Scalable Computing: Practice and Experience
Volume 6, Number 1, pp. 125–139. http://www.scpe.org
ISSN 1895-1767

STATIC ANALYSIS FOR JAVA WITH ALIAS REPRESENTATION REFERENCE-SET IN HIGH-PERFORMANCE COMPUTING

JONGWOOK WOO*

Abstract. Static Analysis of aliases is needed for High-Performance Computing in Java. However, existing alias analyses regarding $*$ operator for C/C++ have difficulties in applying to Java and are even imprecise and unsafe. In this paper, we propose an alias analysis in Java that is more efficient, at least equivalent, and precise than previous analyses in C++. In the beginning, the differences between C/C++ and Java are explained and a reference-set alias representation is proposed. Second, we present *flow-sensitive* intraprocedural and context-insensitive interprocedural rules for the reference-set alias representation. Third, for the type determination, we build the type table with reference variables and all possible types of the reference variables. Fourth, a static alias analysis algorithm is proposed with a popular iterative loop method with a structural traverse of a CFG. Fifth, we show that our reference-set representation has better performance for the alias analysis algorithm than the existing *object-pair* representation. Finally, we analyze the experimental results.

Key words. Static Analysis, Alias Analysis, Java, Compiler, Parallel and Distributed Computing, Reference-Set, Flow-Sensitive, Context-Insensitive

1. Introduction. Java has become a popular language in distributed and parallel computing because it is platform independent and object-oriented. More specifically, in Java, objects are accessed by references, consequently, there might be many aliases in a piece of Java code. An alias situation occurs when an object is bound by more than two names. Thus, alias information is very useful to avoid side effects from the object bound and codes with alias information are better candidates for high performance computing such as parallelizing compiler as well as compiler optimization. Recent studies [1, 2, 3, 4, 5, 6, 8, 9, 13, 14, 15] have analyzed aliases to avoid side effects statically. With detected aliases information, we may avoid race conditions, context switch and communication overhead for parallelizing computing environment.

Previous studies [2, 3, 4, 5, 6, 8, 9, 18] proposed alias analyses for C/C++ by representing the alias relation with objects because of the concepts of pointers and pointer-to-pointer in C/C++. This research aims at improving the efficiency and the accuracy based on the safety of the alias information detected. However, the representation of alias relations based on * operator is not sufficiently optimized to apply to Java. Thus, we have proposed referred-set representation [1] that makes a more efficient and precise analysis for Java theoretically than previous methods. We can define the *precision* as the metric when all possible data are predicted statically and remove redundant data as possible. However, it might have a large time complexity in the usage of the computed alias set. Thus, in this paper, we propose an alternative alias relation, *reference-set* representation by extending our reference-pair representation which is a pair of reference variables [13]. Within a procedure, *flow-sensitive* analysis is applied. In a *flow-sensitive* intraprocedural analysis, each statement includes all the alias information at the point. The information is propagated to the next statement and subsequently computed in the CFG of a method. Among procedures, context-insensitive call graph is built. Each procedure is represented by a single node in a call graph and analyzed the node even for different calling contexts. Thus, data information is computed efficiently in context-insensitivity graph and the computed data are safe. The *context-sensitive* approach is characterized by a data flow analysis based on path-sensitivity, so each procedure may be analyzed separately for different calling contexts. Thus, we use context-insensitivity calling graph. We also can define the $safety$ as the metric when all possible data are predicted statically and collected so that any possible aliased element is not removed. At a procedure p, $May\ Aliases$ may refer to the same storage location, that is, in some execution instances of the p on some path through a flow-graph. For example, in a procedure p, when one path of p contains $x = y$ and another path $x = z$, we can say that x may refer to y or z. At a procedure p, $Must\ Aliases$ must refer to the same storage location, that is, in all execution instances of the p on all paths through a flow-graph. For example, in a procedure p, when every path of p contain only $x = y$, we can say that x must refer to y.

We analyze the existing alias representations and propagation rules in C++ [2, 3, 4, 6] in order to build much better solution in Java. Our alias analysis in Java presents three contributions for the efficiency and preciseness without losing its safety. First, we introduce the *reference-set* representation to present an alias information in Java. Second, we propose more precise data propagation rules of aliases for the *reference-set*

*Computer Information Systems Department, Simpson Tower Room 604, California State University, Los Angeles, CA 90032-8530 (jwoo5@calstatela.edu).

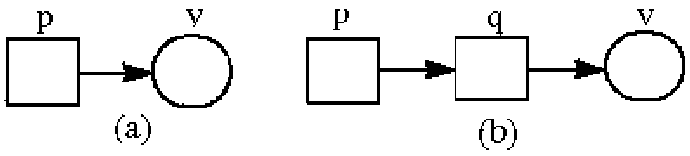

FIG. 1.1. *Relation between a pointer and an object in C++*

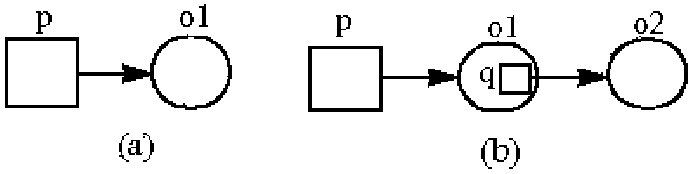

FIG. 1.2. *Relation between a reference and an object in Java*

representation. These rules are based on *flow-sensitive* and context-insensitive analysis. Finally, converting existing algorithms [1, 6, 13] to the *reference-set* representation and the rules, our calling graph (CG), control flow graphs (CFG) and a type table (TT), we propose a more precise and efficient *flow-sensitive* alias analysis algorithm.

In this paper, §2 presents the differences between pointer in C++ and reference in Java. §3 explains the existing studies for aliases. §4 introduces the *reference-set* alias representation for Java. §5 describes the structures to build our algorithm. §6 explains our propagation rules for *Flow-Sensitive* intraprocedural and Context-Insensitive interprocedural analyses. §7 shows the alias analysis algorithm and compares the complexities of our and existing algorithms. §8 shows the experimental results of the *reference-set* and existing *object-pair* representations. Finally, the conclusion is presented.

2. Pointer in C/C++ and Reference in Java. In static analysis of an object-oriented language, naming of an object has been used as data representation for data flow analysis. The object naming is considered to represent aliases in C++ and Java. In C++, static objects declare object names. Also, dynamic objects and pointer-valued objects have their own names for an alias analysis. A pointer variable name is a name to point an object that contains the address of a pointed-to object. In Fig 1.1 (a), pointer variable name p is naming a pointer-valued object that contains its address value. Dereferenced pointer p is naming the object that is pointed to by p. A variable name p that is not a pointer is naming an object that contains the address of the variable. There exist alias relations among pointer-valued objects because of pointer-to-pointer relationships. Therefore, in the previous studies [5, 6, 7], when pointer p points to an object of v, the alias relation is represented as $< *p, v >$. Fig 1.1 (b) shows that a pointer points to another pointer variable that complicates the alias analysis, where a box depicts a pointer-valued object and a circle is a non-pointer object. Those alias relations are represented as $\langle *p, q\rangle$ and $\langle *q, r\rangle$.

