc . This is represented by a constraint among frame types, like $F_{pc+1} \subseteq F_{pc}$. For a jump instruction (**ifne**), the frame types at the current and jump addresses are also constrained by the same inclusion.

When the instruction reads or writes a local variable, the type of the instruction and that of the variable are constrained³.

- When the instruction at pc is `invoke`, the argument types of the target method and the types of values on the current stack are the same, and the return type of the method and type to the top value at the next address of the `invoke` instruction are the same. Since our rules require that a method have only one combination of argument types, the BTA is *monovariant* analysis.
- When the instruction at pc is `ireturn`, the value on top of the current stack has the same type as the return type of the current method.

Side-effects: The typing environments that satisfy the above rules are consistent in terms of original execution order. That is, when an annotated program is executed disregarding the annotations, the program does not “go wrong.” However, a specializer, which is constructed from an annotated program, does not have exactly the same execution order as the original program—it executes both branches of a dynamic conditional jump. As a result, the specializer may go wrong with side-effecting operations.

For example, consider a specializer constructed from the following method, specifying that x and y are dynamic and static, respectively:

```
int f(int x, int y)
{ if (x > 0) y = y + 1;
  else      y = y * 2;
  return g(x, y / 2); }
```

Since y is static, our typing rules give static types to the expressions $y=y+1$, $y=y*2$, and $y/2$. However, a correct BTA would give a dynamic type $y/2$ because the value of y after the `if` statement depends on the value of x , which is dynamic.

We solve this problem by adding the following constraints. Assume $P[pc] = \text{if } L$. Let M_{pc} be the set of addresses at which execution paths from $pc + 1$ and L merge. For each $pc' \in M_{pc}$, let $l_{pc'}$ be the set of local variables that are updated by the `istore` instruction during the execution paths from $pc + 1$ to pc' and L to pc' , and $n_{pc'}$ be the maximum number of stack entries at pc' that are pushed during the execution paths from $pc + 1$ to pc' and L to pc' . The following constraint requires that the local variables and stack entries that are computed in a branch of a dynamic conditional jump be dynamic after the merger of its branches:

$$\forall pc' \in M_{pc}. T_{pc'} = \beta_1 \cdots \beta_{n_{pc'}} \cdot \sigma,$$

$$B[pc] \leq \beta_1 \quad (i = 1, \dots, n_{pc'}) \quad \forall x \in l_{pc'}. B[pc] \leq F_{pc'}[x],$$

where the second constraint specifies that the top $n_{pc'}$ stack entries at pc' have larger types than that of the instruction at pc , and the third specifies that the updated local variables have larger types than that of the instruction at pc .

³ The current rules prohibit ‘lifting’ the values of local variables for simplicity.

| P | B | T | P | B | T |
|-----------------------------|-----|--|----------------------|-----|--|
| iload 1 | S | ϵ | iconst 1 | S | $S \cdot D \cdot D \cdot \epsilon$ |
| ifne L2 | S | $S \cdot \epsilon$ | isub | S | $S \cdot S \cdot D \cdot D \cdot \epsilon$ |
| L1: iconst 1 | D | ϵ | invoke Power.power 2 | S | $S \cdot D \cdot D \cdot \epsilon$ |
| goto L0 | S | $D \cdot \epsilon$ | imul | D | $D \cdot D \cdot \epsilon$ |
| L2: iload 0 | D | ϵ | goto L0 | S | $D \cdot \epsilon$ |
| iload 0 | D | $D \cdot \epsilon$ | L0: ireturn | S | $D \cdot \epsilon$ |
| iload 1 | S | $D \cdot D \cdot \epsilon$ | | | |
| $R[\text{Power.power}] = D$ | | $\forall pc \in \text{Dom}(F). F[pc] = [0 \mapsto D, 1 \mapsto S]$ | | | |

Fig. 7. BTA result of power.

5.3 Example of BTA

Figure 7 shows an example BTA result of `power` when the binding-times of `x` and `n` are dynamic and static, respectively. The binding-times of instructions and stacks are shown in the “ B ” and “ T ” columns, respectively. The frame at each address has the binding-times described at the bottom of the figure. The BTA result is effectively the same as that of the source-level BTA; *i.e.*, instructions that correspond to a static/dynamic expression at source-level have static/dynamic types, respectively.

