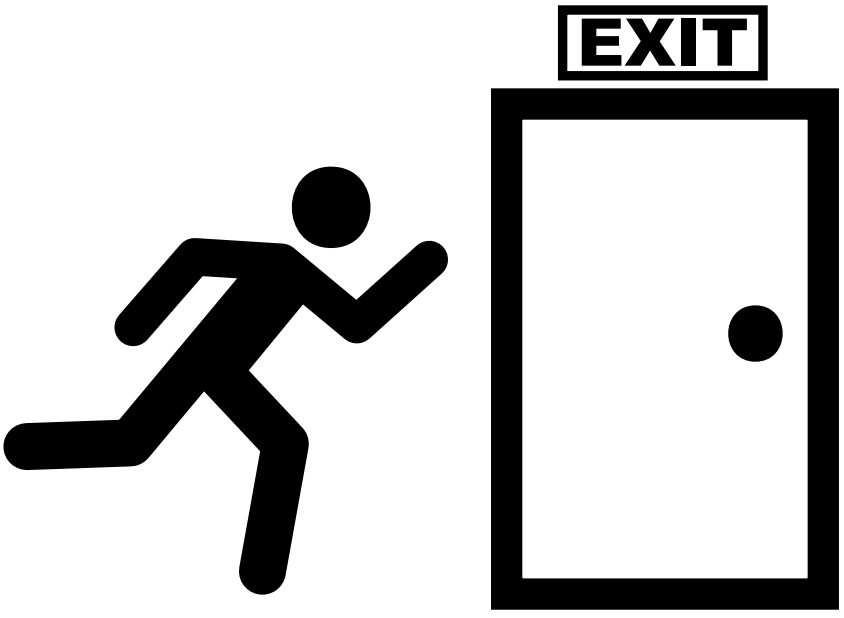
Escape Analysis in the Jikes RVM

doktor@dyregod.dk — **Ulf Holm Nielsen**
tnjr@ruc.dk — **Thomas Riisbjerg**
mads@danquah.dk — **Mads Danquah**
tkrogh@ruc.dk — **Troels Krogh**

Supervisor:

jpg@ruc.dk — **John Gallagher**

27th May 2003



Roskilde University
Department of Communication, Journalism and Computer Science
2nd module

Abstract

This paper describes how escape analysis can be used to determine whether an object has a lifetime greater than its scope. An escape analysis algorithm is implemented in the Jikes RVM's optimizing compiler. The results indicate that for some programs as many as 50% of all allocation sites do not escape the creating method. Suggestions on how this information can be used to implement stack allocation in the Jikes RVM are given.

Contents

Part I

Introduction

Foreword

This paper is part of a 2nd module semester at the Department of Communication, Journalism and Computer Science at Roskilde University. The report is made by Ulf Holm Nielsen, Thomas Riisbjerg, Mads Danquah and Troels Krogh.

The report is made with LaTeX.

An electronic version is available at: http://www.dyregod.dk/escape/

We would like thank John Gallagher for being a most helpful supervisor and Mads Rosendahl for giving us an excellent introduction to abstract interpretation, and at last the people on the Jikes RVM mailing list for their quick replies to our questions.

Readers Guide

The main topic of this paper is Escape Analysis. The reader is assumed to have basic knowledge of Computer Science. Particulary language design and program analysis.

An introduction to the following topics will be given.

- Abstract Interpretation
- Escape Analysis
- Java bytecode
- Memory management in the Jikes RVM
- Java bytecode
- The Intermediate Representation used in the Jikes RVM.
- The object model used in the Jikes RVM

This is given to ease the understanding of the implementation of escape analysis, and the suggestions on how to use it in the Jikes RVM.

The paper is divided into the following parts.

Introduction: Gives a general introduction to this paper

Analysis: Gives background knowledge needed to understand the implementation, and the suggestions on how to use it.

Implementation: Presents a Fava-implementation of the escape analysis algorithm decribed in the Analysis part. Suggestions on how the data from the analysis could be used in the Jikes RVM are given.

Analysis Results: The implementation is tested and the test-data are analyzed.

Discussion: Test results are discussed together with the suggested use of the analysis.

Lists and Citations:

Appendix: Source-code and test-results.

Chapter 1

Introduction

Ever since the Java language was introduced in 1995, it has been gaining ground among developers, even though it has often been criticized for its issues concerning program execution time. Java Enterprise Edition has nevertheless become the standard language for developing large enterprise systems in many businesses. Most universities also now have Java as inevitable part of their curriculum.