However, There are no pointer-to-pointer concepts and no pointer operations in Java. An object in Java is created dynamically so that the object becomes an anonymous object that does not have its own name. Thus, each object needs its own naming by binding a reference name and an object name in an alias relation. A reference is a variable that refers to an object as a pointer in C++. Fig 1.2 represents the same structure of objects and variables in Java as the Fig 1.1 in C++. In Fig 1.1, variable v can exists as an object and its object can be represented as $*v$ with the pointer operator $*$. But, in Fig 1.2 (a), the object $o1$ is referred to by the reference variable p; In Fig 1.2 (b), the object $o1$ is referred to by the variable p and the object $o2$ is referred to by the field q of the object $o1$.

Existing alias relations in C++ are similar compact [5, 6, 7] and *points-to* [2, 3, 4] representation. In this paper, we call them as *object-pair* representation because those are a pair of objects. Those relations save spaces by representing all alias relations without using an exhaustive set. Those relations can be used in Java. However, there are some problems to use those representations because only references are used to name objects in Java. Besides, existing alias representations and the analyses in C++ are based on $*$ operator. In Fig 2.1 (a), if there is an assignment statement $*x = z$, the value of the address valued object named by x is changed to

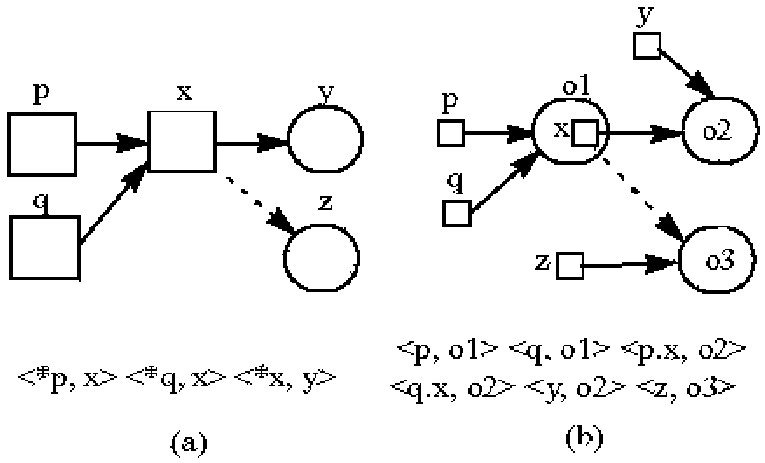

FIG. 2.1. *Difference between a pointer in C++ and a reference in Java*

the value of the address valued object bound by z. Therefore, $< *x, y >$ can be killed and a new alias relation $< x, *z >$ is generated through the compact representation rule [5, 6, 7] as follows.

*object-pair*s A_{IN} before $*x = z$:

$$A_{IN} = \{< *p, x >, < *q, x >, < *x, y >\} \tag{2.1}$$

*object-pair*s after $*x = z$ that implies $x = \&z$:

$$Kill : A_{IN}(*x) = \{\langle *x, y\rangle\} \tag{2.2}$$

$$Gen : \bigcup_{\langle *x,u\rangle \in A_{IN}, AR \in A_{IN}(*z)} \{AR(*u/*z)\} = \{\langle x, x\rangle\} \otimes \{\langle z, z\rangle\} = \{\langle *x, z\rangle\} \tag{2.3}$$

finally, *object-pair*s A_{OUT}

$$\begin{aligned} A_{OUT} &= (A_{IN} - Kill) \cup Gen = (\{\langle *p, x\rangle, \langle *q, x\rangle, \langle *x, y\rangle\} \\ &\quad -\{\langle *x, y\rangle\}) \cup \{\langle *x, z\rangle\} = \{\langle *p, x\rangle, \langle *q, x\rangle, \langle *x, z\rangle\} \end{aligned} \tag{2.4}$$

In Fig 2.1 (b), an alias relation via *compact representation* in Java is shown. If there is an assignment statement $p.x = z$ when the variable z refers to an object $o3$, we can compute alias relations with the *compact representation* rule as follows: *object-pair*s A_{IN} before $p.x = z$:

$$AIN = \{< p, o1 >, < q, o1 >, < z, o3 >, < y, o2 >, < p.x, o2 >, < q.x, o2 >\} \tag{2.5}$$

*object-pair*s after $p.x = z$:

$$\begin{aligned} &Kill : A_{IN}(p.x) = \{\langle p.x, o2\rangle\} \\ &Gen : \bigcup_{<p.x,u>\in A_{IN}, AR \in A_{IN}(z)} \{AR(u/z)\} = \{\langle p.x, p.x\rangle\} \otimes \{\langle z, z\rangle\} \\ &\qquad\qquad = \{\langle p.x, z\rangle\} \end{aligned} \tag{2.6}$$

finally, *object-pair*s A_{OUT} is

$$
\begin{aligned}
A_{OUT} &= (A_{IN} - Kill) \cup Gen \\
&= (\{\langle p, o1\rangle, \langle q, o1\rangle, \langle z, o3\rangle, \langle y, o2\rangle, \langle p.x, o2\rangle, \langle q.x, o2\rangle\} \\
&- \{\langle p.x, o2\rangle\}) \cup \{\langle p.x, z\rangle\} \\
&= (\{\langle p, o1\rangle, \langle q, o1\rangle, \langle z, o3\rangle, \langle y, o2\rangle, \langle q.x, o2\rangle, \langle p.x, z\rangle\}
\end{aligned} \tag{2.7}
$$

However, for the correct relation, $< q.x, o2 >$ of A_{OUT} should be killed. The incorrect result comes from the fact that, in Java, a reference name is used for naming an object without a dereferencing operator such as $*$ in C++. Therefore, we believe that the traditional rule is not applicable to Java to detect precise alias relations as well as *compact representation* may have more aliased elements in Java as shown in Fig 2.1 (b). To obtain the correct result in this example, $p.x$ should be recognized not only as a memory location that contains its addressed value in $< p.x, o2 >$ but also as an object that is referred to by the reference $p.x$. To solve this problem, an alias relation for an address-valued object should be presented by extending a *compact representation*. Otherwise, a data flow equation for aliases should recognize the difference. Therefore, reference names for an alias relation should be meant as dereferencing and the $< p.x, o2 >$ alias relation for the alias computation should be analyzed differently.

3. Related Works. *Pande* [2, 3] presented the first algorithm which simultaneously solved type determinations and pointer aliases with *points-to* alias set representation in C++ programs. *points-to* has the form $< loc, obj >$ where *obj* is an object and *loc* is a memory location of the object *obj*. *points-to* pair is essentially *points-to* relation as introduced by *Emami* [9]. *Emami* proposed it to reduce extraneous alias pairs generated in certain cases with alias pairs of *Landi* [18].