6 Specializer Construction

From a BTA-annotated program, a specializer is constructed in JVMIL. Basically, the specializer contains the static instructions in the annotated program and *instruction-generating* instructions corresponding to the dynamic instructions in the annotated program. Here, we describe the construction of a specializer by using pseudo-instructions, which are eventually translated into sequences of JVMIL instructions. The specializer is executable as a Java method.

The extended JVMIL for defining specializers contains all the JVMIL i instructions and six pseudo-instructions:

$$\begin{aligned}
 \textit{instruction}_g ::= & \textit{instruction} \mid \text{GEN } \textit{instruction} \mid \text{LIFT} \mid \text{LABEL } L \\
 & \mid \text{SAVE } n [x_0, \dots] \mid \text{RESTORE} \mid \text{INVOKEGEN } m [x_0, \dots] \quad ,
 \end{aligned}$$

where *instruction* is a set of instructions in JVMIL i . A specializer is constructed by translating each annotated instruction as follows.

- Static instruction i becomes instruction i of the specializer.
- Dynamic instruction i is translated into pseudo-instruction `GEN i` . When `GEN i` is executed at specialization time, the binary representation of i is written in the last position of the specified array.

| <i>P</i> | <i>B</i> | <i>T</i> | |
|---------------------------------------|----------|----------------------------|----------------------------|
| 0 <code>iload 0</code> | <i>S</i> | ϵ | <code>mult_gen(int)</code> |
| 1 <code>iload 1</code> | <i>D</i> | $D \cdot \epsilon$ | 0 <code>iload 0</code> |
| 2 <code>imul</code> | <i>D</i> | $D \cdot D \cdot \epsilon$ | 1 <code>LIFT</code> |
| 3 <code>ireturn</code> | <i>S</i> | $D \cdot \epsilon$ | 2 <code>GEN iload 1</code> |
| $F_{pc} = [0 \mapsto S, 1 \mapsto D]$ | | | 3 <code>GEN imul</code> |
| | | | 4 <code>ireturn</code> |

| | |
|--|--------------------------------------|
| (a) BTA result of <code>int</code> <code>mult(int, int)</code> . | (b) Con- structed specializer. |
|--|--------------------------------------|

Fig. 8. Example LIFT instruction.

- When an instruction has a different type than that of the value pushed or popped by the instruction, pseudo-instruction `LIFT` is inserted. More precisely, (1) when a static instruction at pc pushes a value onto the stack and $T[pc + 1] = D \cdot \sigma$, `LIFT` is inserted *after* the instruction; (2) when a dynamic instruction at pc pops a value off the stack and $T[pc] = S \cdot \sigma$, `LIFT` is inserted *before* the instruction. The execution of a `LIFT` instruction pops value n off the stack and generates instruction “`iconst n`” as an instruction of the specialized program.

Figure 8 shows an example `LIFT` instruction. Method `int mult(int, int)` computes the product of two integers. The BTA result of `mult` with initial binding-time assignment $[0 \mapsto S, 1 \mapsto D]$ is presented in Figure 8(a).

Since local variable 0 is static, the type of `iload 0` at address 0 is also static. The loaded value, which appears at the top of the stack at address 1, has a dynamic type because the dynamic `imul` instruction at address 2 uses it. Therefore, a `LIFT` instruction is inserted at address 1 of specializer `mult_gen` (Figure 8(b)).

When `mult_gen` is invoked with an argument, say 3, the `iload 0` instruction at address 0 pushes 3 onto the stack. Then the `LIFT` instruction pops 3 off the stack and generates “`iconst 3`”. As a result, a specialized program has an instruction sequence of `iconst 3`, `iload 1`, and `imul`, in which the static value 3 is “lifted.”

