Vol. 46 No. SIG 1(PRO 24) IPSJ Transactions on Programming Jan. 2005

Regular Paper

Logic-based Binding Time Analysis for Java Using Reaching Definitions

Susumu Yamazaki,[†1,†2] Takayuki Kando,[†3]
Michihiro Matsumoto,[†1,†2] Tsuneo Nakanishi,[†2,†4,†5]
Teruaki Kitasuka[†2] and Akira Fukuda[†2,†5]

A recent trend in program development is the employment of generic software components such as libraries and frameworks. Typically, however, it is difficult to achieve both genericity and runtime execution efficiency simultaneously. Therefore, many researchers have studied program specialization, which is one technique to translate a generic program automatically into an efficient program specialized for a specific runtime environment. However, it is very difficult to implement a system that can specialize practical applications. Although some possible reasons exist for the problem, this paper focuses on the problems of instruction-dependent processes. Each necessary analysis for existing program specializer systems must include instruction-dependent processes. Therefore, not only does the code size of the specializer get larger, but maintainability and reliability are also degraded. Then, we propose a new algorithm of logic-based binding time analysis using reaching definitions analysis, which is widely used among many other analyses. We also evaluate how this technique improves the implementation simplicity of binding time analyzer: the code size is reduced by about 10%. Especially, instruction-dependent processes are almost entirely eliminated.

1. Introduction

Recently, development of applications with more features is desired within a shorter time span. Therefore, it is necessary to employ generic software components such as libraries, frameworks, components, etc. Currently, we are working to develop a methodology for embedded systems in which these problems are rapidly getting more important.

Realization of genericity requires code for adaptation for various conditions and environments such as calculations of parameters and adaptation logic. Runtime execution efficiency is degraded because generic components include the code for their adaptation. This problem cannot be ignored, especially in the field of embedded systems, where a program is often modified by hand to improve execution efficiency while sacrificing genericity.

Conditions and environments are classified into two categories: one is fixed *statically* before program execution, the other is changed *dynamically* at runtime. Consequently, the program calculations are also divided into two parts: one can be computed in advance from the static conditions and environments, the other must be computed dynamically at runtime. Classification according to when the conditions, the environments, and the result of calculations are fixed is called *binding times*.

A real application often includes static conditions and environments. Similarly, a part in the program, especially the code for adaptation, may be frequently static because it is derived from static conditions and environments.

Therefore, it improves runtime execution efficiency to compute static calculations in advance and to execute only essentially dynamic calculations. We can regard the above-mentioned program modification by hand in the field of embedded systems as a kind of speculative calculation.

If the modification can be realized to generate a program automatically and advance computed static calculations, it can reduce the cost of the modification by hand; moreover, it can mitigate risks of modification mistakes.

One technique that realizes this is *program specialization* [7]. It is an optimization technique that translates automatically from a generic program and some parameters to an efficient program whose parts derived from the parameters are computed in advance.

Unfortunately however, it is difficult to develop a practical program specialization system

†1 Fukuoka Laboratory for Emerging & Enabling Technology of SoC, Fukuoka Industry, Science & Technology Foundation
†2 Graduate School of Information Science and Electrical Engineering, Kyushu University
†3 freelance
†4 System LSI Research Center, Kyushu University
†5 Computing and Communications Center, Kyushu University

that supports imperative languages such as C, and object-oriented languages such as C^{++} and Java. The reasons are as follows:

- **Bloated System Problem**: To describe analyzers and a code translator for program specialization, the system must provide processes that correspond to all instructions. As a result, the system tends to bloat.
- **Maintainability Problem**: As the result of bloated system problem, it is very expensive to ensure that the system performs correctly.
- **Architectural Problem**: Little care has been given to simplify the program specialization system or to compose it using general analyzers and optimizers because the target of existing works is not real applications with practical languages, but languages that do not have many primitives or toy problems.

In this paper, we propose the **Reaching Definitions Method (RDM)**, which is a new algorithm based on logic-based binding time analysis 10). It simplifies the program specialization system using reaching definitions analysis.

The above problems are solved by the RDM:

- **Bloated System Problem**: The RDM abstracts dependence between instructions by reaching definitions. Consequently, almost all the **instruction-dependent code** is eliminated.
- **Maintainability Problem**: The control/data flow of RDM is so simpler than the legacy analysis that the cost of ensuring correctness is less.
- **Architectural Problem**: The RDM is composed of basic program analyses such as reaching definitions analysis. Consequently, we can apply RDM to practical languages.

The rest of this paper is organized as follows. Section 2 describes the background of this paper. Section 3 describes the Direct Method (DM), which is the naive implementation of logic-based binding time analysis for Java virtual machine language (JVML). Section 4 proposes the RDM. Section 5 describes these implementation and assesses and compare the DM, the RDM and constraint-based binding time analysis, which are used widely. Finally, Section 6 summarizes this paper and discusses a future direction.

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

Fig. 1 The power program in Java.

2. Background

In this section, we explain program specialization, logic-based binding time analysis, and bytecode specialization.

2.1 Program Specialization and Binding Time Analysis

Assume that there are a program f and two kinds of its input s and d, and that $f(s,d)$ represents the program execution. Program specialization (or partial evaluation) is an algorithm that is given f and s; it generates f_s defined by $\forall d.f_s(d) = f(s,d)$.

Now, inputs can be divided into two categories by the time these become available (binding times):

- **static**: these are available in the specialization time,
- **dynamic**: these are not available until runtime.

Binding times of each part of a program are derived from binding times of its inputs. Static parts of the program can be evaluated in the specialization process.