Carini [6] proposed a *flow-sensitive* alias analysis in C++ with *compact representation*. The *compact representation* is an alias relation that has a name object or one level of dereferencing. The *compact representation* of alias relation was introduced by *Choi* [5] to eliminate redundant alias pairs.

Chatterjee [4] presented a *flow-sensitive* alias analysis in object-oriented languages with *points-to* alias set representation in C++. It improves the efficiency and safety of *points-to* alias set representation comparing to *Pande*'s method.

The compact and *points-to* alias representation are highly similar. However, the *points-to* alias representation contains *may* or *must* alias information [5].

Woo [1] introduced a *flow-sensitive* alias analysis in Java with $referred - set$ alias representation, which is alternate to this paper. *Referred-set* is a set of objects that may be pointed by a reference variable and an alias set is a collection of *referred-set*. It is used to reduce extraneous alias pairs while applying the *compact* and *points-to* alias representations in C/C++ to Java.

4. *Reference-Set*: Alias Relation Representation. For a more precise alias analysis in object-oriented languages, the type information of the objects accessed are needed and this information can be collected more safely via alias information [2, 3]. It is known as a type inference. The type information can be used for overridden methods resolution in Java. The more precise the type information, the more precise alias analysis becomes. As shown in §2, to find a correct alias information in Java, we should present an alias representation that does not consider $*$ operator. Otherwise, we have to extend or renewal existing alias computation rules for Java. The second is not easy to implement the rules without $*$ operator. Thus, in this section, we propose the *reference-set* representation and later, we show that it improves the efficiency of the alias computation and the type inference.

DEFINITION 4.1. Reference-set *is a set of alias references that consist of more than two references which refer to an object;* $R_i = r_1, r_2, \ldots, r_j$*: for each* j*, initially* $j > 2$ *and* r_j *is a reference for an object; when* r_j *and* r_k *are in the same path and qualified expressions with a field* f*,* r_j *and* r_k *can be represented with a* $R_i.f$ *with a* reference-set R_i *for an object* i*; During data flow computing in an alias analysis,* $j > 1$ *when passing and passing back an object at a call site.*

The alias set contains the entire alias information at the statement.

DEFINITION 4.2. *Alias set is a set of* reference-set*s at a statement* s*;* $A_s = R_1, R_2, ..., R_i$

In a statement s of a program, each *reference-set* and alias set for the alias relation in Fig 4.1 are represented as follows.

$$R_1 = \{a, b\},\ R_2 = \{R_1.e, c, d\},\ R_4 = \{f.h, g\},\ A_s = \{R_1, R_2, R_4\}$$

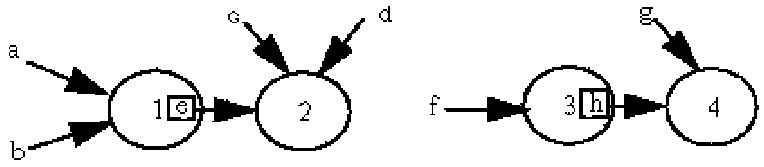

FIG. 4.1. *The relationships between reference and objects*

An alias analysis algorithm computes the alias sets in a program. Each statement collects an alias set from its predecessor and updates it with the statement itself and passes the resulting alias set to its successor(s). Since the alias computation should be iteratively done until the alias sets and a calling graph have converged for the program, it affects the efficiency of the whole algorithms. For example, there is an assignment statement $a = g$ in Fig 4.1. It means that the reference a refers to an object that the reference g refers to. Therefore, the element a of the *reference-set* R_1 is killed and the elements a should be copied to the *reference-set* R_4. The time complexity of this computation depends on the space complexity of each representation. Thus, the efficiency of whole algorithms is improved via *reference-set* representation.

5. Data-Flow Structures. Aliases can be computed with data-flow equations. For the computation of the aliases, we define our data-flow equations, calling graph (CG), and control flow graph (CFG) in this section. A *type table* contains all possible types of reference variables.

A CG is needed to compute the alias set of an interprocedural analysis between a calling and a called methods at a call statement. Our CG is a directed graph defined as $\langle N_{CG}, E_{CG}, n_{main}\rangle$, where N_{CG} is a set of nodes and each node is a method shown one time in a CG even though it may be called many times; where E_{CG} is a set of directed edges connected from caller(s) to callee(s) and one edge is connected even though a caller may invoke the callee many times and all edges are connected when many callers invoke one callee; n_{main} is the main method that executes initially in a Java program. During our algorithm proceeds, our CG grows as in the previous works [4, 5, 6, 7] by adding nodes.

CFGs can be used to compute the alias sets of the *intraprocedural* propagations. Our CFG is a directed graph defined for each method as $\langle N_{CFG}, E_{CFG}, n_{entry}, n_{exit}\rangle$, where N_{CFG} is a set of nodes with n_{entry}, n_{exit}, and each statement of the method; E_{CFG} is the set of directed edges that represent the control and alias set information between a predecessor and a successor statements; n_{entry} represents the entry node of the method; n_{exit} represents the exit node of the method.

In our CFG, seven node types are proposed based on their purposes: *Entry* that is the n_{entry} of the CFG, *Exit* that is the n_{exit} node, *Assignment statement, Call statement, Return Statement, Flow construct node (if, while etc.)*, and *Merging nodes*.

The *flow construct node* is a node which signifies the start of the if or *while* clause. In an If node, each clause is branched from the node. All the branched clauses are merged into a merging node. In a *while* node, a *merging node* is not necessary and a directed edge is connected from the last node of the while loop to the *flow construct node*.

Reference variables dynamically refer to objects in Java. Thus, the types of a reference variable are determined statically at the type declaration and dynamically during the processing of the algorithm. A *type table* is built during the process and it contains three columns: the reference variable, its declared type, and its overridden method types. Type inference can be processed with a *type table* which contains the declared and dynamic types of each reference variable. The declared type represents static and shadowed variable type information of a reference variable. The dynamic types represent possible overridden method types of the reference variable. Types of each reference variable can be given in a constant time.

6. *Flow-Sensitive* Context-Insensitive Rules for *Reference-Set*. The propagation and computation of alias information is made through the nodes in a CFG of each method. Let $in(n)$ and $out(n)$ be the input alias set of a node n transferred from predecessor nodes and output alias set held on *exit* from a node n respectively.

$$in(n) = \bigcup out(pred(n))$$
$$out(n) = Trans(in(n)) = Mod_{gen}[Mod_{kill}(in(n))] \tag{6.1}$$

In this equation, $pred(n)$ represents a predecessor node of the node n. Mod_{kill} denotes the alias set modified after killing some *reference-set*s of $in(n)$ and Mod_{gen} is the subsequent alias set after generating the new *reference-set*s on Mod_{kill}.