- Static `invoke m` is translated into pseudo-instruction `INVOKEGEN m [x0, x1, ...]`, where x_0, x_1, \dots are the dynamic local variables at the current address. When `INVOKEGEN` is executed, (1) instructions that save local variables x_0, x_1, \dots to the stack and move values on top of the stack to the local variables are generated, (2) a specializer for m is invoked, and (3) instructions that restore saved local variables x_0, x_1, \dots are generated. The number of values moved from the stack to the local variables in (1) is the number of dynamic arguments of m .
- When dynamic conditional jump `ifne L` is dynamic, the specializer has an instruction that generates `ifne`, followed by the instructions for the ‘then’ and ‘else’ branches. In other words, it generates specialized instruction se-

| | |
|--|---|
| <pre> Method Power.power_gen(int) iload_1 ifne L5 L2:GEN iconst_1 L4:ireturn L5:GEN iload_0 GEN iload_0 </pre> | <pre> iload_1 iconst_1 isub INVOKEGEN Power.power_gen 2 [0] GEN imul L13:ireturn </pre> |
|--|---|

Fig. 9. Specializer definition with pseudo-instructions.

quences of both branches, one of which is selected by the dynamic condition. First, the jump instruction is translated into two pseudo-instructions: `GEN ifne L` and `SAVE n [x0, x1, ...]`, where n and $[x_0, x_1, \dots]$ are the number of static values on the stack that will be popped during the execution of the ‘then’ branch and a list of static local variables that may be updated during execution of the ‘then’ branch, respectively. In addition, pseudo-instruction sequence `LABEL L; RESTORE` is inserted at label L . When `SAVE` is executed at specialization time, the top n values on the current stack and the local variables x_0, x_1, \dots are saved. The execution of `RESTORE` resets the saved values on the stack and in the frame.

Figure 9 shows the definition of specializer `power_gen` with pseudo-instructions, constructed from method `power`.

The specializer definition is further translated into a Java method so that it takes (1) an array `byte[] code` in which instructions of the specialized program are written, (2) an `int` index `i` that indicates at which `code` index the next instruction should be written, (3) an object `ConstantPool cp` that manages a ‘constant pool,’⁴ and (4) the static arguments of the original method. When invoked, the method writes instruction `code` from index `i` and returns the index immediately after the last instruction is generated. The definition of the specializer for `power` is shown in Appendix A.

Figure 10 shows the instructions for specialized `power` with 2 as a static argument. Some sequences are unnecessary, such as those that swap local variables and stack entries. They would be eliminated by the JIT compiler. In fact, our micro-benchmarking showed that the execution of the specialized program after JIT compilation has no overheads due to unnecessary instructions.

7 Current Status and Performance Measurement

We have implemented a prototype BCS system for `JVMLi`, the restricted version of `JVML`. As mentioned, a specializer generates specialized instructions into a `byte` array. After attaching the information needed to form a `ClassFile` format

⁴ A constant pool is storage that keeps values of constants in a JVM class file.

| | | | | | | | | |
|--------|-----|------------|--|----|----------|--|----|----------|
| Method | int | power(int) | | 8 | istore_1 | | 16 | imul |
| 0 | | iload_0 | | 9 | iload_0 | | 17 | istore_1 |
| 1 | | iload_0 | | 10 | iload_1 | | 18 | istore_0 |
| 2 | | istore_1 | | 11 | istore_0 | | 19 | iload_1 |
| 3 | | iload_0 | | 12 | iconst_1 | | 20 | imul |
| 4 | | iload_1 | | 13 | istore_1 | | 21 | ireturn |
| 5 | | istore_0 | | 14 | istore_0 | | | |
| 6 | | iload_0 | | 15 | iload_1 | | | |
| 7 | | iload_0 | | | | | | |

Fig. 10. Specialized version of `power`.

of JVM, JVM's class loader transforms the array into an executable method, performing JIT optimizations⁵.

We measured the performance of both specialized programs and specialization processes using the system.

The performance of specialized programs was measured by executing codes of `power(x, 30)` (1) without any specialization, (2) specialized by BCS or RTS, or (3) specialized offline (*i.e.*, compiled after specializing the source-level program). The programs were executed in both Java and C. The code specialized by RTS in C was generated by hand based on existing RTS techniques. The source-level specialization for (3) was also done by hand.

All programs were executed on a lightly loaded 300 MHz Pentium II PC/AT compatible with 256 MB memory, running MS-WindowsNT 4.0. The Java versions were compiled and executed on a Sun JDK 1.1.7 with a Symantec JIT compiler. The C versions were compiled using Cygnus GCC 2.7.2 with `-O2` option. In order to exclude startup times and JIT compilation times for measuring the performance of specialized programs, the benchmark program first executes a specialized code, and then measures the time spent for a loop that repeatedly executes the code.