Two kinds of program specialization exist: online and offline. Online specialization performs *binding time analysis (BTA)* and code generation concurrently. Offline specialization separates the binding time analysis phase and the code generation phase. In this paper, we adopt offline specialization.

BTA is analysis which derives binding times of every part (statements, expressions, variables, etc.) of a program from binding times of its inputs. For example, the result of BTA for the program in **Fig. 1** is shown by **Fig. 2**: where the underlined parts are dynamic, the others are static.

2.2 Makholm's Logic-based Binding Time Analysis

Several ways to analyze binding times ex-

Table 1 Comparison of constraint-based BTA and logic-based BTA.

	constraint-based BTA	logic-based BTA
basis	type inference	dependence
constraint class	various	single
BTA algorithm	constraints construction normalization minimal solution	graph construction graph search

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

Fig. 2 The annotated power program.

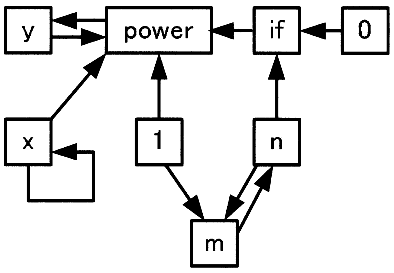

Fig. 3 Constraints as a graph.

ist, but this paper specifically addresses logic-based BTA[10]. **Table 1** shows a comparison of it and constraint-based BTA, which is widely used. Constraint-based BTA regards the binding time of a term as a type; the inferences of types under the constraints derive from typing rules corresponding to grammar. On the other hand, logic-based BTA uses only dependence relationships based on data-flow.

We explain the details of logic-based BTA using an example. We can naturally derive the constraints from dependence relationships of each line in Fig. 1 as follows:

(1) If n or 0 is dynamic, then so must if be.
(2) If 1 is dynamic, then so must the return value of power be.
(3) If n or 1 is dynamic, then so must m be.
(4) If x of the caller is dynamic, then so must x of the callee be.
(5) If m of the caller is dynamic, then so must n of the callee be.
(6) If the return value of power is dynamic, then so must y be.
(7) If x or y is dynamic, then so must the return value of power be.
(8) If if is dynamic, then so must the return value of power be☆.

These constraints can be represent by the form "If α is dynamic, then so must β be." or "If α_1 or $\alpha_2 \cdots$ is dynamic, then so must β be." Therefore we can consider the constraints as the graph in **Fig. 3**, where a logical expression such as "α is dynamic" is a vertex, where "if ... then" is a directed edge, and where "or" is a merge of edges.

This graph represents dependence relationships between terms such as variables and expressions. All BTA must do is find all terms that must be dynamic. If a term t is dynamic, all terms depending on t must be dynamic. To use the graph, the vertices that can be reachable from the vertex corresponding to t in the graph correspond to the terms. Therefore, BTA can be considered as a graph search problem.

Consider analyzing binding times of the program in Fig. 1 using the graph in Fig. 3. Presume that x is dynamic. To traverse the graph from the vertex of x, we can traverse the vertices of y and the return value of power, and can not do the others. This means that y and the return value are dynamic; the others are static. This result equals the result of Fig. 2.

2.3 Bytecode Specialization

The above explanation assumes that BTA and specialization are source-to-source program analysis and translation, respectively. In contrast, bytecode specialization[11),12)] uses Java Virtual Machine Language (JVML)[9)].

The primary feature of bytecode specialization is the ability to realize portable and efficient runtime specialization. Runtime specialization realizes specialization using values that are computed at runtime. Bytecode specialization has high portability because the byte-

☆ This constraint is derived from *non-local side-effects under dynamic control*[7)]. The following discussion of the proposed methods abbreviates it, but it can be introduced easily.

```
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 1                                   // (case n ≠ 0) push n
 5 iconst 1                                  // push 1
 6 isub                                      // compute n − 1
 7 istore 2                                  // pop m
 8 iload 0                                   // push x as arg. #0
 9 iload 2                                   // push m as arg. #1
10 invokestatic int Power.power(int,int)     // call method
11 istore 2                                  // pop y
12 iload 0                                   // push x
13 iload 2                                   // push y
14 imul                                      // compute x × y
15 ireturn                                   // return x × y
```

Fig. 4 The power program in JVML.

Table 2 Typical bytecode instructions.

`iload n`	push the current value of the `n`-th local variable onto the stack
`istore n`	pop a value off the stack and assign it to the `n`-th local variable
`iconst n`	push a constant `n` onto the stack
`iadd`	pop two values off the stack and push the sum of them onto the stack
`isub`	pop two values off the stack and push the difference of them onto the stack
`imul`	pop two values off the stack and push the multiple of them onto the stack
`ifne l`	pop a value off the stack and jump to the address `l` in the current method if the value is not zero
`invokestatic m`	(1) pop n values off the stack (2) save the current frame and program counter (3) assigns the poped value into the variables $0, ..., (n-1)$ in a newly allocated frame (4) jump to the first address of method `m`
`ireturn`	(1) pop a value off the stack (2) dispose of the current frame and restore the saved one (3) push the value on the restored stack (4) jump to the next address of the saved program counter

code specialization target is JVML, which is widely used among various platforms. It also has higher execution efficiency than other runtime specialization systems because its state-of-the-art technology drastically improves JIT, which is used by a bytecode specializer. The program specializer and JIT complement each other at the point of efficiency effect: the former is useful for larger scale optimization using application nature; the latter is useful for instruction level optimization based on runtime profile information.

JVML is a typed virtual machine language based on the stack machine model. Our current implementation has limitations: it does not support exception and multi-threading.