6.1. *Flow-Sensitive* Intraprocedural Analysis. The propagation rules for *intraprocedural* analysis are described below for every CFG node type except an *entry* and a *call statement node* type. The rule consists of premises and conclusions divided by a horizontal line. The premise can have the form of conditional implication that is interpreted as follows: when a given condition holds, the implied equation has a meaning and can be solved. When all premises hold, the equations in the conclusions are solved for $out(n)$.

First, we define a *flow construct* and *merging node* type rule in (6.2): n_{pred} is a predecessor set of node n. Given n_{pred}, $out(n)$ of node n is the union of all predecessor node sets.

$$in(n) = \bigcup_{p \in n_{pred}} out(p)$$

$$\frac{n_{pred} : predecessor\ node\ of\ n}{out(n) = in(n)} \tag{6.2}$$

The next rule (6.3) concerns the node type of an *assignment statement*.

$$\begin{gathered} in(n) = out(npred) \\ n_{pred} : predecessor\ node\ of\ n \\ x = LHS, \\ y = RHS, \\ \forall\, i, j\ Ri, Rj \in in(n) \rightarrow [Mod_{kill}(in(n)) = \{R_i \mid kill\, x \in R_i\} \\ \cup \{R_i \mid\ kill\, R_j.f \in R_i\ when\, q \in R_j\ and\, x = q.f\}] \text{.....①} \\ \wedge [in(n) = in(n) - Mod_{kill}(in(n))] \wedge [KILL(in(n)) = \{x,\ R_j.f\}], \text{.....②} \\ \forall k\ R_k \in in(n) \rightarrow [Modgen(in(n)) = \{R_k \mid R_k = R_k \cup KILL(in(n))\ when\, y \in R_k\}] \\ \wedge [in(n) = in(n) - Mod_{gen}(in(n))], \text{.....③} \\ \hline out(n) = Mod_{kill}(in(n)) \cup Mod_{gen}(in(n)) \cup in(n) \end{gathered} \tag{6.3}$$

LHS and RHS respectively stand for the left hand side and the right hand side of an assignment statement. $KILL(in(n))$ is a *reference-set* of references killed by $Mod_{kill}(in(n))$. At a statement ①, $\{R_i \mid kill\ x \in R_i\}$ is to remove the element x from the set R_i. $out(n)$ of the node n is a union of $Mod_{kill}(in(n))$, $Mod_{gen}(in(n))$, and $in(n)$.

In order to show how the above rule can be applied to alias analysis, we analyze an *assignment statement* $a.e = f.h$ in a statement of Fig 4.1. Initially, *reference-set* R_1, R_2, R_3 and alias set $in(n)$ are expressed as follows for the statement:

$$\begin{gathered} R1 = \{a, b\}\ \ R2 = \{R_1.e, c, d\}\ \ R3 = \{f.h, g\} \\ in(n) = \{R_1, R_2, R_3\} \end{gathered}$$

Because LHS is a qualified expression related to both R_1 and R_2, $Mod_{kill}(in(n))$, $in(n)$, and $KILL(in(n))$ are computed based on ① and ② as follows:

$$\begin{gathered} R_1 = \{a, b\}\ and\ R_2 = \{R_1.e, c, d\}\ then\ R_2 = \{c, d\} \\ Mod_{kill}(in(n)) = \{R_2\},\ in(n) = \{R_1, R_3\},\ KILL(in(n)) = \{R_1.e\} \end{gathered}$$

Since R_3 includes RHS, $Mod_{gen}(in(n))$ and $in(n)$ are computed based on ③ as follows:

$$Mod_{gen}(in(n)) = \{R_3 \mid R_3 = R_3 \cup \{R_1.e\} = \{R_1.e, f.h, g\}\} = \{R_3\},\ in(n) = \{R_1\}$$

Finally, $out(n)$ is the union set of $Mod_{kill}(in(n))$, $Mod_{gen}(in(n))$, and $in(n)$ as follows:

$$out(n) = Mod_{kill}(in(n)) \cup Mod_{gen}(in(n)) \cup in(n) = \{R_2, R_3, R_1\}$$
$$when\ R_1 = \{a, b\},\ R_2 = \{c, d\},\ R_3 = \{R_1.e, f.h, g\}$$

The following is the another example to detect *reference-set* of Fig 2.1 (b) by assuming the current node as n with the assignment statement $p.x = z$:

$$R_1 = o1 = \{p, q\},\ R_2 = o2 = \{p.x, q.x, y\} = \{R_1.x, y\},\ R_3 = o3 = \{z\}$$
$$in(n) = \{R_1, R_2, R_3\}$$

Because LHS is a qualified expression related to R_1, $Mod_{kill}(in(n))$, $in(n)$, and $KILL(in(n))$ are computed based on ① and ② of *Assignment Node* rule (6.3) as follows:

$$R_1 = \{p, q\}\ and\ R_2 = \{R_1.x, y\}\ then\ R_2 = \{y\}$$
$$Mod_{kill}(in(n)) = \{R_2\}\ in(n) = \{R_1\}\ KILL(\ in(n)) = \{R_1.x\}$$

Since R_3 includes RHS, $Mod_{gen(in(n))}$ and $in(n)$ are computed based on ③ of *Assignment Node* rule (6.3) as follows:

$$Mod_{gen}(in(n)) = \{R_3 \mid R_3 = R_3 \cup \{R_1.x\}\} = \{z, R_1.x\} = R_3,\ in(n) = \{R_1\}$$

Thus, $out(n)$ is the union set of $Mod_{kill}(in(n))$, $Mod_{gen}(in(n))$, and $in(n)$ as follows:

$$out(n) = Mod_{kill}(in(n)) \cup Mod_{gen}(in(n)) \cup in(n) = \{R_2, R_3, R_1\}$$
$$when\ R_1 = \{p, q\},\ R_2 = \{y\},\ R_3 = \{z, R_1.x\}$$

Finally, $out(n)$ has the following $referenceset$, which has the correct aliased elements:

$$out(n) = \{R_1, R_2, R_3\}\ when\ R_1 = \{p, q\},\ R_2 = \{y\}, R_3 = \{z, R_1.x\}$$

With this example for Fig 2.1 (b), we have shown that our reference set of $out(n)$ is more precise than the aliased elements A_{OUT} of the *compact representation* shown in the §2.