There is still room for significant improvements, even though the criticism of Java's speed has decreased in recent years as the speed of the virtual machine(VM) has improved, and the cost of computing power has continued to decline. A lot of effort have gone into making Java faster, but it still loses out as compared to languages like C and C++. In 1995 first thought would have been that Java is slow because it's interpreted. That argument no longer holds with the latest generations of just-in-time compilers and hotspot compilers.[SM02].

Many of the restrictions and features in the Java language are also in part to blame for the slower execution. This is for instance the case for bounds checking of arrays and garbage collection. In those cases it is necessary for either the compiler or the interpreter to optimize the code to eliminate as much of the overhead as possible.

In Java there is no way to programmatically control the garbage collector. This can result in garbage collection at inconvenient places in program execution. This shows as sudden pauses and increased cpu activity.

A possible way to reduce the garbage collection load is to reduce the amount of objects allocated on the heap. A C++ programmer can explicitly allocate objects on the stack and thus not be concerned about deallocating the object explicitly as it will automagically be deallocated when the current holding stack frame is popped. As a standard Java VM can only allocate objects on the heap, this is not an option for a Java programmer. Nor would this be desired as it would complicate the language and thus sacrificing the ease of use which was intended.

If Java could be made to automatically allocate some objects on the stack, it could reduce the number of objects needed to be handled by the garbage collector and then possibly reduce the negative effects of Javas garbage collection. This has been done with success in Marmot[FKR+00].

A way to identify objects suitable for allocation on the stack is by using escape analysis. Escape analysis identifies those objects that escapes the scope of the method they are allocated in.

The purpose of this paper is to study the feasibility of allocating objects on the stack in the Jikes RVM. This is done by implementing an escape analysis algorithm based on abstract interpretation in the Jikes RVM's optimizing compiler. The Jikes RVM already does escape analysis on its objects, but the extracted information is only used to determine wether an object is thread-local. An overview of the steps necessary to implement stack allocation in the Jikes RVM will be given.

Part II

Analysis

Chapter 2

Introduction to Abstract Interpretation

In this chapter the basic concepts of abstract interpretation are presented. This theoretical knowledge is necessary since the escape analysis algorithm implemented in this project is based on abstract interpretation.

Abstract interpretation is a framework for extracting various forms of information from a program by attempting to predict its behavior. This is done by transforming the program to use *abstract values* instead of the so-called *concrete values* it normally operates on, while preserving the semantics of the program. The abstract domain is highly dependent on the information one wishes to extract from the program. [Ros95]

As an example, constant-propagation will be shown within the abstract interpretation framework. A small, contrived program will be used as the basis for the analysis, shown in listing 2.1. Various aspects of abstract interpretation will be explained along the way as they are used.

The hypothetical pseudo-language follows a C-like syntax and operates on integer values for variables only. Constant Propagation analyses uses the set of possible values for each variable in the program, and identifies which variables can be replaced by constants.

2.1 The concrete and abstract domains

The first task is to define concrete and abstract domains for the interpretation. The concrete domain is defined as the set of all natural numbers, Z (ignoring that computers can only represent a limited set of numbers). The abstract domain A is defined as the set of all possible values for every variable in the program, in this case, Z again. For variables that are assigned more than one value, the element $\top$ is also included in the abstract domain, to represent "any value". Likewise, $\bot$ will represent undefined values.

It is important to differentiate among the concrete values and the abstract values, even though they both belong to Z (except $\top$ and $\bot$, which only belong to A). The abstract values represent sets of concrete values. The abstract domain is partially

ordered, and is thus structured as a lattice, shown in figure 2.1. $\top$ has a higher value than the numeric values, which in turn have higher values than the undefined element, $\bot$.

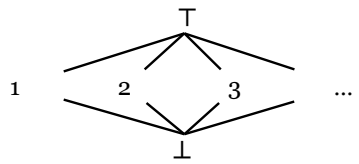


Figure 2.1: The abstract domain

2.2 The representation and abstraction functions

The representation function γ maps elements from the abstract domain onto sets in the concrete domain. In this example, γ maps elements from A to $\mathbb{P}(Z)$. Thus, $\top$ represents all (any) numbers, a specific number represents itself, and $\bot$ represents an uninitialised value (nothing.)

$\gamma : A \rightarrow \mathbb{P}(Z)$
$\gamma(\top) = Z$
$\gamma(\{x\}) = \{x\}$
$\gamma(\bot) = \varnothing$

Likewise, there exists an *abstraction function* α that maps sets of concrete elements onto the abstract domain. Given a set containing a single concrete value, α returns an abstract value representing the concrete value. If the variable may contain several values, α returns $\top$, representing any number in Z.

