Run-time Bytecode Specialization

A Portable Approach to Generating Optimized Specialized Code*

Hidehiko Masuhara[1] and Akinori Yonezawa[2]

[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. This paper proposes a *run-time bytecode specialization* (BCS) technique that analyzes programs and generates specialized programs at run-time in an intermediate language. By using an intermediate language for code generation, a back-end system can *optimize the specialized programs after specialization.* As the intermediate language, the system uses Java virtual machine language (JVML), which allows the system to easily achieve practical *portability* and to use sophisticated just-in-time compilers as its back-end. The binding-time analysis algorithm, which is based on a type system, covers a non-object-oriented subset of JVML. A specializer, which generates programs on a per-instruction basis, can perform *method inlining at run-time.* The performance measurement showed that a non-trivial application program specialized at run-time by BCS runs approximately 3–4 times faster than the unspecialized one. Despite the large amount of overheads at JIT compilation of specialized code, we observed that the overall performance of the application can be improved.

1 Introduction

Given a generic program and the values of some parameters, *partial evaluation* techniques generate a specialized program with respect to the values of those parameters[11, 17]. Most of those techniques have been studied as *source-to-source* transformation systems; *i.e.,* they analyze programs in a high-level language and generate specialized programs in the same language. They have been successful in the optimization of various programs, such as interpreters, scientific application programs, and graphical application programs[4, 13, 20].

Run-time specialization (RTS) techniques[10, 12, 18, 22] efficiently perform partial evaluation at run-time (1) by constructing a *specializer* (or a *generating extension*) for each source program at compile-time and (2) by directly generating native machine code at run-time. The drastically improved specialization

* To appear in Proceedings of *Second Symposium on Programs as Data Objects (PADO-II)*, Aarhus, Denmark, May 2001.

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 specialization. Several studies reported that RTS can improve performance of programs for numerical computation[18, 22], an operating system kernel[25], an interpreter of a simple language[18], etc.

One of the problems of RTS systems is a trade-off between efficiency of specialization and efficiency of specialized code. For example, Tempo generates specialized programs by merely copying pre-compiled native machine code. The performance of the generated code is 20% slower than that is generated by compile-time specialization on average[22]. Of course, we could optimize specialized programs at run-time by optimizing generated code after specialization. It however makes *amortization*[1] more difficult.

In this paper, we describe an alternative approach called *run-time bytecode specialization* (BCS), which is an automatic *bytecode-to-bytecode* transformation system. The characteristics of our approach are: (1) the system directly analyzes program and constructs specializers in a bytecode language; and (2) the specializer generates programs in the bytecode language, which makes it possible to apply optimizations *after specialization* by using just-in-time (JIT) compilation techniques.

As the bytecode language, we choose the Java virtual machine language (JVML)[19], which provides us practical portability. The system can use existing compilers as its front-end, and widely available Java virtual machines, which may include sophisticated JIT compilers, as its back-end. The analysis of JVML programs is based on a type system derived from the one for JVML[27]. A specializer can be basically constructed from the result of the analysis, and can perform method inlining at run-time.

Thus far, we have developed our prototype system for a non-object-oriented subset of JVML; the system support only primitive types, arrays, and static methods. Although the system does not yet support important language features in Java, such as objects and virtual methods, it has sufficient functionality to demonstrate fundamental costs in our approach, such as efficiency of specialized code and overheads of specialization and JIT compilation.

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 the performance measurement of our current implementation. Section 5 discusses related studies. Section 6 concludes the paper.

2 Run-Time Specialization

2.1 Program Specialization

An offline partial evaluator processes programs in two phases: *binding-time analysis* (BTA) and *specialization.* BTA takes a program and a list of the binding-

[1] In RTS systems, a specialization process of a procedure is *amortized* if the amount of reduced execution time of the procedure becomes larger than the time elapsed for the specialization process.

times of arguments of a method in the program and returns an annotated program in which every sub-expression is associated with binding-time. The binding-time of an expression is *static* if the value can be computed at specialization time or *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 phase, the annotated program is executed with the values of the static parameters, and a specialized program is returned as a result. The execution rules for static expressions are the same as the ordinary ones. The rule for the dynamic expressions is to return the expression 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 Overview

Run-time specialization techniques efficiently specialize programs by generating specialized programs at machine code level[10, 12, 15, 18, 22, 29].

Given a source program and binding-time information, an RTS system efficiently generates specialized programs at run-time in the following way. It first performs BTA on the source program, similar to compile-time specialization systems. It then compiles dynamic expressions into fragments of machine code, called *templates.* It also constructs a *specializer* that has the static expressions in the source program and operations for copying corresponding templates into memory in place of the dynamic expressions. At run-time, when a specializer is executed with a static input, it executes the static expressions, and directly generates a specialized program at machine code level.