The rule (6.4) for the *return statement node* type is presented as follows with the *reference-set* of a return variable r. In the rule, $LOCAL$ stands for a local variable set defined in a method M such as local variable and formal parameter variables.

$$\frac{\begin{array}{c} in(n) = out(npred) \\ n_{pred} : predecessor\ node\ of\ n \\ M : callee,\ LOCAL(M) = \{v \mid v\ is\ a\ local\ variable\ of\ M\} \\ \forall i\ R_i \in in(n)\ for\ r :\ return\ reference \\ \rightarrow [Mod_{kill}(in(n)) = \{R_i \mid kill\ x \in R_i\ for\ x \in LOCAL(M)\}] \wedge \\ [R_r = \{R_r | kill\ x \in R_r\ for\ x \in LOCAL(M)\ when\ r \in R_r\}], \end{array}}{out(n) = Mod_{kill}(in(n))} \tag{6.4}$$

The next is the rule (6.5) for an *exit node* type.

$$in(n) = \bigcup_{p \in n_{pred}} out(p),$$
$$n_{pred} :\ predecessor\ node\ of\ n,$$

$$\frac{\begin{array}{c} M : callee,\ LOCAL(M) = \{v \mid v\ is\ a\ local\ variable\ of\ M\}, \\ \forall i\ R_i \in in(n) \\ \rightarrow [Mod_{kill}(in(n)) = \{R_i \mid kill\ x \in Ri\ for\ x \in LOCAL(M)\}] \\ \wedge [in(n) = in(n) - Mod_{kill}(in(n))] \end{array}}{out(n) = Mod_{kill}(in(n)) \cup in(n)} \tag{6.5}$$

6.2. Context-Insensitive Interprocedural Analysis. Interprocedural propagation rules should be considered for a *call statement node* and an *entry node*. The data flow of an alias set in a *call statement* denotes that an alias information of the statement is propagated to a called method and it affects an alias information of the called method. The affected information are passed back to the *call statement* of the calling method after computing the alias set of the called method. The alias set from the called method modifies the alias set of the *call statement* when the return alias set includes non-local variables and actual parameters.

We virtually divide a call node into a *precall node* and a *postcall node* to simplify the computation of a call statement. A *precall node* collects an alias set from a *predecessor node* of a current call node and computes its own alias set $out(n)$ with the collected set. This alias set is propagated to the *entry node* of the called method. During the propagation, the $reference - sets$ for references which are inaccessible from the called method are killed. Since this set is an input of the *postcall* node and is not modified, it does not need to propagate to the called method. The $out(n)$ of the *precall* node is not propagated to the *postcall* node because the called method might modify the set. As in previous approaches [2, 3, 4], if we do not kill the alias relations affected by the called method for the subsequent analysis, it might build nonexistent call relations and cause the subsequent analysis to become inefficient.

A *postcall* node collects the modified kill set of the *precall* node and exit nodes alias set of all possible called methods. The following rule (6.6) computes an out set of a *precall* node.

$$\frac{\begin{array}{c} in(n) = out(n_{pred}), \\ n_{pred} : predecessor\ node\ of\ n \\ RHS = E_c.M_c, \\ RHS = M_c, \\ \forall i,\ a_i = the\ ith\ actual\ parameter\ of\ the\ callee\ Mc, \\ \forall i,\ f_i = the\ ith\ formal\ parameter\ of\ the\ callee\ M_c, \\ \forall i,\ R(a_i) \in in(n) \rightarrow [R_{pass}(a_i) = \{a_i, f_i\}] \wedge [R(a_i) = R(a_i) - R_{pass}(a_i)], \\ RHS = M_c,\ \forall i,\ R(a_i) \in in(n),\ v\ is\ a\ non\ local\ variable\ in\ the\ callee, \\ M_c \rightarrow [R(v) = R(v) - \{v\}] \wedge [R_{pass}(v) = \{v\}] \wedge [PASS(M_c) = \cup\{R_{pass}(ai), R_{pass}(v)\}], \\ RHS = E_c.M_c, \forall i,\ R(a_i) \in in(n) \forall f, \\ R(E_c.f) \in in(n) \rightarrow [R(E_c.f) = R(E_c.f) - \{E_c.f\}] \wedge \\ [R_{pass}(Ec.f) = \{Ec.f\}] \wedge [PASS(M_c) = \cup\{R_{pass}(a_i), R_{pass}(E_c.f)\}] \end{array}}{out(n) = in(n)} \tag{6.6}$$

$PASS(M_c)$ represents the set of actual, formal parameters and non-local variables in a called method M_c.$R_{pass}(a_i)$ is a set of reference variables accessible by a called method when passing from a caller to the called method M_c.$R_{pass}(v)$ is a set of non-local variables accessible by a called method in the called method M_c.

In the following propagation rule (6.7) of an *entry node*, the propagated set can be computed as in an *assignment statement node*. $PRECALL(M_c)$ is a *precall node* of *call statement node*s that invoke this called method node. This set can be computed by considering ingoing edges of the called method M_c in a CG. An entry node merges alias sets from the *precall* nodes and then propagates the merged set to its subsequent node.

$$\frac{\begin{array}{c} in(n) = \bigcup_{p \in PRECALL(M_c)} PASS(p), \\ PRECALL(M_c) : a\ precall\ node\ of\ the\ callee\ M_c \end{array}}{out(n) = in(n)} \tag{6.7}$$

The rule (6.8) of the *postcall node* is defined as follows.

$$
\begin{gathered}
in(n) = \bigcup_{p \in n_{precall}} out(n_{precall}) \\
n_{precall} : \ a\ precall\ node\ of\ n \\
RHS = E_c.M_c \\
\rightarrow FIELD(E_c) = \{f \mid \ f\ is\ a\ field\ name\ in\ an\ object\ referred\ by\ E_c\}, \\
RHS = M_c \rightarrow FIELD(E_c) = \phi, \\
RHS = new\ M_c \rightarrow FIELD(E_c) = \phi \wedge A(r), \\
EXIT(M_c) = \{e \mid e\ is\ an\ exit\ alias\ set\ from\ a\ possible\ callee\ method\ M_c\}, \\
LHS = \phi, \forall R_{passb} \in EXIT(M_c) \\
\rightarrow [R_{passb} = R_{passb} - \{v \mid v\ is\ a\ local\ variable\ in\ the\ callee\ M_c\}] \\
\wedge [EXIT(M_c) = \bigcup_{for\ all\ M_c} R_{passb}], \\
LHS \neq \phi, \ \forall R_{passb} \in out(return\ node) \\
\rightarrow [R_{passb} = R_{passb} - \{v \mid v\ is\ a\ local\ variable\ in\ the\ callee\ M_c\}] \\
\wedge [EXIT(M_c) = \bigcup_{for\ all\ M_c} R_{passb}], \\
\forall i\ R_i \in EXIT(M_c), \forall j R_j \in in(n) \rightarrow [R_i \mid R_i = R_i \cup R_j\ when\ i = j] \\
\wedge [EXIT(M_c) = EXIT(M_c) - R_i] \wedge [in(n) = in(n) - R_j], \\
exit(RHS) = \bigcup_{e \in EXIT(Mc)} out(e) \cup \bigcup_{p \in n_{precall}} out(precall\ node) \cup \bigcup_{for\ all\ i} R_i, \\
LHS = \phi \rightarrow out = exit(RHS), \\
LHS = x, \forall i, j\ R_i, R_j \in exit(RHS), R(RHS) \in exit(RHS) \\
\rightarrow [Mod_{kill}(exit(RHS)) = \{R_i \mid kill\ x \in Ri\} \\
\cup \{R_i \mid kill\ R_j.f \in R_i\ when\ q \in R_j\ and\ x = q.f\}] \wedge [KILL(exit(RHS)) = \{x, R_j.f\}] \\
\wedge [exit(RHS) = exit(RHS) - Mod_{kill}(exit(RHS))], \\
R(RHS) \in exit(RHS) \rightarrow [Mod_{gen}(exit(RHS)) \\
= \{R(RHS) \mid R(RHS) = R(RHS) \cup KILL(exit(RHS))] \\
\wedge [exit(RHS) = exit(RHS) - Mod_{gen}(exit(RHS))] \\
\underline{\wedge [out = Mod_{kill}(exit(RHS)) \cup Mod_{gen}(exit(RHS)) \cup exit(RHS)]} \\
out(n) = out
\end{gathered}
\tag{6.8}
$$