$\alpha : \mathbb{P}(Z) \rightarrow A$
$\alpha(\{x,y,...\}) = \top$
$\alpha(\{x\}) = x$

2.3 Soundness of the analysis

The abstraction is sound since it correctly determines variables that are only assigned one value. Any variables that might possibly be assigned more than one value are marked as such and aren't transformed in the analysis. Thus the analysis only modifies the program when it is certain that the modification is safe.

The relationship between γ and α can be expressed in that for a given statement x, the set of concrete values represented by x is contained in the concrete values represented by the abstraction of x. That is, $x \subseteq \gamma(\alpha(x))$. The only unsafe operation in constant propagation would be to mark a true variable (one that assumes many values) as a constant, replacing it with it's value. However, this is prevented since the set of values represented by that variable is *not* contained in the singleton set of the constant.

See [Ros95] for a more formal proof of the correctness of abstract interpretation.

Listing 2.1: Program before analysis

```
1 main()
2 {
3    a = 4;
4    b = 2 * a;
5    c = 20;
6    c = 3;
7    d = a + b + c;
8 }
```

2.4 Interpreting the abstract program

Given an abstract state e.g. a variable with the abstract value $\{4\}$ it is possible to define a function to evaluate a given expression in the abstract domain.

The abstract representation of the expression $a = b$ is the abstracted operation $a \approx b$. When applied to the abstract state $t = \langle a : \{4\}, b : \{3\}\rangle$, it will be updated to $\langle a : \{3,4\}, b : \{3\}\rangle$. Each concrete expression has a corresponding abstract expression resulting in an abstract value instead of a contrete value.

In interpreting the program in listing 2.1, the variable a is encountered on line 3. It is assigned the abstract value $\{4\}$, because a is assigned the literal value 4. On line 4, b is assigned to the literal value 2 multiplied by the value of a. At the current point in the analysis, a is known to hold the constant value $\{4\}$. It might seem obvious to the reader that b will be assigned the constant value $\{8\}$, however a may be reassigned at a later point in the program, and thus no longer hold a constant value.

Thus, only simple literal assignments will be considered as potential constants in this example, so whenever an expression is encountered, the receiving variable is set to the "any value" element $\top$ regardless of whether the statement only contains constants or not.

2.5 Least upper bound of abstract values

An upper bound of two elements x and y is z, such that $x \leq z$ and $y \leq z$. The *least upper bound* z is the the upper bound $z \leq z'$ for any other upper bound z' of x and y.

Line 5 in listing 2.1 assigns the abstract value $\{20\}$ to c, updating the state to $\langle a : \{4\}, b : \top, c : \{3,20\}, d : \bot\rangle$. However, line 6 adds 6 to the abstract value of c, making it $\{6,20\}$. When the representation function γ is passed an abstract value consisting of a set of more than one value, γ performs the least upper bound operation on each element of the state before mapping back to the concrete domain. Applying the least upper bound to the state $\langle a : \{4\}, b : \top, c : \{3,20\}, d : \bot\rangle$ results in the state $\langle a : \{4\}, b : \top, c : \top, d : \bot\rangle$.

2.6 Using the results

When a variable has successfully been determined to be a constant, all that remains is to transform the program accordingly. The resulting program is shown in listing 2.2, in which the variable `a` has been replaced by the literal value `4`.

Listing 2.2: Program after analysis with some variables replaced by constants

```
1 main()
2 {
3    b = 2 * 4;
4    c = 20;
5    c = 3;
6    d = 4 + b + c;
7 }
```

Chapter 3

Escape Analysis

This chapter introduces escape analysis. It contains the foundation necessary to understand the escape analysis described in chapter 5. First of a description of how objects can escape a method is given. This will then be illustrated in the last two examples

3.1 Introduction to Escape Analysis

Escape analysis attempts to identify those objects whose lifetimes exceed the scope in which they are created. Such objects are said to *escape* their scope. Broadly speaking, objects may escape their scope via assignments to non-local variables, by being passed as parameters to methods, or by being returned by a method. The details of how and when objects can escape from methods will be explained in section 3.2. The analysis also attempts to identify those objects that are only accessed by the thread in which they were created. Objects accessible from other threads than their creator are said to have *escaped* their thread. This will be explained in section 3.3.

