Electronic Notes in Theoretical Computer Science 82 No. 2 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 15 pages

Code Annotation for Safe and Efficient Dynamic Object Resolution

Andreas Hartmann, Wolfram Amme [1]

Institut für Informatik
Friedrich-Schiller-Universität Jena
Jena, Germany

Jeffery von Ronne, Michael Franz [2]

Information and Computer Science
University of California, Irvine
Irvine, CA, United States

Abstract

The execution time of object oriented programs can be drastically reduced by transforming ”non escaping” objects into a collection of its component scalar data fields. But for languages that support dynamic linking, this kind of optimization (which we call ”object resolution”) can usually only be performed at runtime, when the entire program is available for analysis. In such cases, the resulting performance increases will be offset by the additional costs that arise during the analysis and restructuring phases.

In this paper, we describe work in progress, which provides an annotation technique that reduces the runtime overhead required for performing object resolutions. Our method performs a partial static escape analysis of each class at compile-time and then annotates the intermediate representation of that class with information which the just-in-time (JIT) compiler can use for object resolution. We apply this technique to the safeTSA intermediate representation, producing a simple extension to safeTSA's type system that guarantees a safe and verifiable transmission of the annotated program.

1 Introduction

Garbage collection is an important aide to programmer productivity, but it requires extra runtime overhead to manage memory allocations and deallocations. In a naive Java implementation, all objects will be created in a garbage collected heap, while local variables, containing booleans, characters, numeric types and references to

[1] Email: {krs,amme}@informatik.uni-jena.de
[2] Email: jronne@ics.uci.edu, franz@uci.edu

objects, are stored on the more efficient runtime stack. Allocating objects on the heap is less efficient, because of the extra overhead of finding free space during allocation and finding unreachable objects when the garbage collector runs.

In general, reducing the number of objects managed by a garbage collector will improve its performance. Object resolution [3] is an optimization technique that aims to decrease the garbage collector's workload by allocating as much data as possible on the stack instead of the heap. Investigations have shown that a significant number of objects behave like traditional automatic variables and can be allocated on the runtime stack. McDowell reports in [14] that in the programs in that study, between 0% and 56% of the objects could be stack allocated, but typically 5–15% of the objects could be allocated on the stack. These results demonstrate object resolution's potential as a productive optimization technique for object oriented programs.

Object resolution can be performed only for objects whose lifetime can never exceed the scope of the method in which the object is created. The objects for which this is true can be determined through an escape analysis such as that in [5]. If the escape analysis determines that the object is restricted to the method's static scope, the object's fields can be allocated on the stack[4].

Unfortunately in the presence of dynamic loading, the static object resolution of even simple applications is often infeasible. Indeed, in most cases the information necessary for object resolution is only available at execution time when the contents of external methods are available. Running a simple escape analysis algorithm for the Java Grande Benchmark programs, it turned out that most of the candidates found for object resolution cannot be optimized at compile time due f.e. unknown constructors of created objects from different classes or the access to fields of arrays where the index is not known. Deferring the analysis until runtime, however, means that the CPU cycles spent performing the escape analysis are CPU cycles that are not spent executing the program. For this reason, it is beneficial to minimize the runtime cost of the escape analysis; this can be done by performing a partial escape analysis at compile-time and attaching the results as annotations to the individual program modules.

In the paper we introduce a safe and efficient annotation technique for the transportation of escape information. Our method statically derives each class's escape information during its compilation to an intermediate representation and then annotates that representation with information that can be used by the JIT compiler in order to quickly identify and transformation non-escaping objects into their scalar counterparts. This technique is realized as an extension to the safeTSA intermediate representation[1].

The rest of the paper is structured as follows: In section 2, we discuss in more detail why a static, offline object resolution is insufficient for modern programs. Section 3 gives a brief introduction into the safeTSA intermediate representation, and section 4 describes the extensions of safeTSA's machine model required for the

[3] also called stack allocation[4]

safe transmission of escape annotations. In section 5, we discuss related work, and section 6 concludes discussing the prospects for further development.