$Exit(RHS)$ is a set of exit nodes of all possible called methods as explained before. We can compute $exit(RHS)$ in a CG by integrating all $out(precall\ node)$ and outgoing edges from callers and their exit nodes. Out is an alias set of the exit node of a callee. If we assume the Fig 6.1 (a) as a status after executing a statement s, the alias set A_s of the statement s is:

$$
\begin{gathered}
A_s = R_2, R_3 \\
where\ R_2 = \{a.f, b, c, R_3.f\}\ and\ R_3 = \{R_2.f, c\}
\end{gathered}
$$

After executing the call statement t in Fig 6.1 (b), the result alias set of its *precall node* can be computed in the following sequence of rule applications:

$$
\begin{gathered}
in(t) = \{R_2, R_3\}, \\
RHS = a.update(c), \\
a_i = c, f_i = i, \\
R_{pass}(ai) = \{c, i\}, R(a_i) = R_2 = R_2 - \{c\} = \{a.f, b, R_3.f\}
\end{gathered}
$$

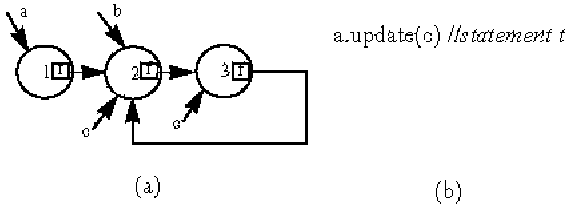

FIG. 6.1. *Example of an interprocedural analysis*

$$
\begin{gathered}
or\ R(a_i) = R_3 = R_3 - \{c\} = \{R_2.f\}, \\
R(a.f) = R_2 = R_2 - \{a.f\} = \{b, R_3.f\}\ and\ R_{pass}(a.f) = a.f, \\
PASS(a.update) = \{R_{pass}(ai), R_{pass}(a.f)\}, \\
out(t) = \{R_2, R_3\}
\end{gathered}
$$

The $PASS(a.update)$ of the precall node propagates to the *entry node* of the callee *update*(). The result alias set of the *exit node* can be computed as follows:

$$
\begin{gathered}
R_{pass}(a_i) = \{c, i\}, R_{pass}(a.f) = \{a.f\}, \\
R_{pass}(a_i) = R_{pass}(a.f) = \{c, i, a.f\} = R_{pass}(R_2)\ for\ R_2, \\
R_{pass}(ai) = \{c, i\} = R_{pass}(R_3)\ for\ R_3, \\
in(u) = \{R_{pass}(R_2), R_{pass}(R_3)\}, \\
R(b) = \{b, i.f, c.f\}, \\
out(u) = updateexit = \{R(b), R_{pass}(R_2), R_{pass}(R_3)\}
\end{gathered}
$$

The result set of the *postcall node* at the statement t is computed with the exit alias set of the *update*() and the propagation rule of the postcall node as follows:

$$
\begin{gathered}
in(t) = out(t_{precall}) = \{R_2, R_3\}\ where\ R_2 = \{b, c.f\}, R_3 = \{c.f\}, \\
FIELD(a) = \{a.f\}, \\
EXIT(update) = update_{exit} = \{R(b), R_{pass}(R_2), R_{pass}(R_3)\}, \\
where\ R_{pass}(R_2) = \{c, i, a.f\}\ and\ R_{pass}(R_3) = \{c, i\}, \\
R(b) = R_{passb} = \{b, c.f\},\ R_{passb}(R_2) = \{c, a.f\},\ R_{passb}(R_3) = \{c\}
\end{gathered}
$$

for the caller,

$$
\begin{gathered}
EXIT(update) = \{R(b), R_{passb}(R_2), R_{passb}(R_3)\}, \\
R_2 = R_2 \cup R_{passb} \cup R_{passb}(R_2) = \{b, R_3.f, c.f, c, a.f\}, \\
R_3 = R_3 \cup R_{passb} \cup R_{passb}(R_3) = \{R_2.f, b, c.f, c\},
\end{gathered}
$$

Thus, $out(t) = R_2, R_3$

7. Static Analysis Algorithm for Aliases. Algorithm 7.1 represents our alias analysis algorithm. This algorithm is adapted on the interprocedural type analysis algorithm [6]. It is one of iterative methods for interprocedural data flow analysis based on a *CG* [10]. The iterative algorithm executes its computation by visiting all nodes of a *CG* in order until fixed point of the data status and nodes are achieved.

Our algorithm traverses each node of a *CG* in a topological order and a reverse topological order in order to possibly shorten the execution time for the fixed point [5, 6, 7]. The ending point in our algorithm means that the topology of a *CG* and alias set $out(n)$ are not changed anymore.

The set $TYPES$ represents the possible class types for a callee to build a safe *CG*. $TYPES_{table(r)}$ is a set of dynamic types of a reference variable r in a *type table*. The reference r also maintains its static type in the

Algorithm 7.1 StaticAliasAnalysis

```
construct an initial CG with main method;
repeat
  for all node n ∈ N_cfg(T.M) in structural order do
    for all node n ∈ N_cfg(T.M) in structural order do
      if n is a call statement node then
        if (RHS = E_c.M_c) then
          compute the set of inferred types from the reference-set for E_c.
          compute the set TYPES resolved from the inferred types and class hierarchy.
        else if (RHS = M_c) then
          TYPES := {T}
        else if (RHS = newM_c) then
          TYPES := {M_c}
        end if
        if LHS exists then
          {TYPES_table(LHS) = TYPES_table(RHS);}
        end if
        for all type t ∈ TYPES do
          if t.M_c is not in CG then
            create a CG node for Mc;
          end if
          if no edge from T.M to t.M_c with a label n then
            connect an edge from T.M to t.Mc with a label n;
          end if
        end for
        compute out(n_precall) for a precall node n_precall;
        compute out(n_postcall) for a postcall node n_postcall;
      else
        if n is an assignment statement node then
          TYPES_table(LHS) = TYPES_table(RHS);
        end if
        if n is a merging statement node then
          TYPES_table(LHS) = TYPES_table(LHS) + TYPES_table(RHS);
        end if
        compute out(n) using data-flow equation and propagation rule;
      end if
    end for
  end for
until CG and alias set for every CFG node converge
```

type table. Inner loop is for *intraprocedural* alias analysis for each method. While proceeding the inner loop, each node of the CFG of a method is traversed with computing an alias set of each node and the computed alias set is propagated to the next node. Each node in our algorithm is visited on structural order for this. Structural order is defined that while visiting nodes from an entry node to an exit node, for the if flow construct node, each branch is traversed first then finally its merging node is visited. We improve the efficiency with the structural order than the previous work [6].