Knowing that an object escapes or is returned from its scope or thread can lead to a number of optimizations; mainly in memory management and thread-lock elimination, as is the subject of this paper. [Rin] lists other uses for escape analysis, such as eliminating array bounds checks.

3.2 Escaping a method

The life-time of an object always starts with a statement similar to `SomeClass t = new SomeClass()` in which the variable `t` is assigned the reference to a new object. At that very point, `t` is the only reference to the object. The object can only escape the method in which it was created if another variable is assigned the value of `t`, a method is invoked with `t` as a parameter, or if `t` is returned from the method. Objects that cannot escape are not accessible from outside the method in which they were created. Such objects are also called *local* objects.

3.2.1 Escaping through assignment

An object escapes its method if it is assigned to a variable with a greater life-time than the method in which the object was defined. Such a variable could be the field of an object that escapes the method, or a static field in a class. Simple `x = y` assignments don't directly affect the escapement of an object, but it is important to propagate any escape- and return-information for a given variable backwards through its defining statements to the `new` statement that created the object. An object escapes if it is assigned to a field belonging to `this`, since the lifetime of `this` always exceeds the lifetime of any instance methods.

As shown in listing 3.1, the reference `b` escapes the method by being assigned to the static field `s` in the `C` class. Since `b` is defined as a copy of the `a` reference, `a` must also escape. Thus it is known that the object created on line 1 will escape the method.

Listing 3.1: Escaping though a static field

```
1 C a = new C();
2 C b = a;
3 C.s = b;
```

3.2.2 Escaping as a parameter

An object may escape its defining method m_1 if it is passed as a parameter to another method m_2. m_2 is analyzed, and if the given parameter is found to escape in m_2, then the object also escapes m_1. Listing 3.2 shows the variables `a` and `b` being passed to the `foo` method. Inside `foo`, the formal argument `x` escapes into the static field `s` in `C` but argument `y` does not escape. Therefore, the variable `a` escapes though `foo` and `b` does not.

Listing 3.2: Escaping though a parameter

```
1 C a = new C();
2 C b = new C();
3 foo( a, b );
4
5 void foo( C x, C y ) { C.s = x; y.f = null; }
```

3.2.3 Escaping as a return-value

Finally, the most obvious way for a an object to escape it's defining method is to be returned from the method. In listing 3.3, `a` escapes `bar` though the `return` statement.

It is possible to "catch" objects returned by a method by treating the method as a `new` statement. `foo` has access to the object created in `bar`. However, the returned object does not escape from `foo`. Some of the resulting analyses take advantage of this [GS00].

Listing 3.3: Escaping though a return statement

```
1 void foo() { C x = bar(); return; }
2 C bar() { C a = new C(); return a; }
```

3.3 Escaping a thread

An object cannot escape to another thread if it does not escape its defining method. In fact, there are only two ways an object can escape to another thread: being assigned to a static field in a class, and being assigned to another object that escapes the thread. If an object is only accessible to a single thread, there is no reason to synchronize access to the object.

3.4 Applying the analysis results

Once the escape analysis is complete, a number of objects will be known not to escape their methods. This section describes those optimizations which are the subject of this paper. For other uses of escape analysis, see [Rin].

Listing 3.4: Unanalysed program

```
1  C a( C x, C y ) {
2     C t = new C( x.f + y.f );
3     return t; }
4
5  C m() {
6     C x = new C();
7     C y = new C();
8     C z = new C();
9     C p = a( x, y );
10    C q = a( p, z );
11    return q; }
```

3.4.1 Local variables

Once an object *o* is known not to escape a method *m*, the allocation of the *o* can be moved from the heap to *m*'s stack frame.

In Java, objects are normally allocated on the heap and deallocated by a garbage collector. Allocating storage on the stack is cheaper than allocating on the heap: a stack frame is going to be created for *m* anyway; there is no need to find a slot of unallocated memory on the heap large enough to hold the object; and most importantly there is no need for the garbage collector to consider objects on the stack, since they will be deallocated automatically once the method terminates.

The `new` keyword in Java is responsible for allocating storage for the object on the heap as well as calling the object's constructor. When compiled to bytecode, the

`new` keyword from Java is substituted by a `new` instruction that allocates storage for the object on the heap, as well as calls the object's constructor to initialize the object after allocation. The keyword `newStack` is introduced to Java. `newStack` operates just like `new`, only it allocates the object on the stack instead of the heap, before calling the constructor. In practice such a keyword would be introduced on a lower level than Java, either bytecode or some intermediate form internal to the compiler.