As shown in Table 1(a), the BCS-generated program optimized by the JIT compiler ran approximately 3-times faster than the unspecialized one. The efficiency of the program was close to that of programs specialized and compiled offline. The BCS-generated program executed *without* JIT compilation and the traditional RTS generated program were significantly slower than the statically specialized programs. As discussed in the previous section, this slowness is due to the overhead of instructions that save local information and that pass parameters between the code fragments generated by different specializers.

Table 1(b) shows the specialization times of BCS on a JVM with and without a JIT compiler. The first row (i) shows the total time spent for specialization, which can be divided into (ii) the execution times of the specializers and (iii) the execution times of other routines, such as class loading and JIT compilation. The small difference in the latter execution times (iii) between "with JIT" and "without JIT" suggests that the overhead of JIT compilation is low.

⁵ The system also works with JVMs without JIT compilers.

| specialization | BCS | | traditional RTS in C |
|----------------|----------|-------------|-------------------------|
| | with JIT | without JIT | |
| unspecialized | 0.74 | 11.8 | 1.47 |
| run-time | 0.25 | 7.1 | 0.83 |
| offline | 0.21 | 1.6 | 0.26 |

(a) Execution times of specialized programs
(not include JIT compilation time).

| | with JIT | without JIT |
|----------------------------|----------|-------------|
| (i) total time | 610 | 724 |
| (ii) specializer execution | 50 | 142 |
| (iii) other routines | 560 | 582 |

(b) Specialization times.

Table 1. Execution and specialization times of method `power(x,30)` (μsec).

8 Related Work

Wickline *et al.* proposed an RTS system that generates instructions in a stack virtual machine language[30]. In their system, a specializer is constructed from a source-level program by using a customized compiler, while BCS uses an existing compiler from source-to-JVML. To the best of our knowledge, the specialized instructions in their system are executed in an interpretive manner.

Leone and Lee suggested in their study on FABIUS, an RTS system for a subset of ML, that register allocation at specialization time would be beneficial [16, 17]. Poletto *et al.* proposed a retargetable, intermediate-machine language called ICODE[22, 23] that can be used as a back-end of run-time specialization, which performs optimizations such as register allocation after specialization. Although the advantages of doing this seem to be similar to those of our system, our approach successfully optimizes across inlined method invocations by choosing a stack-type virtual machine.

Sperber and Thiemann showed that a specializer that generates programs at the machine-code level can be derived by composing a source-level specializer generator (a so-called ‘cogen’) and a compiler[24]. In their approach, the simplicity of the compiler is essential to ensuring a successful composition; the application of their approach to practical optimizing compilers would thus be difficult.

9 Conclusion

In this paper, we proposed bytecode specialization (BCS), which specializes Java virtual machine language (JVML) programs at run-time. The advantages of this approach are (1) efficient specialization thanks to a specializer constructed from a given program and thanks to code generation at the virtual machine language level and (2) efficient specialized programs thanks to the optimization applied during translation from virtual-machine code to native-machine code. In addition, our binding-time analysis (BTA) algorithm for JVML makes BCS independent of the source-to-bytecode and bytecode-to-native-code compilers, while traditional run-time specialization (RTS) techniques are highly dependent on the semantics of the source-level language and the internals of the compiler. The BTA algorithm uses typing rules that are extensions of those in Stata and Abadi's typing system for JVML. It also uses flow analysis to correctly handle stacks, local variables, and local side-effects.

Thus far, we have implemented a prototype BCS system for a JVML subset and have shown that programs specialized by our system are as efficient as those specialized offline. Our micro-benchmarking showed that BCS with just-in-time (JIT) compilation generates programs as efficient as those specialized offline. BCS also exhibits significantly better specialization speed than source-level partial evaluators with compilers, although it is still room for improvement. We are now extending our rules for the full version of JVML. Since the analysis described in the paper only covers primitive types, an analysis that properly handles references to objects should be done. Furthermore, a *polyvariant* BTA algorithm should be developed in order to specialize programs that include generic routines such as library functions.

Acknowledgments We thank Yuuya Sugita for his discussions on run-time specialization techniques. We also thank Kenjiro Taura, Kenichi Asai, Naoki Kobayashi, and the other members of the Yonezawa research group for their useful comments on early drafts of paper.