Figure 4 shows the result of compilation of the power program in Fig. 1. 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[11]. **Table 2** shows several bytecode instructions.

Figure 5 shows the binding time annotated `power` program in JVML when the binding times of `x` and `n` are dynamic and static, respectively. The binding times of instructions which are displayed in the code column are either S (static) or D (dynamic). Binding times of the operand stack, which is displayed in the stack column, is written as the form $\tau_1 \cdot \tau_2 \cdots \tau_n \cdot \epsilon$, where τ is a binding time of the corresponding stack entry. The binding time of local variables, which are displayed in the local variables column, are denoted as $\emptyset$ or $[i_k \mapsto \tau_k]$, where the binding time of the variable corresponding to i_k is τ_k. Only live local variables are displayed.

Instruction	Code	Stack	Local Variables
0 `iload 1`	S	ϵ	$[0 \mapsto D, 1 \mapsto S]$
1 `ifne 4`	S	$S \cdot \epsilon$	$[0 \mapsto D, 1 \mapsto S]$
2 `iconst 1`	D	ϵ	$\emptyset$
3 `ireturn`	D	$D \cdot \epsilon$	$\emptyset$
4 `iload 1`	S	ϵ	$[0 \mapsto D, 1 \mapsto S]$
5 `iconst 1`	S	$S \cdot \epsilon$	$[0 \mapsto D]$
6 `isub`	S	$S \cdot S \cdot \epsilon$	$[0 \mapsto D]$
7 `istore 2`	S	$S \cdot \epsilon$	$[0 \mapsto D]$
8 `iload 0`	D	ϵ	$[0 \mapsto D, 2 \mapsto S]$
9 `iload 2`	S	$D \cdot \epsilon$	$[0 \mapsto D, 2 \mapsto S]$
10 `invokestatic int Power.power(int,int)`	D	$S \cdot D \cdot \epsilon$	$[0 \mapsto D]$
11 `istore 2`	D	$D \cdot \epsilon$	$[0 \mapsto D]$
12 `iload 0`	D	ϵ	$[0 \mapsto D, 2 \mapsto D]$
13 `iload 2`	D	$D \cdot \epsilon$	$[2 \mapsto D]$
14 `imul`	D	$D \cdot D \cdot \epsilon$	$\emptyset$
15 `ireturn`	D	$D \cdot \epsilon$	$\emptyset$

Fig. 5 Binding-time annotated `power`.

Although there is no object in the `power` example, according to Ref. 1), escape analysis [2),4),15)] and side-effect analysis [5)] are necessary to support objects. Escape analysis and side-effect analysis are required to decide which part to perform dynamically creation and side-effecting operation of objects, respectively. We explain and discuss it in detail in the Section 4.5.

3. Direct Method: Logic-based Binding Time Analysis for Java

The Direct Method (DM) is the naive implementation of logic-based binding time analysis for JVML.

We describe the assumptions of BTA before entering discussion. We assume that BTA should be flow sensitive (or pointwise) and context insensitive (or monovariant).

In this case, "flow sensitive" means that the same stack entry or local variable can have different binding times at each instruction in a method. BTA should be flow sensitive because some different local variables are allocated to the same number of the local variable. For example, in Fig. 4, the local variable m and y are allocated to the 2nd variable on rows 7–9 and 11–13, respectively. Consequently, the binding times of m and y are static and dynamic, respectively.

On the other hand, "context insensitive" means that the same stack entry or local variable along a different control flow path cannot have different binding times. Our BTA should be context insensitive because the cost of context insensitive analysis is much less than that for context sensitive analysis (See Section 3.6 in Ref. 13)).

Now consider logic-based BTA on JVML. First, because we assume flow sensitive and context insensitive BTA, the analyzer creates the graph in which each instruction and corresponding operand stack entry and local variables are vertices. Secondly, the analyzer adds it and directed edges according to dependence relationships of each instruction; then the graph construction is completed. Finally, the analyzer traverses the graph from the dynamic input and produces instructions and variables corresponding to vertices that can be traversed as dynamic.

We next explain the notation before showing the rules of logic-based BTA: P represents the program, PC represents the program counter. B, F, T, A, R are the binding times of an instruction, local variables, the operand stack, the arguments and the return value, respectively. P, B, F, T are indexed by PC, R is indexed by a method m, and A is indexed by m and the number of argument i.

Presume that α and β are binding time variables. We write $\alpha \to \beta$ as "if α is dynamic, then so must β be." We also write $\alpha \leftrightarrow \beta$ as $\alpha \to \beta$ and $\beta \to \alpha$.

F is the map from local variables to binding time variables. Assume that $\alpha_{x,PC}$ is the binding time of a local variable x at PC. $F_{PC} \Rightarrow F_{PC+1}$ indicates that for all local variable x in F, $\alpha_{x,PC} \to \alpha_{x,PC+1}$. Moreover, $F_{PC}[y \mapsto \beta] \Rightarrow F_{PC+1}$ means that for all local