2 Escape Analysis and Object Resolution

In a typical implementation of an object-oriented language, each object is represented in memory as a tag indicating its type, some instance fields and some fields that holds information about methods that can be invoked on that object. If an object's lifetime can be determined to end at or before the allocating method terminates (for example, if only its fields are accessed but the reference to the object is otherwise unused), field accesses to this object can be transformed into local variable accesses of the allocating method. We call this process, of transforming an object's allocation into a collection of local variable declarations and the object's fields' accesses into variable accesses, object resolution.

Escape analysis accuracy limits the effectiveness of finding opportunities for object resolution. Most escape analysis research has concentrated on the development of static whole-program escape analysis techniques. But in environments that provide pervasive separate compilation and dynamic linking (for example, the Java Virtual Machine's dynamic class loading mechanism), a compile-time static escape analysis will often be unable to identify many of the object resolution opportunities. In the presence of such dynamic class loading mechanisms, the contents of external methods cannot be determined at compile-time, so any static compile-time analysis must make worse case assumptions (that is, it must assume that each object passed to an external method will escape) [4] , resulting in an imprecise derivation of escape information and a low rate of object resolution. Despite these restrictions , most escape analyses that have been developed for Java are static compile-time analyses (for example, [7]). As a result most Java escape analyses, require that the potential user, forgot Java's dynamic class loading and separate compilation facilities.

Although, in general, object resolution of Java programs must be deferred until runtime, there a few situations in which static compile-time object resolution will be successful. In fact, there are only two such cases:

- Object resolution always can be performed statically for an array a that is defined in a method f of class c, if and only if, the number of elements of a are known at compilation time and the reference of a is never used, directly or indirectly, except by being passed as a parameter to and among other statically determined methods of c.
- Object resolution for objects o other than arrays that have been defined in a method f of class c can be performed at compilation time, if and only if, the reference to o will only be used, directly or indirectly, to access fields or by being passed as a parameter to and among other statically determined methods of c.

[4] due the Java language specification allows only one class for each classfile

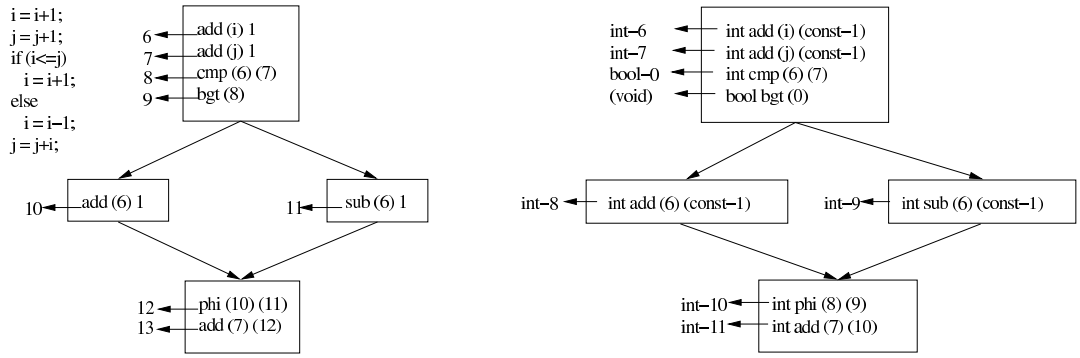
```
private class Complex{
        Complex addRandom2Complex(Complex z){
           RandomComplex r = new RandomComplex();
           z.real += r.real;
           z.imag +=r.imag;
           return z;
        }
}

(a)

class FrameCheck{
       boolean checkIntersect(FRect upper){
            FRect lower = new FRect(10,15,30,30);
            return upper.intersects(lower);
      }
}

(b)
```

Fig. 1. Programs that disallow static object resolution.

In all other cases a conducted object resolution can lead to situations in which the runtime behavior of a program could be changed and therefore object resolution should not be performed at compilation time. Figure 1 shows two example programs written in Java for which a static object resolution cannot be performed.