2.3 Problems

Efficiency. There is a trade-off between efficiency of specialization processes and efficiency of specialized code in RTS systems.

A program that is specialized by an RTS system is usually slower than that specialized by a compile-time specialization system. This is because RTS systems rarely apply optimizations, such as instruction scheduling and register allocation,

to the specialized code, for the sake of efficient specialization. Furthermore, programs that have a number of method invocations (or function calls) would be much slower since method inlining is not performed in several RTS systems.

For example, Noël, et al. showed that the run-time specialized programs have 20% overheads over the compile-time specialized ones on average, in their study on Tempo[22]. As we will see in Section 4, the overheads in the run-time specialized program can overwhelm the speedup obtained by specialization; *i.e.,* the specialized program become slower than the original program.

If an RTS system performed optimizations at run-time, specialized programs would become faster. In fact, there are several systems that optimizes specialized code at run-time[2, 18, 24]. However, the time spent for the optimization processes makes amortization more difficult.

Consequently, an RTS system that can flexibly balance a degree of optimization of specialized code and time for generating specialized code would be beneficial.

Portability. In order to directly generate machine code, RTS systems often depend on the target machine architecture. A typical RTS system includes its own compiler from source code (usually in a high-level language) to native machine code.

Several techniques have been proposed to overcome the problem. For example, Tempo uses standard C compilers for creating templates[8, 22]. `C, which is a language with dynamic code generation mechanisms, generates specialized code in *retargetable* virtual machine languages called VCODE and ICODE[24].

3 Run-time Bytecode Specialization

3.1 Overview

Our proposed *run-time bytecode specialization* (BCS) technique uses a virtual machine (bytecode) language as its source and target languages. It takes a bytecode program as its input, and constructs a specializer in the same bytecode language. At run-time, the specializer, which runs on a virtual machine, generates specialized programs in the same bytecode language. As a virtual machine language, we choose the Java virtual machine language (JVML)[19].

We aim to solve the problems in the previous section in the following ways:

Efficiency. Instead of directly generating specialized code in a native machine language, BCS generates it in an intermediate (bytecode) language. When the system is running on a JVM with a JIT compiler, the specialized code is optimized into a native machine language before execution. We also plan to control the quality of specialized code and the speed of JIT compilation processes by integrating our system with JVMs that have interfaces to its JIT compilers, such as OpenJIT[23].
Another functionality of specializers in BCS is that they can perform method inlining at run-time. Although the specialized code with method inlining has

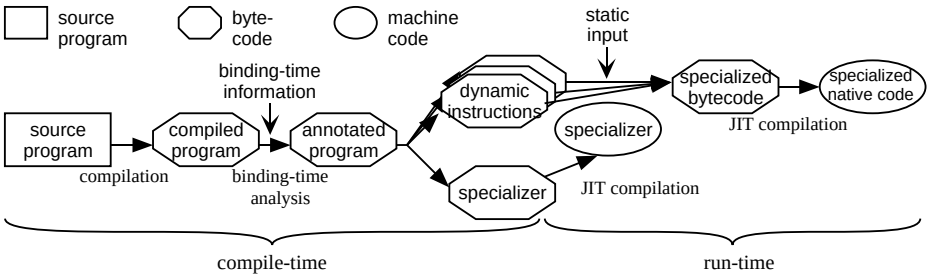

Fig. 1. Overview of BCS.

a certain amount of overheads for saving/restoring local variables at bytecode level, a JIT compiler can remove most of them according to our experiments.

Portability. As is shown in a previous run-time code generation system[9, 24], code generation at the virtual machine level can improve portability. Current BCS system generates specialized code in the standard JVML; the generated code can be executed on JVMs that are a widely available to various platforms.

The input to the BCS system is a JVML program. This means that the system does not depend on the syntax of high-level languages. Instead, run-time specialization can be applied to any language for which there exists a compiler into JVML. In fact, there are several compilers from various high-level languages to JVML[3, 6, etc.], which would be used as a front-end when we extended our system to support the fullset of JVML.

As shown in Figure 1, a compiler first translates a source program written in a high-level language (*e.g.,* Java) into JVML bytecode. The compiled program is annotated by using our BTA algorithm. From the annotated program, a specializer for generating the *dynamic instructions* is constructed. At run-time, 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.

In the following subsections, we present the outline of each process in BCS briefly. More detailed description can be found in the other literature[21].