$$\frac{\begin{array}{c}P_{PC} = \texttt{iconst}\ n\\ F_{PC} \Rightarrow F_{PC+1}\\ T_{PC} = \sigma\\ T_{PC+1} = \alpha \cdot \sigma\\ B_{PC} \leftrightarrow \alpha\end{array}}{A, R, B, F, T, PC \vdash P}
\quad
\frac{\begin{array}{c}P_{PC} = \texttt{iadd}\\ F_{PC} \Rightarrow F_{PC+1}\\ T_{PC} = \alpha \cdot \beta \cdot \sigma\\ T_{PC+1} = \gamma \cdot \sigma\\ \alpha \rightarrow B_{PC}\\ \beta \rightarrow B_{PC}\\ B_{PC} \leftrightarrow \gamma\end{array}}{A, R, B, F, T, PC \vdash P}
\quad
\frac{\begin{array}{c}P_{PC} = \texttt{iload}\ x\\ F_{PC} \Rightarrow F_{PC+1}\\ T_{PC} = \sigma\\ T_{PC+1} = \alpha \cdot \sigma\\ F_{PC}[x] \rightarrow B_{PC}\\ B_{PC} \leftrightarrow \alpha\end{array}}{A, R, B, F, T, PC \vdash P}
\quad
\frac{\begin{array}{c}P_{PC} = \texttt{istore}\ x\\ F_{PC}[x \mapsto \alpha] \Rightarrow F_{PC+1}\\ T_{PC} = \beta \cdot \sigma\\ T_{PC+1} = \sigma\\ \beta \rightarrow B_{PC}\\ B_{PC} \rightarrow \alpha\end{array}}{A, R, B, F, T, PC \vdash P}$$

$$\frac{\begin{array}{c}P_{PC} = \texttt{ifne}\ L\\ F_{PC} \Rightarrow F_{PC+1}\\ F_{PC} \Rightarrow F_L\\ T_{PC} = \alpha \cdot \sigma\\ T_{PC+1} = \sigma\\ T_L = \sigma\\ \alpha \rightarrow B_{PC}\end{array}}{A, R, B, F, T, PC \vdash P}
\quad
\frac{\begin{array}{c}P_{PC} = \texttt{invokestatic}\ m\\ F_{PC} \Rightarrow F_{PC+1}\\ n = \mathit{arity}(m)\\ T_{PC} = \alpha_{n-1} \cdots \alpha_0 \cdot \sigma\\ 0 \leq i < n.\ \alpha_i \leftrightarrow A_{m,i}\\ T_{PC+1} = \beta \cdot \sigma\\ R_m \rightarrow B_{PC}\\ B_{PC} \rightarrow \beta\end{array}}{A, R, B, F, T, PC \vdash P}
\quad
\frac{\begin{array}{c}P_{PC} = \texttt{ireturn}\\ T_{PC} = \alpha \cdot \sigma\\ \alpha \rightarrow B_{PC}\\ B_{PC} \leftrightarrow R_m\end{array}}{A, R, B, F, T, PC \vdash P}$$

Fig. 6 Rules of logic-based BTA.

variables x in F except y, $\alpha_{x,PC} \rightarrow \alpha_{x,PC+1}$, and that $\beta \rightarrow \alpha_{y,PC+1}$.

Figure 6 shows part of rules of logic-based BTA; we explain its details as follows:

- An instruction that pushes the result onto the stack of the next address (such as `iconst`, `iadd` and `iload`) creates a bi-directional edge between the instruction and the result. The reason why the edge is bi-directional is that it is necessary to make the binding time of the stack entry be dynamic if there is flow merging and if one stack entry is dynamic.
- An instruction that pops information off the stack (such as `iadd`, `istore`, `ifne` and `ireturn`) creates an edge from the stack entry to the instruction.
- An instruction that gets the value from a local variable (`iload`) creates an edge from the local variable to the instruction.
- Almost all instructions create edges from local variables of the instruction to the ones of the next instruction. Only an instruction that assigns a value to a local variable (`istore`) creates such edges, except the local variable, and creates an edge from the instruction to the local variable of the next instruction.
- An instruction that switches the control flow (`ifne`) creates edges from local variables to those of the instructions that are potentially jumped to from the instruction (potentially transitional instructions)☆ and unifies the binding time of stack of the original instruction and the potentially transitional instructions.
- An instruction that calls a method (`invokestatic`) creates bi-directional edges between the arguments and the stack entries included in the arity of the method *arity*(m), edges from the return value of the method to the instruction and the stack top of the next instruction. The reason why the edges to the arguments are bi-directional is that the binding times of the method arguments are context insensitive.
- An instruction that returns from a method invocation (`ireturn`) creates a bi-directional edge between the instruction and the result value of the method. The reason why the edge is bi-directional is that return instructions in a method should have the same binding times.

How to support objects is according to Ref. 1). Our method does not support *partial static* binding times, where ones of object fields are static and the others are dynamic.

We show an example: **Fig. 7** shows the sample `inc` program, and **Fig. 8** shows the JVML code that is compiled from the `inc` program. **Figure 9** shows its graph.

4. Reaching Definitions Method: Improved Binding Time Analysis

This section presents a new BTA algorithm.

☆ Potentially transitional instructions do not include instructions that are jumped into because of an exception throwing.

```
class Increment {
  static int inc(int v) {
    int w = v + 1;
    return w;
  }
}
```

Fig. 7 The inc program.

```
Method int Increment.inc(int)
0 iload 0      // push v
1 iconst 1     // push 1
2 iadd         // compute v + 1
3 istore 1     // pop w
4 iload 1      // push w
5 ireturn      // return w
```

Fig. 8 The JVML representation of the inc program.