The method $addRandom2Complex()$ in Figure 1 (a), first generates a random complex number. It, then, adds this number to the complex number received as its parameter. Finally, it returns the result of the addition to the callers side. At first glance it appears that the object r can be transformed into two stack allocated variables $real$ and $imag$. A closer view of the program, however, exposes that the condition for static object resolution of general objects which we stated above is not fulfilled. This is because a constructor call in Java can be seen as a method call which takes an implicit reference to the allocated object as an argument, but the constructor of class $RandomComplex$ is not a member of class $Complex$. If one were to attempt to stack allocate r but then when $RandomComplex$ class was loaded, its default constructor were to, for example, append a reference to the object to a static linked list, then the reference could remain alive after the $addRandom2Complex()$ method were to exit, and the reference would now point to a location which may be reused for another purpose, violating language type safety. [5] Figure 1 (b) shows another example which is not optimizable at compile time. The class $FrameCheck$ offers a method $checkIntersects()$ to determine if a newly placed upper frame intersects an already existing lower frame. The lower frame is passed to a method called $intersects()$ that checks a possibly intersection to a given frame. As in the first example, the methods called (in this case, both the constructor $FRect()$ and the method $intersects()$) are not defined inside the class

[5] indeed static compile-time object resolution would be successful for a similar method $addComplex2Complex$, as the constructor of the analyzed class is accessible



(a) Program in SSA

(b) Type-Safe Program in safeTSA

Fig. 2. Sample program

and therefore may change at runtime.

3 SafeTSA

SafeTSA[1] is a type safe intermediate representation designed as a machine independent intermediate representation which is both trivial to verify and easy to translate into optimized machine code. The safeTSA representation is a static single assignmentvariant in that it differentiates not between variables of the original program, but only between unique definitions of these variables.

Unlike straightforward SSA representations, safeTSA provides intrinsic and tamper-proof *referential integrity* as a well-formedness property of the encoding itself. Another key idea of safeTSA is *type separation*: values of different types are kept separate in such a manner that even a hand-crafted malicious program cannot undermine type safety and concomitant memory integrity. Details can be found in [1]. The following, is a short introduction to safeTSA's type separation, which will be used in the next section to safely encode the results of partial static escape analysis.

Figure 2 (a) shows on the left side a source program fragment and on the right side a sketch of how this might look translated into SSA form. Each line in the SSA representation corresponds to an instruction that defines a new SSA variable. These variables are named by labeling the instructions with consecutive integer numbers; in this illustration, an arrow to the left of each instruction points to a label that designates the specific target variable implicitly specified by each instruction. References to previously computed values in other instructions are denoted by enclosing the label of the previous value in parentheses - in our depiction, we have used (i) and (j) as placeholders for the instructions that compute the initial values of i and j.

One problem with SSA representation lies in the type safety of a program, i.e. be assure that all operand uses are type conform. A malicious code supplier might want to provide us with an illegal program in which instructions will use operands that are non type conforming, e.g. in Figure 2 (a) the integer addition in (10) would

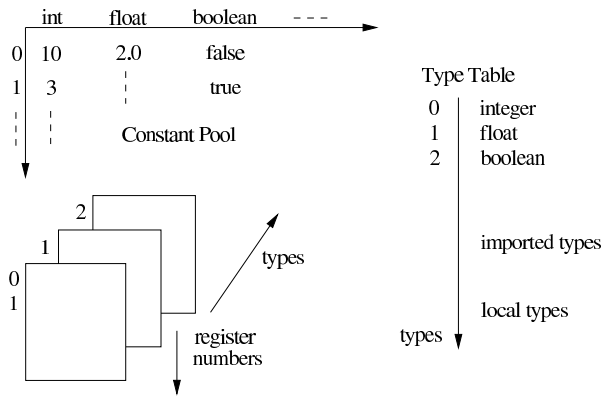


Fig. 3. Implied Machine Model of safeTSA

be applied to an boolean operand. Type safety is guaranteed in our intermediate representation by the concept of *type separation*.

While the "implied machine model" of ordinary SSA is one with an unlimited number of registers (=SSA variables), safeTSA uses a model in which there is a separate *register plane* for every type. The register planes are created implicitly, taking into account the predefined types, imported types, and local types occurring in the mobile program (Figure 3). In safeTSA the selection of the appropriate register plane is implicitly given by an operation, that is, every instruction automatically selects the appropriate plane for the source and destination registers. The operands of the instruction merely specify the particular register numbers on the thereby selected planes. Moreover, the destination register on the appropriate destination register plane is also chosen implicitly—on each plane, registers are simply filled in ascending order.