3.2 Source and Target Language

As mentioned, our source and target language is JVML, which is a stack-machine language with local variables and instructions for manipulating objects. Currently, a subset of the JVML instructions is supported. Restrictions are:

– Only primitive types and array types are supported. (*i.e.,* objects are not supported yet).

```
Method int Power.power(int,int)
 0 iload 1                          // push n
 1 ifne 4                           // go to 4 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 as arg. #0
 6 iload 1                          // push n
 7 iconst 1                         // push 1
 8 isub                             // compute (n − 1) as arg. #1
 9 invokestatic int Power.power(int,int)  // call method
10 imul                             // compute x × (return value)
11 ireturn                          // return x × (return value)
```

Fig. 2. Method `power` in JVML.

- All methods must be *class methods* (*i.e.,* methods are declared `static`).
- Subroutines (`jsr` and `ret`), exceptions, and multi-threading are not supported.

Figure 2 shows the result of compiling method `power` (Section 2.1) into JVML. A method invocation creates a *frame* that holds an *operand stack* and *local variables*. An instruction first pops zero or more values off the stack, performs computation, and pushes zero or one value onto the stack.

The `iconst` n instruction pushes a constant n onto the stack. The `isub` (or `imul`) instruction pops two values off the stack and pushes the difference (or multiple) of them onto the stack. 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 `invokestatic` $t_0\ m(t_1, \ldots, t_n)$ instruction invokes method m with the first n values on the stack as arguments. The `invokestatic` instruction (1) pops n values off the stack, (2) saves the current frame and program counter, (3) assigns the popped values into variables $0, \ldots, (n-1)$ in a newly allocated frame, and (4) jumps to the first address of method m. The `ireturn` instruction (1) pops a value off the stack, (2) disposes of the current frame and restores the saved one, (3) pushes the value on the restored stack, and (4) jumps to the next address of the saved program counter. The caller uses the value at the top of the stack as a returned value.

3.3 Binding-Time Analysis

Strategy Our BTA algorithm is a flow sensitive and monovariant (context insensitive) analysis for the subset of JVML based on a type system. From the viewpoint of BTA, the subset of JVML is mostly similar to high-level imperative languages such as C. Therefore, the analysis should be careful about the following respects, unlike the analyses for functional languages:

– Since compilers may assign different variables to an operand-stack entry or a local variable, the analysis should be *flow sensitive*[16]; *i.e.,* it should allow an operand-stack entry or a local variable to have a different binding-time at each program point in a method.
– As JVML is an *unstructured* language (*i.e.,* it has a 'goto' instruction), merge points of a conditional jump and loops are implicit. The algorithm therefore has to somehow infer this information.

The BTA algorithm is based on a type system, following the algorithms used for functional languages[1, 14]. As the type system, we use a modified version of a type system of JVML proposed by Stata and Abadi[27][2]. The algorithm, which is described in the other literature[21], consists of the following steps:

1. In a given program, for each address in each method, it first gives three type variables to an operand-stack, to a frame of local variables, and to an instruction at the address. By giving different type variables to local variables at each address, the system achieves flow sensitivity, as well as the original Stata and Abadi's system.
2. It then applies typing rules to each instruction of a method, and generates constraints among the type variables.
3. It also generates additional constraints that treat *non-local side-effects under dynamic control*[17, chapter 11] by using the result of a flow analysis.
4. It finally computes a minimal set of assignments to type variables that satisfies all the generated constraints.

Example Figure 3 shows an example BTA result of `power` when the binding-times of `x` and `n` are dynamic and static, respectively[3]. The binding-time of an instruction, which is displayed in the B column, is either S (static) or D (dynamic). The binding-time of a stack, which is displayed in the T column, is written as $\tau_1 \cdot \tau_2 \cdots \tau_n \cdot \epsilon$ (a stack with n values whose types are τ_1, $\tau_2, \ldots$, from the top value). The binding-time of a frame of local variables, which is displayed in the F column, is denoted as $\emptyset$ (an empty frame) or $[i_k \mapsto \tau_k]$ (a frame whose local variable i_k has type τ_k). Note that the domains of the frame types 'shrink' along the execution paths. This is because our BTA rules generate constraints on only types of live local variables, and the types of unused ones do not appear in the result.

The BTA result is effectively the same as that of the source-level BTA; *i.e.,* instructions that correspond to a static or dynamic expression at source-level have the static or dynamic types, respectively.

[2] They design their type system for formalizing the JVM's verification rules in terms of subroutines (`jsr` and `ret`). Here, our current analysis merely uses the style of their formalization, and omits complicated rules for subroutines.