In listing 3.4, the method `m` creates three instances of the `C` class, referenced by `x`, `y` and `z`. An analysis of the method `a` will show that its parameters do not escape, therefore neither of the references `x`, `y` or `z` escape from `m` and are suitable for stack allocation. In listing 3.5 the `new` keyword has been substituted by `newStack` as a result of the analysis.

3.4.2 Return values

When a method m_0 calls a method m_1 that returns a new object, it is possible to allocate m_1's return-value on m_0's stack frame, if the returned object does not escape m_0. A copy of m_1, m_1', is created which does not allocate a new object. Instead, m_1' accepts an additional parameter: a reference r to the object allocated on m_0's stack, which is initialized by the constructor.

In listing 3.4 the method `a` returns a new `C` object, and is marked as such during the analysis. When analysing the `m` method, the method `a` will be known to return a new object, and the variables `p` and `q` will be treated as references to new objects created by `new` statements. Since `p` does not escape from `m`, it can be allocated on `m`'s stack. In listing 3.5, a copy of the `a` method, `a2`, is created and made to accept a reference `r` to the object created in `m`. `a2` does not allocate a new object, it merely calls the `C` constructor to initialize the area pointed to by the reference `r`, denoted by `r.C()`. The call to `a` on line 9 of the original program is substituted with a call to `a2` in the transformed program in listing 3.5.

Listing 3.5: Result of analysis

```
1  C a( C x, C y ) {
2     C t = new C( x.f + y.f );
3     return t; }
4
5  void a2( C x, C y, C r ) { r.C( x.f + y.f ); }
6
7  C m() {
8     C x = newStack C();
9     C y = newStack C();
10    C z = newStack C();
11    C r = newStack C();
12    a2( x, y, r );
13    C q = a( p, z );
14    return q; }
```

3.4.3 Synchronization elimination

Given a class C with a synchronized method m_s, if an object o of class C does not escape its thread, the synchronization can be removed. This is done by substituting any occurrences of m_s with m, a copy of m_s without the synchronization. [WR99]

Chapter 4

Static Single Assignment Form

The escape analysis examined in this paper requires that the code to which it is applied is in Static Single Assignment (SSA) form. Therefore a short introduction to this form of optimization will be presented.

A program written in an imperative language may have several assignment statements, where each variable may be assign different values any number of times. An example in shown in listing 4.1.

Listing 4.1: Regular assignments

```
1 y = 4;
2 x = 1 + y;
3 z = y + x;
4 x = 2 + z;
5 s = 1 +y;
```

This form of assignment is impractical when performing certain analyses on the code such as dead code elimination and common subexpression elimination. However if the code is put on SSA form these optimizations can easily be applied. SSA form implies that each local variable can only be assigned a value once. Every time a value is assigned to a variable x a clone of x is created. Figure 4.1 shows the how a "straight-line code" block can be transformed into SSA form.

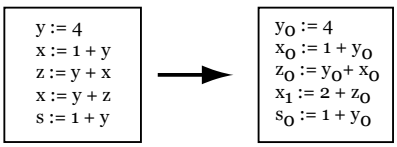


Figure 4.1: SSA tranformation

SSA form introduces the ϕ function. This function is introduced at points where the flow of control meets. It handles the assignment of variables after branches and loops. An example of an if-then-else is given in listing 4.2. Here the x can be assigned two different values, which keeps us from determining which of the x values should be assigned to z after the conditional statement is exited. This problem becomes obvious in listing 4.3 where the conditional branch is in SSA form.

Listing 4.2: Conditional branch

```
1 if (x > 0)
2    x = x*5
3 else
4    x = x/5;
5 Z = x;
```

Listing 4.3: Conditional branch in SSA form

```
1 if(x0 > 0)
2    x1 = x0*5
3 else
4    x2 = x0/5;
5 Z0 = φ(x1,x2);
```

Figure 4.2 shows the control flow graph of listing 4.2. The control flow graph on the left hand side of the figure shows how the statement branches and the control flow merges again. In the case on the left hand side it is not possible to determine what value z will assumed.

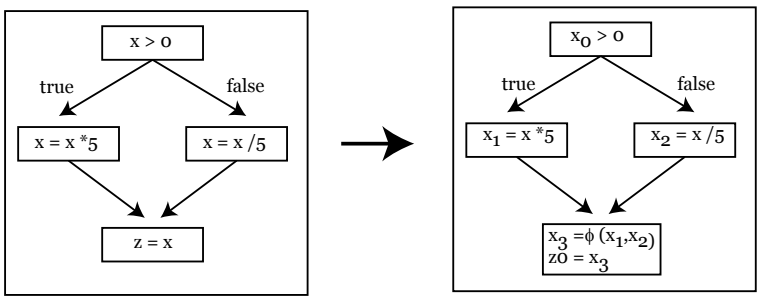


Figure 4.2: A control flow graph transform into SSA form

On the right hand side however the ϕ has been introduced and the control flow graph is now in SSA form. The ϕ will determine which of the branches where taken in the conditional statement.