In this example, the operation *integer-addition* takes two register numbers as its parameters, *src1* and *src2*. It will implicitly fetch its two source operands from register *integer-src1*, *integer-src2*, and deposit its result in the *next available integer register*. There is no way a malicious adversary can change integer addition to operate on operands other than integers, or generate a result other than an integer. The result of applying type separation to the program of Figure 2 (a) are shown in Figure 2 (b).

Construction of Memory Safety

For the construction of memory safety our machine model provides for every reference type *ref* a matching type *safe-ref* that denotes that the variable has been null-checked. Similarly, for every array *arr* we provide a matching type *safe-index-arr* whose instances may assume only values that are index values within legal range.

Null-checking then becomes an operation that takes an explicit *ref* source type and an explicit register number on the corresponding register plane. If the check succeeds, the *ref* value is copied to an implicitly given register (the next available) on the plane of the corresponding *safe-ref* type, otherwise an exception will be

generated. Similarly, the index-check operation will take an array and the number of an integer register, check that the integer value is within bounds, and if the check succeeds, copy the integer value to the appropriate *safe-index* register plane.

Null-checking and index-checking can be generalized to include all type-cast operations: an *xupcast* operation involves a dynamic check and will cause an exception if it fails. In the case of success, it will copy the value being cast to the next available free register on the plane of the target type (only the dynamic check will result in actual code at the target machine, but not the copy operation). The *downcast* operation never fails and will never result in any actual target code.

All memory operations in safeTSA require that the storage designator is already in the *safe* state; i.e., these operations will take operands only from the register plane of a safe-ref or safe-index type, but not from the corresponding unsafe types. There are four different primitives for memory access:

getfield ref-type object field

setfield ref-type object field value

getelt array-type object index

setelt array-type object index value

where *ref-type* denotes a reference type in the type table, *object* designates a register number on the plane of the corresponding safe-ref type, *field* is a symbolic reference to a data member of *ref-type*, and *value* designates a register number on the plane corresponding to the type of *field*. Similarly, for array references, *object* designates a register on the plane of the array type that contains the array's base address and *index* designates a register on the array's safe-index plane that contains the index.

Pimitive Operations and Method Calls

Primitive operations in safeTSA are subordinate to types and there are only two generic instructions:

primitive base-type operation operand1 operand2...

xprimitive base-type operation operand1 operand2...

where *base-type* is a symbolic reference into the type table, *operation* is a symbolic reference to an operation defined on this type, and *operand1*...*operandN* designate register numbers on the respective planes corresponding to the parameter types of the operation. In each case, the result is deposited into the next available register on the plane corresponding to the result type of the operation.

Three primitives provide method invocation with and without dynamic dispatch:

xcall base-typer method operand1 operand2...

xcall-instance base-type receiver method operand1 operand2...

xdispatch base-type receiver method operand1 operand2...

where *base-type* identifies the static type of the receiver object, *receiver* designates the register number of the actual receiver object on the corresponding plane, *method* is a symbolic reference to the method being invoked, and *operand1* . . . *operandN* designate register numbers on the respective planes corresponding to the parameter types of the method. The result will be deposited into the next available register on the plane corresponding to the result type of the method.

4 Code Annotation in SafeTSA

Code annotation for supporting efficient runtime object resolution can be added to safeTSA by extending its type system. A static compile-time partial escape analysis is performed individually on each program unit and the resulting escape information is added to the unit as annotation. These annotations can be safely transported by adding an additional dimension of register planes in safeTSA's machine model and storing the escape annotations as type specializations within the safeTSA representation.

At runtime, we need to know, for each location where objects are created, whether they can outlive the method in which its memory is allocated (that is, whether the objects' definition can escape the scope of the surrounding method); if it is known that they cannot escape, then they can be allocated on the stack as part of the method's call frame. But at the time individual program units are compiled only a partial answer can be provided to the this question. Each definition of a reference to an object [6] is categorized as *normal* (that is, no guarantees are made about whether it will escape), as *strongly-bounded* (that is, it is only used for field accesses within the surrounding method or passed as a strongly bounded parameter to other methods in the same program unit, and thus does not escape the surrounding method's scope) or as *maybe-bounded* (that is, the reference is not returned by the method or copied directly to the heap, but it may be passed as a parameter to other methods from which it may escape).