[3] The instruction sequence was slightly modified from Figure 2, so that any conditional jump has merge points within the method. A preprocessor inserts a unique `ireturn` instruction at the end of the method, and replace all `ireturn` instructions with `goto` instructions to the inserted `ireturn` instruction.

	instruction	B	T	F
	`iload 1`	S	ϵ	$[0 \mapsto D, 1 \mapsto S]$
	`ifne L2`	S	$S \cdot \epsilon$	$[0 \mapsto D, 1 \mapsto S]$
`L1` :	`iconst 1`	S	ϵ	$\emptyset$
	`goto L0`	S	$D \cdot \epsilon$	$\emptyset$
`L2` :	`iload 0`	D	ϵ	$[0 \mapsto D, 1 \mapsto S]$
	`iload 0`	D	$D \cdot \epsilon$	$[0 \mapsto D, 1 \mapsto S]$
	`iload 1`	S	$D \cdot D \cdot \epsilon$	$[1 \mapsto S]$
	`iconst 1`	S	$S \cdot D \cdot D \cdot \epsilon$	$\emptyset$
	`isub`	S	$S \cdot S \cdot D \cdot D \cdot \epsilon$	$\emptyset$
	`invokestatic` `int Power.power(int,int)`	S	$S \cdot D \cdot D \cdot \epsilon$	$\emptyset$
	`imul`	D	$D \cdot D \cdot \epsilon$	$\emptyset$
	`goto L0`	S	$D \cdot \epsilon$	$\emptyset$
`L0` :	`ireturn`	S	$D \cdot \epsilon$	$\emptyset$

Fig. 3. BTA result of `power`.

3.4 Specializer Construction

From an original program and a result of BTA, a specializer is constructed in "pure" JVML. It generates specialized code on a per-instruction basis at runtime[18]. For each dynamic instruction in the original program, the specializer has a sequence of instructions that writes the bytecode of the instruction into an array. The specializer also performs method inlining by successively running specializers of a method caller and callee, and by inserting a sequence of instructions that saves and restores local variables appropriately.

Here, we describe the construction of a specializer by using pseudo-instructions. Note that those pseudo-instructions are used only for explanation, and they are replaced with sequences of pure JVML instructions in the actual specializer. The specializer is executable as a Java method.

The extended JVML for defining specializers contains the JVML instructions and pseudo-instructions, namely, `GEN` *instruction*, `LIFT`, `LABEL` L, `SAVE` n $[x_0, \ldots]$, `RESTORE`, and `INVOKEGEN` m $[x_0, \ldots]$, where *instruction* is a standard JVML instruction. Figure 4 shows an example definition of specializer `power_gen` with pseudo-instructions, constructed from method `power`. 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 an array where specialized code is stored.
- When an instruction has a different binding-time 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 $T[pc+1] = D \cdot \sigma$, where σ denotes an arbitrary stack type, `LIFT`

```
Method Power.power_gen(int)
    iload_1                          iload_1
    ifne L2                          iconst_1
 L1:iconst_1                         isub
    LIFT                             INVOKEGEN Power.power_gen(int) []
    goto L0                          GEN imul
 L2:GEN iload_0                      goto L0
    GEN iload_0                   L0:return
```

Fig. 4. Specializer definition with pseudo-instructions.

is inserted *after* the instruction. The `iconst_1` at `L1` in Figure 3 is an example. (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.

- Static `invokestatic` t_0 $m(t_1, \ldots, t_n)$ is translated into pseudo-instruction `INVOKEGEN` m`_gen`$(t_{j_1}, \ldots, t_{j_k})$ $[x_0, x_1, \ldots]$, where $t_{j_1}, \ldots, t_{j_k}$ are the types of static arguments, and $x_0, x_1, \ldots$ are the dynamic local variables at the current address. When `INVOKEGEN` is executed, (1) instructions that save local variables $x_0, x_1, \ldots$ to the stack and move values on top of the stack to the local variables are generated, (2) a specializer m`_gen` is invoked, and (3) instructions that restore saved local variables $x_0, x_1, \ldots$ 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 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 sequences of both branches, one of which is selected by the dynamic condition[4]. First, the jump instruction is translated into two pseudo-instructions: `GEN ifne` L and `SAVE` n $[x_0, x_1, \ldots]$, where n and $[x_0, x_1, \ldots]$ 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 $x_0, x_1, \ldots$ are saved. The execution of `RESTORE` resets the saved values on the stack and in the frame.