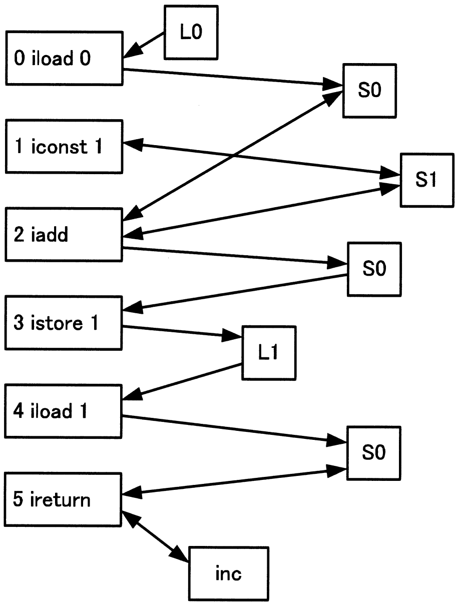

Fig. 9 Graph of the inc program.

4.1 Instruction-dependent Process Problems

Figure 10 shows a sample implementation of the DM in a Java like language. The body of the method `createGraph` is a loop that iterates the sequence of instructions `m`. The loop includes an extended `switch` statement that switches by the class. An *instruction-dependent process* is a process for each instruction class.

Because the DM includes many instruction-dependent processes, it has the following three problems at least:

- The source code size is large, despite the similarity between rules of each instruction. We can group similar code.

```
void createGraph(Graph g, Collection<Instruction> m) {
  int pc = 0;
  for (Instruction i : m) {
   switch(i.class) {
    case Iconst: // $P_{PC}$ = iconst $n$
     g.addEdgesOfLocalVariables(pc); // $F_{PC} \Rightarrow F_{PC+1}$
     g.addEdge(i, m[pc + 1].stack[0]); // $B_{PC} \rightarrow \alpha$
     g.addEdge(m[pc + 1].stack[0], i); // $B_{PC} \leftarrow \alpha$
     break;
    case Iadd: // $P_{PC}$ = iadd
     g.addEdgesOfLocalVariables(pc); // $F_{PC} \Rightarrow F_{PC+1}$
     g.addEdge(i.stack[0], i); // $\alpha \rightarrow B_{PC}$
     g.addEdge(i.stack[1], i); // $\beta \rightarrow B_{PC}$
     g.addEdge(i, m[pc + 1].stack[0]); // $B_{PC} \rightarrow \alpha$
     g.addEdge(m[pc + 1].stack[0], i); // $B_{PC} \leftarrow \alpha$
     break;
    ...
   }
   pc++;
  }
}
```

Fig. 10 Instruction-dependent processes in the implementation of the DM.

- Maintainability is bad because there are many test cases that we must perform, which is necessary to ensure that the binding time analyzer is correct.
- It is not reasonable to implement a logic-based binding time analyzer at such a high cost because the result and the intermediate products of logic-based BTA are not reusable in another analysis or optimization, to our knowledge, especially analyses used in specialization such as escape analysis.

We call these *the instruction-dependent process problems.*

In practical specialization systems, the problems are serious. Though we have implemented a specialization system based on the DM, the implementation and test code are so huge and unmanageable that we cannot implement the specializer supporting all instructions, completely.

4.2 Our Approach

We propose a derivation of the BTA result from the result of another general analysis to solve the instruction-dependent process problem.

Consider again the principles of logic-based BTA. The rules of logic-based BTA are derived from the dependence relationships between instructions and variables. They are calculable from the relationships between definitions and the references of the variables. Therefore, the BTA result can be derived from reaching definitions, which is reusable to type analysis, escape analysis, etc.

We explain this concretely as follows. First, the analyzer gets the definition of each stack entry and the local variable of each instruction using reaching definitions. Next, it gets information regarding which stack entry and local variable is used by each instruction. Our analyzer framework [16),17)] provides the information because it is basic to almost all analysis. Finally, the analyzer calculates dependence relationships between each instruction. Therefore, it can create a constraint graph.

For only method invocation, the analyzer must modify edges of the graph between the stack, the arguments, and the result of the method.

This approach solves instruction-dependent process problems. First, the calculation of dependence relationships and the constraint graph construction are independent from instructions. Next, though reaching definitions analysis and the information regarding which stack entry and local variable is used by each instruction depend on instructions, it is reasonable to implement them at a high cost because they are used in other analyses and optimizations such as escape analysis, which is used in object-oriented specialization.

4.3 Algorithm

Figure 11 shows the RDM algorithm.

The algorithm input is the target method `M` and the list of dynamic arguments `A`. The algorithm output is the set of dynamic instructions `BT`.

Variables that the algorithm uses are the graph `G`, a node (an instruction, a local variable or a stack entry) of the graph `N`, an instruction `I`, and a set of done methods or nodes `D`.

Basic procedures and functions of the algorithm are as follows:

- `Parameters(M)` returns the list of the parameter nodes of the method `M`.
- `ReturnValue(M)` returns the return value node of the method `M`.
- `Code(M)` returns the list of instructions in the method `M`.
- `RD(M, N)` returns the reaching definitions of the method `M` and the node `N`.
- `Variables(I)` returns the set of the input local variables and the stack entries that the instruction `I` depends on.
- `MethodSet(I)` returns the set of the methods that the instruction `I` calls potentially.
- `AddEdge(G, N, N')` puts an edge from the node `N` to the node `N'` into the graph `G`.
- `Edges(G, N)` returns all edges from the node `N` in the graph `G`.

```
BTA(M, A)
  D := nil;
  G := nil;
  BT := nil;
  CreateGraph(M, G, D);
  TraverseGraph(BT, G, A);

CreateGraph(M, G, D)
  if(M ∉ D)
    D := cons(M, D);
    for all I in Code(M)
      if(!(I instanceof invoke))
        for all N in Variables(I)
          for all N' in RD(M, N)
            AddEdge(G, N', I);
        if(I instanceof return)
          AddEdge(G, I, ReturnValue(M));
          AddEdge(G, ReturnValue(M), I);
      else  // I instanceof invoke
        for all M' in MethodSet(I)
          CreateGraph(M', G, D);
          for all N in Variables(I)
              and N' in Parameters(M')
            for all N'' in RD(M, N)
              AddEdge(G, N'', N');
              AddEdge(G, N', N'');
          AddEdge(G, ReturnValue(M'), I);

TraverseGraph(BT, G, A)
  D := nil;
  for all N in A
    Sub(BT, G, N, D);

Sub(BT, G, N, D)
  if(N ∉ D)
    D := cons(N, D);
    if(N instanceof instruction)
      BT := cons(N, BT);
    for all N' in Edges(G, N)
      Sub(BT, G, N', D);
```

Fig. 11 Logic-based BTA algorithm using Reaching Definitions.