Thus, the safeTSA machine model will be extended with the three escape classifications, so that for each Java class, there are six reference types (original safeTSA has two). In one dimension, each reference type is classified as either *non-null ("safe")* or *possibly-null*; in the other dimension (shown in Fig. 4, each reference is classified normal, maybe-bounded, or strongly-bounded.

In this extended model, SSA variables that stand for object references of a class A can be assigned a type $mbnd - A$ if the reference definition is maybe-bounded, but SSA variables which refer to references that may escape without restriction will be assigned the normal reference type A. In addition, parameter definition of type A can be annotated as type $mbnd - A$ or $sbnd - A$ [7] .

[6] The definitions of particular interest are the initial memory allocations, the method parameters, and ϕ-functions. References created by array accesses and returned from methods are conservatively treated as having escaped.

[7] Note, in order to preserve dynamically linking across implementation changes that effect escape properties, the caller is not required to match the callee's parameters' escape annotations exactly.

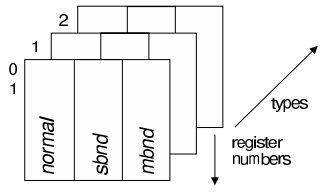


Fig. 4. Extended machine model.

The above type extensions can be used at runtime to identify the objects which can be created on the stack rather than on the heap by checking some simple constraints. Strongly-bound reference creations can always be stack allocated. (The linker must verify that strongly-bounded SSA variables are only passed to callees which annotate the corresponding parameter as strongly-bounded [8].) A maybe-bounded reference variables m can be changed into a strongly-bounded (and stack allocated), if its users are either ϕ-functions which are of or can be changed into having the appropriate strongly-bounded type and/or method calls where the corresponding formal parameter is or can be changed into the appropriate strongly-bounded type [9].

These conditions are implemented by the following procedure:

. Mark all object reference creations *a* that are of type $mbnd$.

. Trace the flow of *a* through ϕ-functions and identify all indirect users.

. Determine for each marked definition *a* the methods *f* that are passed *a* (directly or through ϕ-functions) as an argument [10].

. Check for each method *f* the type of the corresponding formal parameter *p*.
  - If for all *p*, the type of *p* is *sbnd*, the object creation may be optimized.
  - If for any *p*, the type of *p* is normal (that is not *mbnd* or *sbnd*), the object creation may not be optimized.
  - Otherwise, every *p* is *sbnd* or *mbnd* and some *p* is *mbnd*. For every *p* which is *mbnd*, check recursively if *p* is bounded. If all checks succeed the object creation may be optimized, otherwise it must be left unoptimized.

Results may be cached by changing *mbnd* parameters/variables into *sbnd* and normal parameters/variables as appropriate.

Modification of the Instruction Set

The extension of safeTSA's machine model, requires certain changes in the safeTSA instruction set. These changes, described below, are constructed in such a manner to guarantee the safe transmission of the escape information, so that the received

---

These callee's annotations are promises about what the callee will do with the method.

[8] If Java binary compatibility rules are to be observed, these will initially only be calls to final or static methods within the same compilation unit as the caller.

[9] translated, we allow maybe-bounded definitions to be used in strongly-bounded uses, not reverse.

[10] Note, due polymorphism, one call site may have multiple callees, all of which must be considered

partial escape analysis results are guarantied to be correct with respect to the received program.

The definition of a *bounded* object reference excludes it to be used as a return value. Therefore, the return instruction will automatically only select object reference variables from the normal register planes, which guarantees that no operand of type *sbnd* or *mbnd* will ever be returned from this kind of instruction.

An instruction in safeTSA implicitly takes its operand from the SSA variables in the corresponding register plane and stores its result value automatically in the next free variable in the register plane of its result type. This continues to hold for the extended machine model, but some instructions operate equally on references types of different "boundedness", these instructions require an addition modifier *n, s,* or *m* to specify on whether instructions should operate on the normal, *sbnd*, or *mbnd* register planes, respectively. Because there is not a 1:1 correspondence of boundedness between the caller's and the callee's formal method parameters, *operand-list* always denotes a list of pairs *(arg,mod)*, where *arg* stands for the argument operand and *mod* specifies which of the reference-type register planes the argument operand is to be chosen from. In the following table we describe the most important instructions that will affected by the insertion of an additional modifier.