[4] Since JVML is an unstructured language, construction of a generating extension whose control flow visits all the nodes in both branches is not trivial. The algorithm for constructing such a generating extension will be explained in the other literature.

```
Method int power_2(int)
 0 iload_0
 1 iload_0
 2 istore_0
 3 iload_0
 4 iload_0
 5 istore_0
 6 iconst_1
 7 imul
 8 imul
 9 ireturn
```

Fig. 5. Specialized version of `power` with respect to 2.

3.5 Specializer Execution

The specializer definition is further translated into a Java method so that it takes (1) several parameters needed for specialization including an array `byte[] code` in which instructions of the specialized program are written and (2) the static arguments of the original method.

When a program uses the specializer, the following operations are performed: **(CP creation)** A '*Constant Pool*' (CP) object that records lifted values during specialization is created. **(specializer execution)** The specializer method is invoked with static arguments and the other necessary information for specialization. **(class finalization)** From the specialized instructions written in a `byte` array and the CP object, a ClassFile image[5] is created. **(class loader creation)** A ClassLoader object is created[6]. **(class loading)** Using the ClassLoader object, the ClassFile image is loaded into the JVM, which defines a new class with the specialized method. **(Instance creation)** An instance of the newly defined class is created. The program finally can call the specialized method via a virtual method of the instance.

Figure 5 shows the instructions for specialized `power` with 2 as a static argument. Some instructions, such as those that load a value immediately after storing the value, are unnecessary. Those instructions arise to save/restore local variables around inlined methods.

4 Performance Measurement

4.1 An Application Program: Mandelbrot Sets Drawer

As a target of specialization, we took a non-trivial application program that interactively displays the Mandelbrot sets. The user of the program can enter the definition of a function, and the program displays the image of the Mandelbrot set that is defined by using the function. Since the function is given interactively,

[5] Despite its name, a ClassFile image in our system is created as a `byte` array. No files are explicitly created for class loading.

[6] Since some JVM implementations significantly slowed down when a ClassLoader object loads a number of classes in our experiment, we create a class loader for each specialized code. Section 4.3 shows that the time for creating of a ClassLoader object is insignificant among the overall specialization overheads.

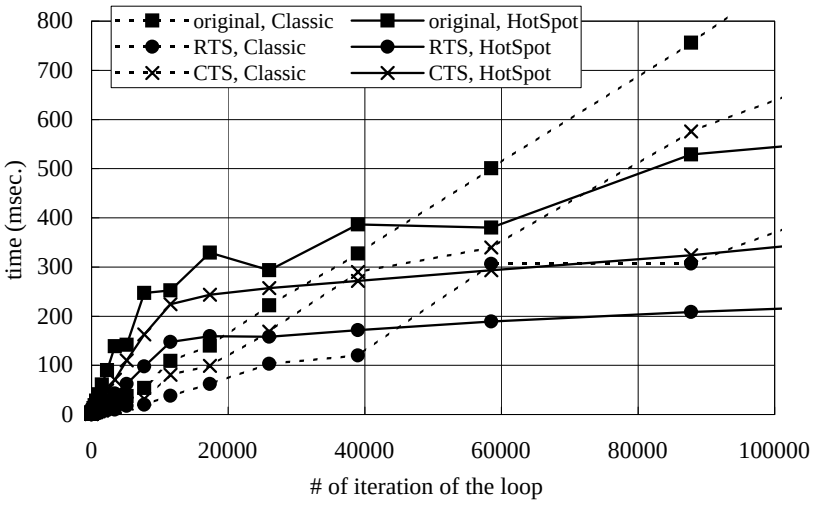

Fig. 6. Execution times of loops of specialized and unspecialized `eval`.

the program defines an interpreter for evaluating mathematical expressions. In order to draw an image of the set, the application have to evaluate the function more than one million times. This means that run-time specialization of the interpreter with respect to a given expression could improve the performance of the drawing process.

In our performance measurements, the method `eval` and its auxiliary methods in the interpreter, which take an expression and a store, and returns the value of the expression, are specialized with respect to an expression "`z*z+c`". Since current BCS implementation does not support objects, we modified the method to use arrays for representing expressions and stores.

We measured execution times of the target methods on two JVMs with different JIT compilers, namely, Sun "Classic" VM for JDK 1.2.1 with sunwjit compiler, and Sun "HotSpot" VM for JDK 1.2.2, in order to examine impacts of a JIT strategy on the specialization performance. All programs are executed on Sun Enterprise 4000 with 14 UltraSPARCs at 167MHz, 1.2GB memory, and SunOS 5.6. Execution times are measured by inserting `gethrvtime` system calls, which is called via a native method.