For a more thourough explanation of SSA see [CFR+91].

Chapter 5

Algorithms for Escape Analysis

Chapter 3 in conjunction with chapter 2 provides the necessary knowledge to understand the escape analysis algorithm being implemented. The chapter describes a relative simple, but effective escape analysis algorithm, which will be implemented in Java. The algorithm in this chapter adds further constraints to the escape analysis explained earlier. It also introduced the concepts of properties for the methods ad variables.

5.1 Previous Work

[CGS+99] and [WR99] have both implemented escape analysis with the purpose of stack-allocating objects in Java. Both use variations of connection graphs as abstractions, that is a graph representing objects as nodes and references between them as edges. Each statement in the program updates the connection graph. A given object o is determined to escape if it is reachable from another object o' in the connection graph, and o' is known to escape. Furthermore, the implementation presented by [WR99] is capable of analyzing only parts of the program, incrementally refining the analysis as more methods are analyzed.

[GS00] presents a faster and simpler algorithm based on abstract program interpretation, which is the basis for the algorithm used in this paper. It assumes that the program has been transformed into SSA form (see chapter 4) as well as split up in Basic Blocks in a Control Flow Graph see section 7.2.1). Additionally, the algorithm does not detect objects that escape their methods but don't escape their threads.

5.2 Refinements to Escape Analysis

The algorithm from [GS00] imposes some additional limitations to the escape analysis presented earlier, specifically the algorithm doesn't consider arrays and places greater restrictions on assignments. Arrays are not considered for stack-allocation because their size often cannot be determined at compile-time. Furthermore, variables occurring on the left hand side of assignments inside loops are not considered

for stack-allocation because it is often not possible to determine the number of iterations of a loop at compile-time, potentially resulting in an infinite number of objects to be allocated on the stack.

5.3 Freshness

A local variable is defined as *fresh* if and only if its defining statement is a `new` statement or a call to a *fresh method*. A method is *fresh* if it returns a fresh variable. Listing 5.1 shows a few examples of statements and their resulting freshness. `a` is directly assigned a reference to a new instance of the class `C`, therefore `a` is a fresh variable. `b` is assigned a copy of the value of `a`, thus also referencing the fresh object created in line 1. However, since the defining statement for `b` is neither a `new` statement or a call to a fresh method, `b` is not fresh. In practice this is not an issue, as most of such assignments will have been stripped by previous optimizations in the compiler pipeline. On line 3, `c` is assigned a value from the static field `C.s`, which cannot possibly be fresh. Line 4 is more interesting; `d` is assigned the return value from the `q` method. Since `q` is a fresh method, as will be shown in the following, `d` must be a fresh variable. The `p` method returns `x`, which is a fresh variable analogous to line 1. Since a fresh method is defined as a method that returns a fresh variable, `p` must be a fresh method. Likewise, `q` returns the fresh variable returned from `p`, making `q` a fresh method as well. The final method, `r`, is not fresh, since it returns a static field in `C`.

Listing 5.1: Freshness of variables and methods

```
1 C a = new C();              fresh
2 C b = a;                  not fresh
3 C c = C.s;                not fresh
4 C d = q();                fresh
5 C p() { C x = new C(); return x; } fresh
6 C q() { C x = p(); return x; } fresh
7 C r() { return C.s; }         not fresh
```

5.4 The Abstract Domain

The abstract domain consists of tuples of four properties for each reference appearing in the program and tuples of two properties for each method.

$$\langle fresh, escape, return, loop \rangle$$

The variable properties are *fresh*, *escape*, *return* and *loop*. The *fresh* property is an element of the lattice τ, and determines if the variable represents a newly instantiated object. The lattice τ is defined as a flat lattice consisting of the complete set of Java reference types C bounded by the $\top$ and $\bot$ elements, such that $\forall c \in C : \bot < c < \top$. $\bot$ represents an undefined variable, $\top$ represents a non-fresh variable, and any value contained in C represents a fresh value of the given type. The lattice is shown below in figure 5.1.