- **new:** The *new* operator stands in safeTSA for the allocation of objects. An application of the new instruction delivers an object that is from a safe reference type, that is implicitly given by this instruction itself. The instruction will be extended by a modifier that specifies in which kind of register plane the result should be stored [11]

  new *class constructor operand-list -mod*, where $mod \in \{n, m, s\}$

- **xupcast, downcast:** Instructions xupcast and downcast copy values from one register set to an another register set. We provide only a single *mod* annotation, and require that both types be of the correct boundedness. This is sufficient to ensure that, for example, an *sbnd* variable cannot be cast to a normal variable and then allowed to escape.

  xupcast *ref-type ref-type object -mod*, where $mod \in \{n, m, s\}$
  downcast *ref-type ref-type object -mod*, where $mod \in \{n, m, s\}$

- **getfield, setfield, getelem, setelem, xdispatch, xcall-instance:**
  Instructions that can be used for memory accesses and method calls have to be extended with a modifier that specifies from which register plane the object that will accessed by the instruction, has to be taken.

  getfield *ref-type object field -mod*, where $mod \in \{n, m, s\}$
  setfield *ref-type object field value -mod*, where $mod \in \{n, m, s\}$
  getelem *array-type object index -mod*, where $mod \in \{n, m, s\}$
  setelem *array-type object index value -mod*, where $mod \in \{n, m, s\}$
  xdispatch *base-type receiver method operand-list -mod*, where $mod \in \{n, m, s\}$
  xcall-instance *base-type method operand-list -mod*, where $mod \in \{n, m, s\}$

[11] The s modifier can be used (and maintain Java binary compatibility) for calls to constructors within the same compilation unit as the caller.

```
final class FRect{
    int x, y, width, height;
    public FRect(int x, int y, int width, int height){
        this.x = x; this.y = y;
        this.width = width;
        this.height = height;
    }
    boolean intersects(FRect r){
        FrameBorder.incDefBorder(r);
        return!((r.x + r.width <= x)||
           (r.y + r.height >= y)||
           (r.x >= x + width)||
           (r.y >= y + height));
    }
}
final class FrameBorder{
    static void incDefBorder(FRect r){
        r.x -= 2; r.y -= 2;
        r.width += 2;
        r.height += 2;
    }
}
```

Fig. 5. Classes FRect (a) and FrameBorder (b)

Example of A Code Annotation

We refer back to the sample program of Figure 1 (b), as an example of how our annotation technique in action. Note that this program is not amenable to static, compile-time object resolution because either of the classes *FRect* or *FrameBorder* could be change independently after the compilation process (invalidating any static compile-time object resolution). Figure 5 shows the class definition of *FRect* and *FrameBorder* as they are loaded by the classloader of our JIT-Compiler.

During the compilation of the classes *FRect*, *FrameBorder* and *FrameCheck*, an extended safeTSA file will be generated for each class. Figure 6 depicts the methods *checkIntersect, intersects* and *incDefBorder* after compilation.

In *checkIntersect* parameter *p1* is assigned the type *sbnd* as the compile statically can verify that that reference is safely bounded. In contrast, the reference to object produced by the *new* operator will be assigned type *mbnd* as it will be passed as an argument to the method *intersects*. In method *intersects* the parameter is assigned type *mbnd* as it will be given as an argument to method *incDefBorder*, but the parameter definition in method *incDefBorder* is assigned a *sbnd* type since the reference variable created from *p1* is safely bounded to the method.