4.2 Performance of Specialized Method

We measured performance of three versions of the `eval` method on the abovementioned JVMs. The first one is the 'original' unspecialized method. The second one is a run-time specialized ('RTS') method generated by the BCS system. The third one is a compile-time specialized ('CTS') method, which is obtained by applying Tempo[22] after translating the original method into a C function.

Table 1. Execution times and relative speeds of `eval` method.

	execution times (μsec.)						relative speed		
	original		RTS		CTS				
VM	B_O	J_O	B_R	J_R	B_C	J_C	B_O/B_R	B_O/B_C	B_R/B_C
Classic	6.405	2,513	2.255	1,330	2.257	1,792	2.841	2.838	0.999
HotSpot	2.774	245,156	0.691	146,795	0.659	159,688	4.014	4.212	1.049

Table 2. Breakdown of specialization overheads.

VM	Classic		HotSpot	
process	time(μsec.)	(ratio)	time(μsec.)	(ratio)
CP creation	46.38	(1.7%)	95.91	(3.1%)
specializer execution	61.67	(2.3%)	194.81	(6.2%)
class finalization	55.77	(2.1%)	125.18	(4.0%)
class loader creation	16.68	(0.6%)	22.14	(0.7%)
class loading	1,907.33	(71.8%)	1,518.18	(48.5%)
instance creation	569.73	(21.4%)	1,172.96	(37.5%)
total (S)	2,657.57	(100.0%)	3,129.19	(100.0%)

Figure 6 shows the execution times of the method, which are measured by the following way. A ClassLoader object in our benchmark program first loads a new class that contains the (either specialized or unspecialized) `eval` method. The program then measures execution time of a loop that repeatedly invokes the `eval` method. Note that the measured time does *not* include specialization process, but *does* include the time of JIT compilation processes because JVMs perform JIT compilation during method invocations. As a result, the curves of the graph are not linear for small iteration numbers.

We therefore estimated, for each curve, execution times of the JIT-compiled body of the method (hereafter referred to as B) and JIT compilation process (J), by using an linear approximation of the curve at large iteration numbers.

Table 1 shows the estimated execution times and relative speed of the body of the method. As we see in the J_O, J_R and J_C columns, JIT compilation processes took from one millisecond to a few hundred milliseconds, depending on the JIT compilers. As we see in the B_O/B_R and B_R/B_C columns, the run-time specialized code runs 3–4 times faster than the unspecialized one does, and achieves almost the same speedup factors as the compile-time specialized code does.

4.3 Specialization Overheads and Break-even Points

Elapsed times for the specialization processes (S) are measured by averaging 10,000 runs. Table 2 shows the time for each sub-process, which is explained in Section 3.5. As we see, 80–90% time of the specialization process is spent for the

Table 3. Break-even points.

VM	over JIT compiled code	over newly loaded code
Classic	961	355
HotSpot	71,975	(less than zero)

ones inside the JVM, namely, class loading and instance creation. We presume that some of overheads could be removed if we integrated our system with a JIT compiler so that the specializer directly generates specialized code in an intermediate representation in the JIT compiler.

A *break-even point* (BEP) is a number of runs of a specialized program needed to amortize the specialization cost over the execution time of the unspecialized program. In programming systems that perform dynamic optimizations, even unspecialized programs have to pay overheads of the optimization, namely JIT compilation time. We therefore calculated two BEPs. The first one assumes that the unspecialized code is already JIT compiled. In this case, a BEP, which is calculated by the formula $(J_R + S)/(B_O - B_R)$, is approximately 1,000–72,000 runs as shown in Table 3. The second one assumes that the unspecialized code is newly loaded, and thus pays the cost of JIT compilation during its execution. The BCS specialized code exhibits a small BEP in this case, which is computed by the formula $(J_R + S - J_O)/(B_O - B_R)$. Note that the benchmark application, in order to draw an image of a given expression, executes the `eval` method for much larger number of times than the BEPs. This means that BCS actually improves the overall execution times of the application.

4.4 Comparison to a Native Code Run-time Specialization System

In order to compare the speedup factors and specialization overheads with a run-time native-code specialization system, we also wrote the same interpreter in C, and specialized by using Tempo 1.194[7]. We have tested two binding-time configurations for specializing the interpreter. The one is to specialize the function with respect to three out of five arguments (shown in the '3/5' row in Table 4), which is the same configuration to the experiment in BCS. The other is to specialize with respect to two out of five arguments (the '2/5' row), in which an array containing a return value index is set to be dynamic. The interpreter is compiled by GCC 2.7.2 with `-O2` option. All the other execution environments are the same to the previous ones.