When the method of a *call statement node* is an overridden method, its resolution should be considered for the safety of the alias set. It is not possible to precisely predict the dynamic overridden method statically. However, we can store all possible types of each reference into a *type table* during computation. Thus, we can safely predict all possible methods invoked with the *type table*. If the method is defined in a type inferred by this type inference, the method defined is a resolved method. By adding all possible methods to the CG via the searching, we can update the CG.

An alias set of each node can be computed as a result alias set $out(n)$ with type information and our

TABLE 8.1
Characteristics of hosts

	Kottos	Ceng	Asadal
Host Type	RS6000	Sun4	Windows 2000
OS	AIX4	SunOS5.6	Windows NT
Java VM	JDK-1.1.1	JDK-1.2.1 _02	JDK-1.2.2

data-flow equations of the propagation rules. Our algorithm has three outer loops. For the most outer loop, R_n and A_r are the number of $reference-sets$ and the maximum number of aliased reference variables for each *reference-set*. We can estimate the worst time complexity of the loop as $O(Rn \times Ar \times Ecg)$ - E_{cg} is the number of edges in a CG. For the second outer loop, the time complexity becomes $O(N_{cg})$ if N_{cg} is the final number of nodes in a CG. For the most inner loop, the time complexity is $O(N_{cfg})$ if N_{cfg} is the maximum number of nodes in a CFG that consists of the maximum number of nodes.

The time complexity of a set of inferred types is $O(R_m)$ when R_m is the number of reference variables in a program code. The time complexity for the possible method resolution is $O(T_i \times H)$ when T_i is the maximum number of subclasses for a superclass and H is the maximum number of the levels in its hierarchy. The time complexity for the resolution of overridden methods and the updating of a CG is $O(T_i \times (H + Ncg + Cc))$ when C_c is the maximum number of call statements to invoke same called methods in a calling method. The worst time complexity of a *precall* and a *postcall* nodes is $O(R_p \times R)$ when R_p is the maximum number of *reference-set*s propagated and R is the maximum number of reference variables in R_p on a call statement.

Therefore, the worst time complexity of the main algorithm is $O(R_n \times A_r \times E_{cg} \times N_{cg} \times N_{cfg} \times (R_p \times R \times R_m + T_i \times (H + N_{cg} + C_c)))$.

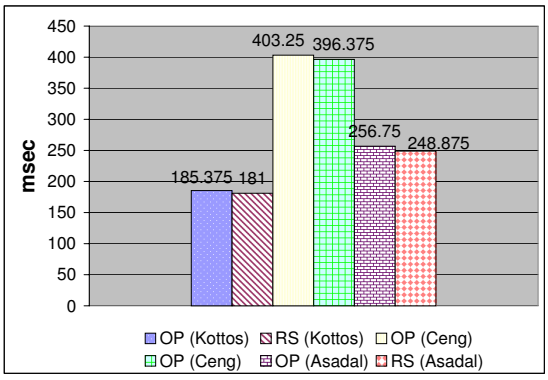

FIG. 8.1. *Run Time of Overridden Methods*

8. Experimental Results. We have executed benchmark codes on alias analysis algorithms with the *reference-set* and the existing *object-pair* representations [2, 3, 4, 6, 7]. We only focus on time for the experiment because the alias detected for these two executions are implicitly the same—the benchmark does not have the indirect object relations that may generate imprecise aliases with the existing work. The algorithm is the part of a Java compiler. Thus, the execution time is to measure the compilation time of the codes. We believe that the theoretic approach of this paper is enough to show our *reference-set* representation is both safe and precise without losing the efficiency. However, we have the three benchmark codes executed, which are: $Overridden\ Methods$, $Binary\ Tree$, and $Ray\ Tracer$. It is to present that our approach runs at least with the similar efficiency comparing to the existing *object-pair* representations [2, 3, 4, 6, 7]. Each benchmark code is executed on three different hosts $Kottos$, $Ceng$, and $Asadal$, which is to collect many experimental results. Table 8.1 presents properties of the hosts for those benchmark codes. $Kottos$ is RS6000 IBM machine; $Ceng$

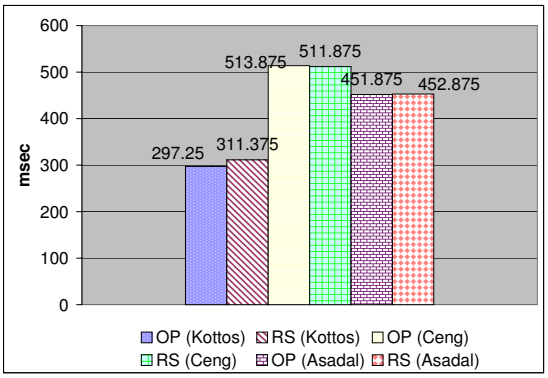

FIG. 8.2. *Run Time of Ray Tracer*

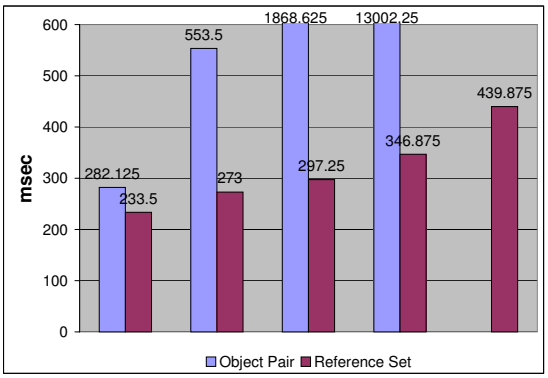

FIG. 8.3. *Run Time of Binary Tree at Kottos*

is Sun4 Unix machine; *Asadal* is Windows machine. The alias analysis systems are built on JavaCC (Java Compiler Compiler) [11] and JTB (Java Tree Builder) [12]. JavaCC is the parser generator and JTB is a syntax tree builder to be used with the JavaCC parser generator. It automatically generates a JavaCC grammar with the proper annotations to build the syntax tree during parsing [12]. The syntax tree is extended by adding the data structures of *reference-set* and *object-pair* representations with class structures of TT and CFG [13].

Overridden Methods is written in C++ initially by *Carini* [6] and adapted in Java by ourselves. It has conditional statements and overridden methods. *Binary Tree* is provided by Proactive group [14]. The binary tree contains many conditional statements and recursive calls that generate potential aliases dynamically. Both can be used to measure the safety, preciseness, and efficiency of the algorithms. *RayTracer* is one of Java Grande's benchmarks to measure the performance of a 3D raytracer [15].