Because the method *checkIntersect* will be called during program execution and the JIT-Compiler has not already compiled this method, the classloader first will load class *FrameCheck*. Preliminary analysis of method *checkIntersect* reveals that the object referenced by variable *lower* could be a candidate for an object resolution since *lower* is of type *mbnd*. Checking if an object resolution on this object can be performed, the classloader will first load class *FRect* and verifies whether the parameter of *intersects* escapes the method. Since the parameter of *intersects* is not

```
enter method boolean checkIntersect(sbnd-FRect p1)

mbnd-FRect-0: new FRect -m
void: xcall <init> FRect (0) \$(10,n),(15,n),(30,n),(30,n) -m
sbnd-#FRect-0: xupcast FRect #FRect (p1) -s
boolean-0: xdispatch #FRect (0) intersects \$(0,m) -s
boolean-1: return boolean 0

enter method boolean intersects(mbnd-FRect p1)

void: xcall FrameBorders #class includeSmallBorders \$(p1,m)
mbnd-#FRect-0: xupcast FRect #FRect p1 -m
int-0: getfield #FRect (0) x -m
int-1: getfield #FRect (0) width -m
int-2: iadd (0) (1)
int-3: getfield #FRect #this x -s
boolean-0: ilte (2),(3)
int-4: getfield #FRect (0) y -m
int-5: getfield #FRect (0) height -m
int-6: iadd (4),(5)
int-7: getfield #FRect #this y -s
boolean-1: ilte (6),(7)
boolean-2: bbor (0) (1)
int-7: getfield #FRect #this width -s
int-8: iadd (3),(7)
boolean-3: igte (0),(8)
boolean-4: bbor (2),(3)
int-9: getfield #FRect #this height -s
int-10: iadd (7),(9)
boolean-5: igte (4),(10)
boolean-6: bbor (4) (5)
boolean-7: bbnot (6)
boolean-8: bbreturn (7)

enter method void incDefBorder(sbnd-FRect p1)

sbnd-#FRect-0: xupcast FRect #FRect p1 -s
int-0: getfield #FRect (0) x -s
int-1: isub (0),const-2
void: setfield #FRect (0) x (1) -s
int-2: getfield #FRect (0) y -s
int-3: isub (2),const-2
void: setfield #FRect (0) y (3) -s
int-4: getfield #FRect (0) width -s
int-5: iadd (4),const-2
void: setfield #FRect (0) width (5) -s
int-6: getfield #FRect (0) height -s
int-7: iadd (6),const-2
void: setfield #FRect (0) height (7) -s
```

Fig. 6. Sample methods after type extension.

defined as a *sbnd* type, class *FrameBorders* has to be loaded and the analysis will proceed with the method *incDefBorder*. The *sbnd* parameter declaration of method *incDefBorder* indicates that the parameter does not escape its method. As a result the analyzer concludes that the parameter definition of method *intersects* also is safely bound and therefore can be transformed into a *sbnd* type. This means that an object reference that is passed to *intersects* cannot escape the method and hence an object resolution on the object accessible via variable *lower* can be performed.

5 Related Work

Earliest investigations in escape analysis were done by Park and Goldberg[15], which developed an escape analysis for functional languages, which is based on reference lifetime, and operating on lists. In [18] Steensgaard has develops an interprocedural flow-insensitive points-to analysis with near-linear time-complexity. Whaley combines points-to and escape analysis into a points-to-escape graph, and uses this data structure in order to eliminate superfluous synchronization of thread-bound objects and to stack-allocate strongly-bounded objects [20]. Threads are also handled in [16], where Salcianu describes a technique for obtaining precise points-to and escape information for objects accessed by multiple threads. Vivien et al.[19] provide an algorithm for incremental pointer and alias analysis.

Choi et al.[5] applied escape analysis to perform stack allocation and reduce inter-thread synchronization utilizing connection graphs. Their method is similar to works done in the area of alias and points-to graphs, but their graphs can be more easily summarized to avoid the effort of recomputing escape information when a method is called in different scopes. Another application is given in [9]; Gay and Steensgaard describe an approach which analysis results will be used for the allocation of objects on the runtime stack. Their technique will typically detect only a subset of objects that can be stack allocated and, in addition, makes the conservative assumption that objects escape when they are stored in other objects.

Blanchet applies the results of his escape analysis to the stack allocation of objects in Java[4]. His technique can determine escape information for assignments precisely. Compared to Blanchets work, our technique benefits from a precomputed SSA form and handles dynamically loaded classes at runtime without requiring a full dynamic escape analysis. Besides escape analyses, there are other analyses that can be used to guide stack allocation or object resolution. Alias analysis[6], reference counting[11,12] and storage use analysis[17] can also be utilized in support object resolution. In general, using these analyses for object resolution will be much more expensive than using a standard escape analysis technique[4].