Table 4 shows the execution times and specialization times that are measured by averaging ten million runs. We observe that the run-time specialized code is slower than the compile-time specialized one in Tempo. Surprisingly, the run-time specialized code that is specialized under the same configuration to the

[7] We set both `reentrant` and `post_inlining` options of Tempo to `true`, and the compiler options for both templates and specializers to `"-O2"`. We also implemented an efficient memory allocator for residual code.

Table 4. Execution and specialization times and break-even points of `eval` in Tempo.

# of static	execution times (μsec.)				relative speed			BEP
	none	RTS		CTS				
args.	B_O	B_R	S	B_C	B_O/B_R	B_O/B_C	B_R/B_C	
3/5	1.278	63.591	85.712	0.298	0.02	4.29	213.4	∞
2/5	1.278	0.789	22.881	0.363	1.62	3.52	2.174	46.8

experiments in the previous subsections is even slower than the original code. We presume that this anomaly is caused by a number of array accesses whose indices are 'lifted' at specialization time. When we made the array to be dynamic (the '2/5' row), the run-time specialized function become faster than the original one, and its break-even point is smaller than the ones in BCS.

Comparing between the execution time in BCS and the one in Tempo, we notice that compile-time specialized codes in those two systems show the similar speedup factors (B_O/B_C). On the other hand, the speedup factors of the run-time specialized code (B_O/B_R) in Tempo are worse than the one in BCS. This can be an evidence of our premise: performing optimizations after specialization could be useful to improve performance of run-time specialized code.

5 Related Work

Tempo is a compile-time and run-time specialization system for C language[22]. Tempo achieves portability by using outputs of standard C compilers to construct specializers. As the specializers simply copy templates to memory at run-time, their BEP numbers are low (3 to 87 runs in their realistic examples). On the other hand, the specializers perform no optimizations and no function inlining at run-time specialization.

DyC is another RTS system for C language[12]. The analysis and specializers can directly handle unstructured C programs. The system generates highly optimized code, by developing its own optimizing compiler for Digital Alpha 21164. It can perform optimizations at run-time specialization[2]. However, the optimizations seem to make the BEP numbers larger (around 700 to 30,000), similar to BCS.

Fabius is an RTS for pure-functional subset of ML, targeting MIPS R3000[18]. Because the source language is a pure functional language, the BTA and specializer construction in Fabius are simpler than those for imperative and unstructured languages. Similar to BCS, specializers in Fabius are on a per-instruction basis and perform function inlining for tail recursive functions. It is also suggested that the specializers would perform register allocation at run-time.

Fujinami proposes a run-time specialization system for C++, targeting MIPS R4000 and Intel x86[10]. The system is designed to perform implicit optimizations; *i.e.,* it specializes a given program with respect to its invariants, which are determined by an automatic analysis. A specialized program runs runs twice as

fast as the one compiled by a statically optimizing compiler. His system achieves this speedup by embedding a number of optimization algorithms into a statically generated specializer. Our approach, on the other hand, is to optimize a specialized code by using a JIT compiler, which is an independent module.

`C is a language with dynamic code generation mechanisms[24]. Unlike other RTS systems, `C programmers have to explicitly specify binding-times of expressions. Similar to BCS, the implementation of `C generates programs in virtual machine languages called VCODE and ICODE. The run-time system of ICODE performs optimizations including register allocation for generated programs, similar to JIT compilers for JVMs.

Bertelsen proposes, independently of BCS, an algorithm for binding-time analysis of a JVML subset, which does not include method invocations nor objects[5]. A specialization process based on the analysis is informally discussed, which is not yet implemented to the authors' knowledge.

JSpec is an off-line, compile-time partial evaluator for Java[26]. The system analyzes and specializes Java programs by applying Tempo, a partial evaluator for C, after translating the Java programs into C. This approach can be compared to ours that uses a compiler from a high-level language to a bytecode language as a front-end. Unlike current BCS implementation, JSpec supports objects whose specialization strategies are specified through *specialization classes*[28].

6 Conclusion

In this paper, we have proposed run-time bytecode specialization (BCS), which specializes Java virtual machine language (JVML) programs at run-time. The characteristics of this approach are summarized as follows: (1) the system directly analyzes a program and creates a specializer in an intermediate language JVML; and (2) the specializer generates programs in JVML, which makes it possible to apply optimizations after specialization by using existing JVMs with just-in-time (JIT) compilers.

The binding-time analysis algorithm is based on a type system, and also uses results of flow analysis to correctly handle stacks, local variables, and side-effects.