The *escape* property for a variable is an element from the binary lattice $\{\top, \bot\}$ and determines if the object referenced by the variable escapes the current method.

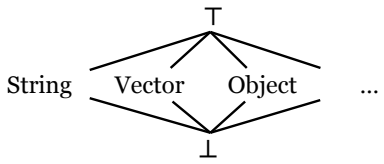


Figure 5.1: The lattice τ representing *freshness*

Likewise, the *return* property determines if the variable is used as a return value. Finally, the *loop* property determines if an assignment to the variable occurs inside a loop.

The method properties consist of a *fresh* property defined as an element in τ, analogous to the variable freshness; as well as a tuple of n values, representing the parameters accepted by the method.

$$\langle fresh, \langle p_0, p_1, ..., p_i \rangle \rangle$$

When analyzing a given method, if formal parameter i escapes, is returned or appears in a loop, the ith element in the tuple is updated to reflect this. The tuple fills the same role as the phantom-nodes in [CGS+99] in connecting arguments passed to methods with the formal parameters used inside a method, for use in inter-procedural analysis.

5.5 Constraints

Updating the abstract state is done via abstract interpretation of each abstract *effect statement*. Each type of statement adds constrains between the properties of the operands, shown below.

`return`$_m$ v:

$$\top \leq returned(v)$$

$$fresh(v) \leq fresh(m)$$

$$escaped(v) \Rightarrow \top \leq fresh(m)$$

`return` statements dictate that the *returned* property of the v operand must be at least $\top$. The method m must be at least as fresh as the return value v, keeping in mind that $\top$ is a greater value than the reference types used to represent fresh variables. Therefore, if v is non-fresh ($\top$) then m must be non-fresh as well. Finally, if the variable v escapes m, then v must not be fresh, as freshness is a requirement for identifying stack-allocable objects.

`throw` v:

$$\top \leq escaped(v)$$

`throw` statements dictate that the object thrown - v - must escape, thus *escaped* property for v must be at least $\top$.

v `= new` c:

$$c \leq \mathit{fresh}(v)$$

The result of a `new` statement is defined as being fresh, thus when instantiating an object of type c, the *fresh* property for v must at least c.

$C.s = v$:

$$\top \leq \mathit{escaped}(v)$$

Assigning the reference v to a static field s in an arbitrary class dictates that the object pointed to by v must escape.

$v = C.s$:

$$\top \leq \mathit{fresh}(v)$$

The object pointed to by a static field cannot possibly be fresh, therefore the *fresh* property of v must be at least $\top$.

$v_0.f = v_1$:

$$\top \leq \mathit{escaped}(v_1)$$

Assigning a reference v_1 to the field of an object forces the object pointed to by v_1 to escape. This is a consequence of not being able to discern assignments to `this.f` when the program is in bytecode-form; if thought of as a variable, `this` always escapes an instance method. Therefore *escaped* must be at least $\top$.

$v_0 = v_1$:

$$\mathit{escaped}(v_0) \leq \mathit{escaped}(v_1)$$

$$\mathit{returned}(v_0) \leq \mathit{returned}(v_1)$$

$$\mathit{loop}(v_0) \leq \mathit{loop}(v_1)$$

$$\top \leq \mathit{fresh}(v_0)$$

When copying a reference to an object, it is important to be able to propagate any changes to the copy (v_0) back to the original reference (v_1), such that any constrains added to the copy also affect the original. Therefore, whenever the *escaped*, *returned*, and *loop* properties for v_0 are updated, the values for v_1 must be equal to or greater than the same values for v_0. This ensures that changes to v_0 are propagated to v_1.

v_0 cannot be fresh since the defining statement for v_0 is clearly neither a `new` statement or a call to a fresh method. Any copy statement constrains the copy to being non-fresh.

$v_0 = v_1.f$:

$$\top \leq \mathit{fresh}(v)$$

As with the simple $v_0 = v_1$ assignment, the copy of the reference must not be fresh.

$v_0 = \phi(v_1, v_2, ..., v_n)$:

$$\top \leq \mathit{fresh}(v_0)$$

$$\forall i \in [1..n];$$

$$\mathit{escaped}(v_0) \leq \mathit{escaped}(v_i)$$

$$returned(v_0) \leq returned(v_i)$$

$$\top \leq loop(v_i)$$

As mentioned earlier, the algorithm assumes that the code is in SSA form. Therefore the analyzed program will contain the equivalent of ϕ-functions, to merge multiple control paths in the flow graph. If any of the parameters to the ϕ function escape or are returned from their scope, the result of the ϕ function must also escape or be returned. The *loop* parameter for the result is set to $\top$ because there is no way do discern if the ϕ function is the result of a loop or a simple branch in the program.