Code annotation for program representations have not been as thoroughly researched as escape analysis, but recently, various annotations have been proposed for enhancing the performance of JIT compilers using Java bytecode and other intermediate representations[3]. Krintz et al.[13] annotates bytecode for increasing the performance of programs executed by JVMs. Franz et al.[8] annotate programs syntax trees with escape information and encode them for safe transportation. Both

[13] and [8] guarantee the safety of their annotations. Hannan uses a functional approach for the annotation of escape information types in [10]. In contrast to our work, he excludes dynamic class loading and gives no algorithm for the computation of types.

6 Conclusion

Static object resolution is often not feasible even in very simple programs. This is because the information necessary for object resolution is often only available at execution time when the whole program is loaded. But dynamic object resolution requires that the time spent determining escape information is taken away from program execution. The runtime cost of a dynamic escape analysis can be reduced by performing a partial escape analysis at compile-time and attaching the results as annotations to the individual program units.

In this paper we have presented a safe and efficient annotation technique that reduces the runtime overhead, which is needed for performing object resolution. Our method performs a partial static escape analysis of each class at compile-time and then annotates the intermediate representation of that class with this information, which the JIT-compiler can use for object resolution. We apply this technique to the safeTSA intermediate representation, producing a simple extension to safeTSA's type system that guarantees a safe and verifiable transmission of the annotated program.

As part of a larger project, we have developed a system consisting of a Java to safeTSA compiler and a JVM extended to support just-in-time compilation from safeTSA to native PowerPC code[2]. Currently we are in the process of adding support for our our code annotation technique into the safeTSA system.

References

[1] W. Amme, N. Dalton, J. von Ronne, and M. Franz. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 137–147, 2001.

[2] W. Amme, J. von Ronne, and M. Franz. Using the safetsa representation to boost the performance of an existing java virtual machine. In *10th International Workshop on Compilers for Parallel Computers (CPC)*, 2003.

[3] A. Azevedo, A. Nicolau, and J. Hummel. An annotation-aware java virtual machine implementation. *Concurrency - Practice and Experience*, 12(6):423–444, 2000.

[4] B. Blanchet. Escape analysis for object-oriented languages: application to Java. *ACM SIGPLAN Notices*, 34(10):20–34, 1999.

[5] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for java. In *Conference on Object-Oriented*, pages 1–19, 1999.

[6] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *ACM SIGPLAN Notices*, 29(6):230–241, 1994.

[7] R. Fitzgerald, T. B. Knoblock, et al. Marmot: An optimizing compiler for Java. Microsoft Technical Report 3, Microsoft, March 2000.

[8] M. Franz, C. Krintz, V. Haldar, and C. H. Stork. Tamper proof annotations. Technical Report 02-10, Department of Information and Computer Science, University of California, Irvine, Mar. 2002.

[9] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *th International Conference on Compiler Construction (CC'2000)*, volume 1781. Springer-Verlag, 2000.

[10] J. Hannan. A type-based escape analysis for functional languages. *Journal of Functional Programming*, 8(3):239–273, May 1998.

[11] L. Hederman. *Compile-time Garbage Collection Using Reference Count Analysis*. PhD thesis, Rice University, Aug. 1988. Also Rice University Technical Report TR88–75 but, according to Rice University's technical report list, this report is no longer available for distribution.

[12] P. Hudak. A semantic model of reference counting and its abstraction. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 351–363. ACM, ACM, Aug. 1986.

[13] C. Krintz and B. Calder. Using annotation to reduce dynamic optimization time. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–167, 2001.

[14] C. McDowell. Garbage reduction in java. In *Internet Publication http://www.cse.ucsc.edu/research/embedded/pubs/gc*, 1982.

[15] Y. G. Park and B. Goldberg. Reference escape analysis: Optimizing reference counting based on the lifetime of references. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 178–189, 1991.

[16] A. Salcianu and M. C. Rinard. Pointer and escape analysis for multithreaded programs. In *Principles Practice of Parallel Programming*, pages 12–23, 2001.

[17] M. Serrano and M. Feeley. Storage use analysis and its applications. In *Proceedings of the 1fst International Conference on Functional Programming*, page 12, Philadelphie, June 1996.

[18] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[19] F. Vivien and M. C. Rinard. Incrementalized pointer and escape analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, 2001.

[20] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. *ACM SIGPLAN Notices*, 34(10):187–206, 1999.