References

1. ACM. *Proceedings of Conference on Programming Language Design and Implementation (PLDI'96)*, volume 31(5) of *ACM SIGPLAN Notices*, Philadelphia, PA, May 1996.
2. ACM. *Proceedings of Conference on Programming Language Design and Implementation (PLDI'97)*, volume 32(5) of *ACM SIGPLAN Notices*, Las Vegas, NV, June 1997.
3. N. Benton, A. Kennedy, and G. Russell. Compiling standard ML to Java bytecodes. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, Baltimore, MD, Sept. 1998.
4. A. A. Berlin and R. J. Surati. Partial evaluation for scientific computing: The supercomputer toolkit experience. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 133–141, Orlando, FL, June 1994. ACM SIGPLAN.

Published as Technical Report 94/9, Department of Computer Science, University of Melbourne.

5. P. Bothner. Kawa – compiling dynamic languages to the Java VM. In *USENIX*, New Orleans, June 1998.
6. C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of Symposium on Principles of Programming Languages (POPL96)*, pages 145–170, St. Petersburg Beach, Florida, Jan. 1996. ACM SIGPLAN-SIGACT.
7. D. R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI'96)* [1], pages 160–170.
8. D. R. Engler and T. A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*, pages 263–272, San Jose, CA, Oct. 1994. ACM. (published as SIGPLAN Notices Vol.29, No.11).
9. N. Fujinami. Automatic and efficient run-time code generation using object-oriented languages. *Computer Software*, 15(5):25–37, Sept. 1998. (In Japanese).
10. Y. Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
11. B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Annotation-directed run-time specialization in C. In *Proceedings of Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, volume 32(12) of *ACM SIGPLAN*, pages 163–178, Amsterdam, June 1997. ACM.
12. B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. An evaluation of staged run-time optimization in DyC. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, 1999.
13. B. Guenter, T. B. Knoblock, and E. Ruf. Specializing shaders. In *SIGGRAPH'95 (Computer Graphics Proceedings, Annual Conference Series)*, pages 343–349, 1995.
14. J. C. Hardwick and J. Sipelstein. Java as an intermediate language. Technical Report CMU-CS-96-161, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Aug. 1996.
15. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
16. P. Lee and M. Leone. Optimizing ML with run-time code generation. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI'96)* [1], pages 137–148.
17. M. Leone and P. Lee. Lightweight run-time code generation. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106, Orlando, FL, June 1994. ACM SIGPLAN. published as Technical Report 94/9, Department of Computer Science, The University of Melbourne.
18. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.
19. H. Masuhara. *Architecture Design and Compilation Techniques Using Partial Evaluation in Reflective Concurrent Object-Oriented Languages*. PhD thesis, Department of Information Science, Faculty of Science, University of Tokyo, Jan. 1999.
20. H. Masuhara, S. Matsuoka, K. Asai, and A. Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In M. E. S. Loomis, editor, *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, volume 30(10) of *ACM SIGPLAN Notices*, pages 300–315, Austin, TX, Oct. 1995. ACM.

21. H. Masuhara and A. Yonezawa. Design and partial evaluation of meta-objects for a concurrent reflective language. In E. Jul, editor, *European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1445 of *Lecture Notes in Computer Science*, pages 418–439, Brussels, Belgium, July 1998. Springer-Verlag.
22. M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI'97)* [2], pages 109–121.
23. M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: a language and compiler for dynamic code generation. *Transactions on Programming Languages and Systems*, 21(2):324–369, Mar. 1999.
24. M. Sperber and P. Thiemann. Two for the price of one: Composing partial evaluation and compilation. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI'97)* [2], pages 215–225.
25. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Conference record of Symposium on Principles of Programming Languages*, pages 149–160, San Diego, Jan. 1998. ACM SIGPLAN-SIGACT.
26. R. Stata and M. Abadi. A type system for Java bytecode subroutines. *Transactions on Programming Languages and Systems*, 21(1):90–137, Jan. 1999.
27. Y. Sugita, H. Masuhara, and K. Harada. An efficient implementation of a reflective language using a dynamic code generation technique. In *Proceedings of the JSSST SIGOOC 1998 Workshop on Systems for Programming and Applications (SPA'98)* <http://www.brl.ntt.co.jp/ooc/spa98/proceedings/>, Kusatsu, Japan, Mar. 1998. Japan Society for Software Science and Technology (JSSST). (In Japanese).
28. Y. Sugita, H. Masuhara, K. Harada, and A. Yonezawa. On-the-fly specialization of reflective programs using dynamic code generation techniques. In J.-C. Fabre and S. Chiba, editors, *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, volume 98–4 of *Technical Report of Center for Computational Physics, University of Tsukuba*, pages 21–25, Vancouver, B.C., Canada, Oct. 1998.
29. S. Taft. Programming the Internet in Ada 95. In *Ada Europe'96*, 1996.
30. P. Wickline, P. Lee, and F. Pfenning. Run-time code generation and Modal-ML. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI'98)*, volume 33(5) of *ACM SIGPLAN Notices*, pages 224–235, Montreal, Canada, June 1998. ACM.