Thus far, we have implemented a prototype BCS system for a JVML subset and have shown that a non-trivial program specialized by our system runs approximately 3–4 times faster than the unspecialized program. The specialization cost can be amortized by 1,000 to 72,000 runs, depending on the JVMs. Those numbers are worse than the ones in the systems that are rather focusing on the specialization speed[18, 22], though.

We are now extending our system to support the full JVML. Since current implementation only supports primitive types and arrays, rules that properly handle references to objects should be devised. To support objects and arrays, the system needs information whether data is modified by other methods or other threads. Such information could be obtained by either static analysis (*e.g.,* the one studied by Choi, et al.[7]) or through user declarations[28]. In practice, it

is also important to support other features, such as multi-threading, and subroutines (*i.e.,* `jsr` and `ret` instructions in JVML) and exceptions. Some may consider that templates of bytecode would reduce specialization costs. As our experiments in Section 4 showed, however, the major sources of specialization overheads are class loading and JIT-compilation. Rather than improving the performance of the bytecode generation process, our current plan is to generate a specialized program directly in an intermediate language of a JIT compiler, by using JVMs with interfaces to JIT compilers[23].

Acknowledgments

We would like to thank Reynald Affeldt, Naoya Maruyama, and Satoshi Matsuoka for discussing on the BCS system. We also would like to thank the anonymous reviewers for helpful comments.

References

1. K. Asai. Binding-time analysis for both static and dynamic expressions. In *SAS'99* (LNCS 1694), 1999.
2. J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *PLDI'96*, pp.149–159, 1996.
3. N. Benton, A. Kennedy, and G. Russell. Compiling standard ML to Java bytecodes. In *ICFP'98*, 1998.
4. A. A. Berlin and R. J. Surati. Partial evaluation for scientific computing: The supercomputer toolkit experience. In *PEPM'94*, pp.133–141, 1994.
5. P. Bertelsen. Binding-time analysis for a JVM core language. Apr. 1999. `http://www.dina.kvl.dk/~pmb/`.
6. P. Bothner. Kawa – compilinig dynamic languages to the Java VM. In *USENIX*, 1998.
7. J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In L. M. Northrop, editor, *OOPSLA'99*, pp.1–19, 1999.
8. C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *POPL'96*, pp.145–170, 1996.
9. D. R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *PLDI'96*, pp.160–170, 1996.
10. N. Fujinami. Determination of dynamic method dispatches using run-time code generation. In *TIC'98*, pp.253–271, 1998.
11. Y. Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprinted from *Systems, Computers, Controls*, 2(5):45–50, 1971.
12. B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. An evaluation of staged run-time optimization in DyC. In *PLDI'99*, 1999.
13. B. Guenter, T. B. Knoblock, and E. Ruf. Specializing shaders. In *SIGGRAPH'95*, pp.343–349, 1995.
14. F. Henglein. Efficient type inference for higher-order binding-time analysis. In *FPCA'91*, pp.448–472, 1991.
15. L. Hornof and T. Jim. Certifying compilation and run-time code generation. In *PEPM'99*, pp.60–74, 1999.

16. L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In *PEPM'97*, pp. 63–73, 1997.
17. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
18. P. Lee and M. Leone. Optimizing ML with run-time code generation. In *PLDI'96*, pp.137–148, 1996.
19. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.
20. H. Masuhara and A. Yonezawa. Design and partial evaluation of meta-objects for a concurrent reflective language. In *ECOOP'98*, pp.418–439, 1998.
21. H. Masuhara and A. Yonezawa. Generating optimized residual code in run-time specialization. In *International Colloquium on Partial Evaluation and Program Transformation*, pp.83–102, 1999.
22. F. Noël, L. Hornof, C. Consel, and J. L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *ICCL'98*, pp.123–142, 1998.
23. H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, Y. Kimura, OpenJIT: an open-ended, reflective JIT compiler framework for Java, In *ECOOP 2000*, pp.362-387, 2000.
24. M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: a language and compiler for dynamic code generation. *TOPLAS*, 21(2):324–369, 1999.
25. C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: streamlining a commercial operating system. In *SOSP'95*, pp.314–324, 1995.
26. U. P. Schultz, J. L. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In *ECOOP99*, pp.367–390, 1999.
27. R. Stata and M. Abadi. A type system for Java bytecode subroutines. *TOPLAS*, 21(1):90–137, 1999.
28. E. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In *OOPSLA'97*, pp.286–300, 1997.
29. P. Wickline, P. Lee, and F. Pfenning. Run-time code generation and Modal-ML. In *PLDI'98*, pp.224–235, 1998.