The algorithm consists of the procedure `CreateGraph` and the procedure `TraverseGraph`.

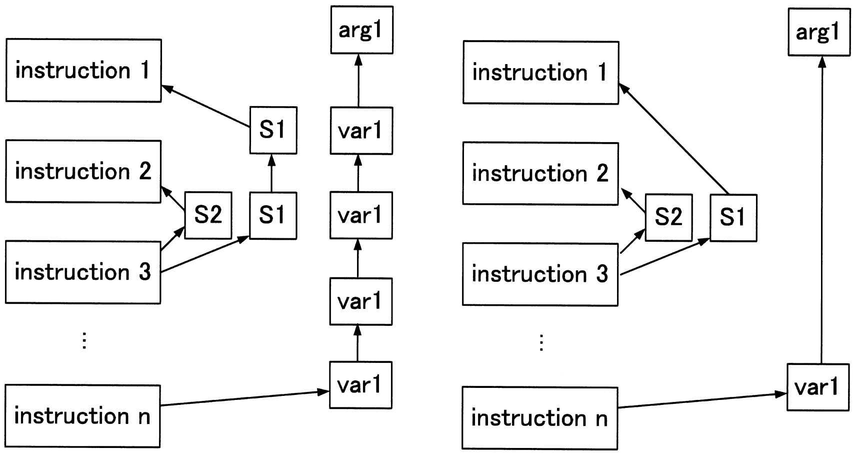

Fig. 12 Graph of the DM (left) and the RDM (right).

`CreateGraph` does the following processes:

(1) Search the call graph in the depth-first manner.
(2) Scan each instruction in the method.
(3) If the instruction `I` is not a method invocation, add an edge from the reaching definitions of `Variables(I)` to `I`.
(4) If the instruction `I` is a return instruction, add a bi-directional edge between the return value and `I`.
(5) If the instruction `I` is a method invocation,
    (a) Collect the potential callee method sets,
    (b) Create the graph of each method `M'`, and
    (c) Add a bi-directional edge between the parameters of `M'` and the reaching definitions of `Variables (I)`.

`TraverseGraph` searches the graph `G` from dynamic arguments `A` in the depth-first manner and makes the binding times of the reached instruction be dynamic.

4.4 Equality between the Direct Method and Reaching Definitions Method

Figure 12 shows graphs of the DM and the RDM. The DM generates a graph with propagating local variables and a stack along control flow graph. In contrast, the RDM generates a graph where it adds edges between instructions depending directly on one another. The start and end instructions in both these graphs are the same. Therefore, results of BTA using both graphs are identical.

4.5 Supporting objects in the Reaching Definitions Method

We show briefly that RDM does not prevent specialization supporting objects based on Ref. 1).

Figure 13 shows binding times of an example code using objects. First, creation of an object should be dynamic if the object is escaped. Next, a field assignment of an object should be dynamic if the object may have visible side-effect. In Fig. 13, we use ideal escape analysis and side-effect analysis. `ESCAPED` and `CAPTURED` means that the objects are escaped and captured, respectively. `SE` means that the assignment has a visible side effect. The center of Fig. 13 shows that underlined expressions and statements are dynamic because `x` is escaped. On the other hand, all expressions and statements in the right of Fig. 13 is static because `x` is captured (not escaped).

In the method `main`, the reaching definition of `x` is the right hand of the first assignment, that is, creation of `LinkedListStack`. The binding times of all use of `x` equals to the binding time of the definition of `x` whether `x` is escaped or not. So does `entry` in the method `add`. This means the RDM does not prevent object creation specialization.

Similarly, there is no dependence between the reaching definitions and side-effects. Consequently, the RDM does not prevent field assignment specialization.

The difference of treatment between objects and other type variables is dealing with their fields. The RDM does not support partially static objects, that is, the binding times of an object and its fields should be same. This rela-

```
class LinkedListEntry {
  LinkedListEntry next = null;
  Object entry;
  LinkedListEntry(Object e) {
    entry = e;
  }
}
class LinkedListStack {
  LinkedListEntry head = null;
  void add(Object e) {
    LinkedListEntry entry
      = new LinkedListEntry(e);
    entry.next = head;
    head = entry;
  }
}
class Main {
  static void main() {
    LinkedListStack x
      = new LinkedListStack();
    x.add(new Object());
    ...
  }
}
```

```
class LinkedListEntry {
  LinkedListEntry next = null;
  Object entry;
  LinkedListEntry(Object e) {
    entry = e^SE;
  }
}
class LinkedListStack {
  LinkedListEntry head = null;
  void add(Object e) {
    LinkedListEntry entry
      = new LinkedListEntry(e)^ESCAPED;
    entry.next = head^SE;
    head = entry^SE;
  }
}
class Main {
  static void main() {
    LinkedListStack x
      = new LinkedListStack()^ESCAPED;
    x.add(new Object()^ESCAPED);
    ...
  }
}
```

```
class LinkedListEntry {
  LinkedListEntry next = null;
  Object entry;
  LinkedListEntry(Object e) {
    entry = e;
  }
}
class LinkedListStack {
  LinkedListEntry head = null;
  void add(Object e) {
    LinkedListEntry entry
      = new LinkedListEntry(e)^CAPTURED;
    entry.next = head;
    head = entry;
  }
}
class Main {
  static void main() {
    LinkedListStack x
      = new LinkedListStack()^CAPTURED;
    x.add(new Object()^CAPTURED);
    ...
  }
}
```

Fig. 13 (left) An example code using objects; (center) Binding times if `x` is escaped; (right) Binding times if `x` is captured.