A Specializer in JVMML

The method in Figure 11 is a specializer definition translated from the one in Figure 9. The arguments of method `power_gen` are (1) the first argument of `power`, (2) an array in which the generated instructions are written, (3) an index of the array, and (4) a constant pool. The value returned by the method is an array containing an updated index and the maximum stack depth of the generated instructions. The latter return value, which is required by the JVM, must be computed at specialization time since the number of stack entries cannot be predicated statically.

```

method int[] Power.power_gen(int, byte[], int, ConstantPool)
// set stack depth
    iconst_2
    istore 4
L0: iload_0
L1: ifne L5
// GEN iconst_1
L2: aload_1
    iload_2
    iconst_4
    bastore
    iinc 2 1
// return index & depth
L3: iconst_2
    newarray int
    dup
    iconst_0
    iload_2
    iastore
    dup
    iconst_1
    iload 4
    iastore
    areturn
// GEN iload_0
L5: aload_1
    iload_2
    bipush 27
    bastore
    iinc 2 1
// GEN iload_0
L6: aload_1
    iload_2
    bipush 27
    bastore
    iinc 2 1
L7: iload_0
L8: iconst_1
L9: isub
// INVOKEGEN Power.
// power_gen 2 [0]
// GEN istore_2
L10: aload_1
    iload_2
    bipush 61
    bastore
    iinc 2 1
// GEN iload_1
    aload_1
    iload_2
    bipush 27
    bastore
    iinc 2 1
// GEN iload_2
    aload_1
    iload_2
    bipush 28
    bastore
    iinc 2 1
// GEN istore_1
    aload_1
    iload_2
    bipush 60
    bastore
    iinc 2 1
// GEN iload_2
    aload_1
    iload_2
    bipush 28
    bastore
    iinc 2 1
// GEN imul
L11: aload_1
    iload_2
    bipush 104
    bastore
    iinc 2 1
// return index & depth
L12: iconst_2
    newarray int
    dup
    iconst_0
    iload_2
    iastore
    dup
    iconst_1
    iload 4
    iastore
    areturn
L14: istore 4
// GEN istore_2
    aload_1
    iload_2
    bipush 61
    bastore
    iinc 2 1
// GEN istore_1
    aload_1
    iload_2
    bipush 60
    bastore
    iinc 2 1
// GEN iload_2
    aload_1
    iload_2
    bipush 28
    bastore
    iinc 2 1
// GEN imul
L11: aload_1
    iload_2
    bipush 104
    bastore
    iinc 2 1
// return index & depth
L12: iconst_2
    newarray int
    dup
    iconst_0
    iload_2
    iastore
    dup
    iconst_1
    iload 4
    iastore
    areturn
L14: istore 4
// GEN istore_2
    aload_1
    iload_2
    bipush 61
    bastore
    iinc 2 1
// GEN istore_1
    aload_1
    iload_2
    bipush 60
    bastore
    iinc 2 1
// GEN iload_2
    aload_1
    iload_2
    bipush 28
    bastore
    iinc 2 1
// GEN imul
L11: aload_1
    iload_2
    bipush 104
    bastore
    iinc 2 1
// return index & depth
L12: iconst_2
    newarray int
    dup
    iconst_0
    iload_2
    iastore
    dup
    iconst_1
    iload 4
    iastore
    areturn
L14: istore 4

```

Fig. 11. Specializer for power in JVMML.