Fig 8.1 presents the execution times of *Overridden Methods*. For all hosts, the execution time of *reference-set* is faster than *object-pair* because our *Type Table* has more efficient structure to search possible types of methods than Carini's [6]. Fig 8.2 is the execution times of *Ray Tracer*. It implies that the benchmark codes such as *Ray Tracer*, which do not contain many aliased references among objects and which is for JVM performance measurement, do not have any big difference in the execution time of alias analysis for any alias representation. For *Binary Tree*, Fig 8.3, Fig 8.4, and Fig 8.5 show that the execution time of *reference-set* becomes faster than *object-pair* relatively as the depth of the recursive calls increase. For *Kottos* of Fig 8.3,

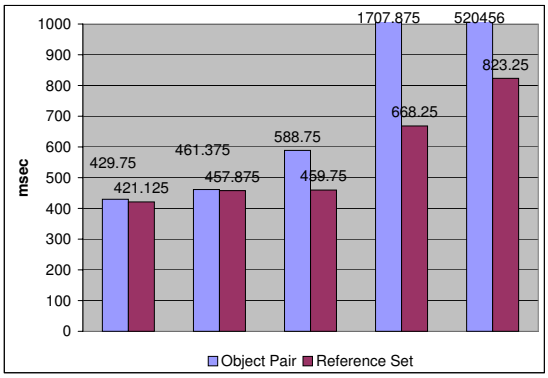

FIG. 8.4. *Run Time of Binary Tree at Ceng*

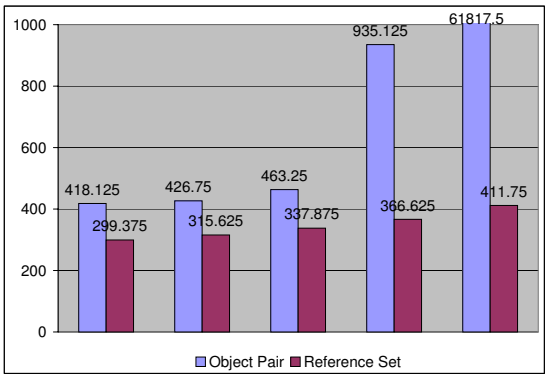

FIG. 8.5. *Run Time of Binary Tree at Asadal*

object-pair is not measurable because the execution time is too long. The result of *Binary Tree* for each host is reasonable because the performance test shows that Windows and Sun hosts are much faster than IBM for Java benchmark [19, 20]. As a result, we can see that *reference-set* is more efficient than *object-pair* for benchmark which has many aliased objects such as *Binary Tree*. Besides, it does not negatively affect the performance for benchmark which does not have many aliased objects such as *Overridden Methods* and *Ray Tracer.*

9. Conclusion. We propose the *flow-sensitive context-insensitive* static analysis algorithm for aliases with *reference-set* alias representation in Java by adapting existing alias analyses [6, 7] in C or C++. We show theoretically that the algorithm is more precise, safe, and efficient than previous studies [2, 3, 4, 6, 7] by using the *reference-set* alias representation, the structural traverse of a *CFG*, and the data propagation rules for the representation. Finally, we show in the experimental results that our approach is more efficient, at least equivalent, comparing to the previous studies [2, 3, 4, 6, 7].

REFERENCES

[1] JEHAK WOO, JONGWOOK WOO AND JEAN-LUC GAUDIOT, *Flow-Sensitive Alias Analysis with Referred-Set Representation for Java*, The Fourth International Conference/Exhibition on High Performance Computing in Asia, pp 485-494, May 2000.
[2] H. D. PANDE AND B. G. RYDER, *Static Type Determination and Aliasing for C++*, LCSR-TR-236, Rutgers Univ., 1995.

[3] H. D. Pande and B. G. Ryder, *Data-flow-based Virtual Function Resolution*, Static Analysis: Third International Symposium (SAS'96), LNCS 1145, Sept., 1996.
[4] R. Chatterjee and B. G. Ryder, *Scalable, flow-sensitive type inference for statically typed object-oriented languages*, Technical Report DCS-TR-326, Rutgers Univ., Aug. 1997.
[5] Jong-Deok Choi, Michael Burke, and Paul Carini, *Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects*, The 20th ACM SIGACT-SIGPLAN Symposium on POPL, pp 232-245, January 1993.
[6] Paul Carini and Harini Srinivasan, *Flow-Sensitive Type Analysis for C++*, Research Report RC 20267, IBM T. J. Watson Research Center, November 1995.
[7] Michael Burke, Paul Carini, and Jong-Deok Choi, *Interprocedural Pointer Alias Analysis*, Research Report RC 21055, IBM T. J. Watson Research Center, December 1997.
[8] Barry K. Rosen, *Data flow analysis for procedural languages*, JACM, 26(2):322-344, April 1979.
[9] M. Emami, R. Ghiya, and L. J. Hendren, *Context-sensitive interprocedural point-to analysis in the presence of function pointers*, SIGPLAN '94 Conference on Programming Language Design and Implementation, pp 242-256, 29(6), 1994
[10] S. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Academic Press, July 1997.
[11] Sun Micro Systems, *JavaCC, The parser Generator*, http://www.suntest.com/JavaCC/, V0.8pre2, 1998.
[12] Purdue University, *West Lafayette, Indiana, USA, Java Tree Builder*, http://www.cs.purdue.edu/jtb/index.html, 2000.
[13] Jongwook Woo, Isabelle Attali, Denis Caromel, Jean-Luc Gaudiot, and Andrew L Wendelborn, *Alias Analysis On Type Inference For Class Hierarchy In Java*, The 24th ACSC 2001, pp 206-214, Jan 29-Feb 2, 2001.
[14] Jongwook Woo, Jehak Woo, Isabelle Attali, Denis Caromel, Jean-Luc Gaudiot, and Andrew L Wendelborn, *Alias Analysis For Java with Reference-Set Representation*, The 8th ICPADS 2001, pp 459-466, June, 2001.
[15] Jongwook Woo, Jehak Woo, Isabelle Attali, Denis Caromel, Jean-Luc Gaudiot, and Andrew L Wendelborn, *Alias Analysis For Exceptions In Java*, The 25th ACSC 2002, pp 321-330, Jan, 2002.
[16] Inria, *ProActive*, Sophia, http://www.inria.fr/oasis/ProActive
[17] Java Grande Forum, http://www.javagrande.org/
[18] W. Landi and B. G. Ryder, and S. Zhang, *A Safe Approximating Algorithm for interprocedural Pointer Aliasing*, in Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, pp 235-248, June 1992.
[19] Pendragon Software *CaffeineMark 3.0*, http://www.pendragon-software.com/pendragon/cm3/results.html
[20] National Institute Standards and Technology, *SciMark 2.0*,http://math.nist.gov/cgi-bin/ScimarkSummary

Edited by: Andrzej Goscinski
Received: December 17, 2002
Accepted: February 02, 2004