Table 3 Code size comparison.

	DM	RDM
total code (line)	737	663
instruction-dependent code	241	8

tionship is realized by adding edges between an object and its fields.

5. Implementation and Assessment

5.1 Implementation

We implement binding time analyzers based on the DM and the RDM using bytecode reading and writing features of Javassist [3]. We implement the DM in Java. On the other hand, we implement the RDM in AspectJ [8] using our bytecode analyzer framework [16),17)]. In addition, the implementation of the DM does not support all bytecode instructions currently, but that of the RDM does.

5.2 Code Assessment

Table 3 shows the code size related to BTA directly. Although we cannot compare directly the DM and the RDM because they are based on different implementation basis, the RDM tends to have a 10% smaller code size than the DM.

The more important difference is the code content. To compare the size of instruction-dependent code, although one the DM is 33% of whole BTA, the one of the RDM is a little more than 1%. The instruction-dependent processes of the RDM are only the processes for method invocation and return☆.

5.3 Computational Complexity Assessment

In this sub-section, we compare our approach and constraint-based BTA.

The RDM also has the advantage to constraint-based BTA because the result and the intermediate products of constraint-based BTA is not reusable to another analysis or optimization as far as we know.

Henglein [6] proposes an efficient algorithm of constraint-based BTA using the fact that the binding times are either static or dynamic. He also discusses computational complexity in detail. The complexity of his algorithm is $O(N\alpha(N, N))$, where N is the code size of the target program and α is the inversion of Ackermann's function.

On the contrary, there is no article that discusses the complexity of logic-based BTA, but our experience shows that it is almost linear. The complexity of logic-based BTA essentially equals the complexity of graph construction and traverse. The complexity of graph con-

☆ The instruction-dependent code size does not include the size of its callee. Assume that there are instructions X and Y and instruction-dependent code of X and Y includes an invocation of the common method m. The size of the instruction-dependent code does not include the size of m.

struction of the DM is $O(N)$. That of the RDM equals the one of reaching definitions analysis; there is an algorithm of it whose complexity is $O(N\alpha(N,N))$ according to Ref. 14). On the other hand, the complexity of the graph traverse depends on the graph shape. We cannot conclude how much it is, but in our current implementation graph, construction takes much more time than graph traverse.

6. Concluding Remarks

We have proposed an improvement that uses reaching definitions (the RDM). It offers simple, reusable, and maintainable implementation. Moreover, this paper compared the DM, which is a naive implementation of logic-based binding time analysis, and the RDM, and evaluated improvement of code quality and quantity. We have observed that the RDM has eliminated 10% of the code related to BTA. In particular, the RDM eliminates almost all instruction-dependent code.

In the future, we will enable our binding time analyzer to support partially static objects. We expect that supporting them improves binding times because they often appear in real applications.

Acknowledgments This research was supported in part by a grant of Fukuoka Project in the Cooperative Link of Unique Science and Technology for Economy Revitalization (CLUSTER) of the Ministry of Education, Culture, Sports, Science and Technology (MEXT).

We are grateful to Etsuya Shibayama in Tokyo Institute of Technology and Hidehiko Masuhara in Tokyo University for many helpful comments and stimulating discussions.

References

1) Affeldt, R., Masuhara, H., Sumii, E. and Yonezawa, A.: Supporting Objects in Run-Time Bytecode Specialization, *Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (*ASIA-PEPM*), Aizu, Japan, pp.50–60, ACM Press (2002).

2) Blanchet, B.: Escape Analysis for Object Oriented Languages. Application to Java^{TM}, *Proc. ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (*OOPSLA 1999*), Denver, Colorado, USA, pp.20–34, ACM Press (1999).

3) Chiba, S.: Load-Time Structural Reflection in Java, *Proc. European Conference on Object-Oriented Programming* (*ECOOP 2000*), Sophia Antipolis and Cannes, France, pp.313–336, Springer-Verlag (2000).

4) Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V.C. and Midkiff, S.: Escape Analysis for Java, *Proc. ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (*OOPSLA 1999*), Denver, Colorado, USA, pp.1–19, ACM Press (1999).

5) Clausen, L.R.: A Java Bytecode Optimizer using Side-Effect Analysis, *Concurrency: Practice and Experience*, Vol.9, No.11, pp.1031–1045 (1997).

6) Henglein, F.: Efficient Type Inference for Higher-Order Binding-Time Analysis, *Functional Programming Languages and Computer Architecture* (*FPCA*), Cambridge, Massachhusstts, USA, pp.448–472, Springer-Verlag (1991).

7) Jones, N.D., Gomard, C.K. and Sestoft, P.: *Partial Evaluation and Automatic Program Generation*, Prentice Hall International (1993).

8) Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G.: An Overview of AspectJ, *Proc. European Conference on Object-Oriented Programming* (*ECOOP 2001*), Budapest, Hungary, pp.327–353, Springer-Verlag (2001).

9) Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification*, 2nd Ed., Addison-Wesley (1999).

10) Makholm, H.: Specializing C — An Introduction to the Principles behind C-Mix, `http://www.diku.dk/forskning/topps/activities/cmix/` (1999).

11) Masuhara, H. and Yonezawa, A.: A Portable Approach to Dynamic Optimization in Runtime Specialization, *New Generation Computing*, Vol.20, No.1 (2001).

12) Masuhara, H. and Yonezawa, A.: Run-time Bytecode Specialization: A Portable Approach to Generating Optimized Specialized Code, *Second Symposium on Programs as Data Objects* (*PADO II*), Aarhus, Denmark (2001).

13) Nielson, F., Nielson, H.R. and Hankin, C.: *Principles of Program Analysis*, Springer, Berlin (1999).

14) Ryder, B.G. and Paull, M.C.: Elimination Algorithms for Data Flow Analysis, *ACM Computing Surveys*, Vol.18, No.3, pp.277–316 (1986).

15) Whaley, J. and Rinard, M.: Compositional Pointer and Escape Analysis for Java Programs, *Proc. ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (*OOPSLA 1999*), Denver, Colorado, USA, pp.187–206, ACM Press (1999).

16) Yamazaki, S., Matsumoto, M., Nakanishi, T.,

Kitasuka, T. and Fukuda, A.: Aspect-Oriented Design and Implementation in Java Bytecode Analyzer Framework, *Proc. 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software* (*ACP4IS*), Lancaster, UK (2004).

17) Yamazaki, S., Matsumoto, M., Nakanishi, T., Kitasuka, T. and Fukuda, A.: A Case Study of Development of a Java Bytecode Analyzer Framework Using AspectJ, *IPSJ Transaction on Programming*, Vol.46, No.SIG1 (PRO24), pp.65–77 (2005).

(Received July 2, 2004)
(Accepted October 11, 2004)

Susumu Yamazaki was born in 1972. He received his B.E. degree in metallurgy from Tokyo Institute of Technology, Japan, in 1995 and his M.S. degree in information science from Tokyo Institute of Technology, Japan, in 1997. He took a doctor's course of Tokyo Institute of Technology from 1997 to 2003. He was a researcher of Graduate School of information science and engineering, Tokyo Institute of Technology, Japan in 2003. Since 2003, he has worked for Fukuoka Laboratory for Emerging and Enabling Technology of SoC, Fukuoka Industry, Science and Technology Foundation. His research interests include compilers, systems and development methodologies. He is a member of ACM, IPSJ and JSSST.

Takayuki Kando was born in 1967. He received his M.E. degree from Nagoya University in 1993. He had worked in FUJITSU LIMITED since 1993 to 1998. In there, He had engaged in research on computer algebra systems. He had worked freelance since 2003. His current research interests are generative programming, program specialization and implementation of programming languages for mathematical software. He is a member of JSIAM, JSSAC, JSSST and IPSJ.

Michihiro Matsumoto received the B.S. degree in mathematics from Kyoto University, Japan, in 1990; and the Master's degree and Ph.D. degree in information science from Japan Advanced Institute of Science and Technology, Japan, in 1998 and 2002, respectively. From 1990 to 2003, he was a software engineer, a researcher or a system engineer of PFU Limited. Since 2003, he has been a researcher of Fukuoka Laboratory for Emerging and Enabling Technology of SoC, Fukuoka Industry, Science and Technology Foundation. His research interests include development methodologies for embedded systems.

Tsuneo Nakanishi received the B.E. degree in communication engineering from Osaka University, Japan, in 1993; and the M.E. and D.E. degrees in information science from Nara Institute of Science and Technology, Japan, in 1995 and 1998, respectively. From 1998 to 2002, he was an assistant professor of Graduate School of Information Science, Nara Institute of Science and Technology, Japan. Since 2002, he has been an associate professor of Graduate School of Information Science and Electrical Engineering, Kyushu University, Japan. His research interests include development enviroment for embedded systems.

Teruaki Kitasuka received the B.E. degree in information science from Kyoto University, Japan, in 1993 and M.E. degree in information science from Nara Institute of Science and Technology, Japan, in 1995. From 1995 to 2001, he worked for the Sharp Corporation, where he developed the personal computer. Since 2001, he has been a research associate of Graduate School of Information Science and Electrical Engineering, Kyushu University, Japan. His research interests include mobile computing, embedded systems, parallel and distributed systems, compiler, and computer architecture.

Akira Fukuda received the B.E., M.E., and Ph.D. degrees in computer science and communication engineering from Kyushu University, Japan, in 1977, 1979, and 1985, respectively. From 1977 to 1981, he worked for the Nippon Telegraph and Telephone Corporation, where he engaged in research on performance evaluation of computer systems and the queueing theory. From 1981 to 1991 and from 1991 to 1993, he worked for the Department of Information Systems and the Department of Computer Science and Communication Engineering, Kyushu University, Japan, respectively. In 1994, he joined Nara Institute of Science and Technology, Japan, as a professor. Since 2001, he has been a professor of Graduate School of Information Science and Electrical Engineering, Kyushu University, Japan. His research interests include embedded systems, system software (operating systems, compiler, and runtime systems), mobile computing, parallel and distributed systems, and performance evaluation. He received 1990 IPSJ (Information Society of Japan) Research Award and 1993 IPSJ Best Author Award. He is currently the chair of SIG System Evaluation in IPSJ. He is a member of the ACM, the IEEE Computer Society, the IEICE, the IPSJ, and the Operations Research Society of Japan.