$v_0 = v_1.m(v_2, ..., v_n)$:

$$\forall i \in [2..n]:$$

$$\forall g \in methods - invoked(v_1, m):$$

$$let f = formal - var(g, i), c = returned(f) in$$

$$escaped(v_0) \leq_c escaped(v_i)$$

$$returned(v_0) \leq returned(v_i)$$

$$loop(v_0) \leq loop(v_i)$$

$$escaped(f) \leq escaped(v_i)$$

$$fresh(g) \leq fresh(v_0)$$

If a formal parameter i is returned from m, v_i is constrained to v_0, effectively acting as a $v_0 = v_1$ assignment. Furthermore, if a formal parameter i escapes m, then the variable passed as v_i must also escape.

5.6 Results

A local object is suitable for stack-allocation if it is fresh, doesn't escape, isn't returned and doesn't appear in a loop. The return value from a method m is stack-allocable if m is fresh. The parameter p_i escapes though a method if

Chapter 6

Introduction to the Jikes RVM

In this chapter an overview of Jikes RVM is introduced. The two compilers and the main compiler system is presented. This is explained in order to provide an insight into how the Jikes RVM works and where the optimization in form of escape analysis may be applied. Also this chapter gives a short introduction to some general aspects of the Jikes RVM.

6.1 The Jikes Research Virtual Machine (RVM)

The Jikes RVM is an open source Java virtual machine (JVM) based on the Jalapeño project[dev03a][dev03b]. The Jalopeño project was a research project initiated by IBM. In October of 2001 IBM released the source code as open source, and renamed the project to Jikes RVM. The Jikes RVM is meant as a testbed for new Virtual Machine (VM) technologies, and is capable of running most Java applications. In order to maintain its position as an open source VM, it has to utilize open source class libraries, more specifically the libraries developed in the GNU classpath project[Cla03]. Since these libraries are not completely compatible with the ones in Suns JDK 1.4, the Jikes RVM cannot run all Java-applications.

The Jikes RVM takes a new approach regarding compiling code. Instead of providing both a Java interpreter and a Just-In-Time compiler, Jikes RVM compiles every line of bytecode into native code and then invokes the code. This makes it easy to mix code from the different compilers in Jikes[BCF$^+$99].

One of the interesting features of the Jikes RVM is that it's almost completely implemented in Java. Apart from about 1000 lines of C code, which facilitate access to the operating system resources[AAB$^+$00], the rest is written in Java . But even though it is implemented in Java, it does not need a second JVM (host JVM) to enable it to run. It does however need a host JVM in order to bootstrap itself. The bootstrapping process makes the core elements of the RVM, such as the class loader, the object allocator, a compiler among others, available in a boot-image. The host JVM is needed to kickstart the bootstrapping process by running the Java program that compiles the core elements of the JVM into native code. Before this happens, the core elements have to be compiled into bytecode since the compilers in the Jikes RVM only compile bytecode. The compiled elements are then stored in the boot-image.

The Jikes RVM consists of four different subsystems.

- The runtime subsystem
- The thread and synchronization subsystem
- The memory-management subsystem
- The compiler subsystem

In this paper we are primarily concerned with the compiler subsystem and the memory-management subsystem, since these are relevant in the implementation of stack allocation. Escape analysis can also be used to optimize the thread and synchronization subsystem, but since this has already been implemented in Jikes RVM, it will not be discussed further in this text.

6.2 The Compiler Subsystem

Basically this system consists of two compilers and the *Adaptive Optimization System* (AOS). The two compilers are the *baseline* compiler and the *optimizing* compiler. If the AOS is enabled the code will be optimized, while it is running. Each method may be compiled several times, at different levels of optimization. Without the AOS each method is compiled once, using the specified compiler.

6.2.1 The Optimizing Compiler

The optimizing compiler is by far the most interesting compiler in the Jikes RVM. It yields highly optimized code, but the compilation process is also slower. It works by translating bytecode into an Intermediate Representation(IR) (see section 7), upon which it can perform various forms of optimizations.

The optimizations done by the optimizing compiler can be divided into three groups: local, global and inter-procedural. Local optimizations are optimizations inside a single extended basic block (see section 7). Global optimizations are done on a method level, spanning several basic blocks. Inter-procedural Optimizations are done across several methods.

Optimizations that fall under the local category are local common-subexpression elimination and local constant propagation. Both are flow sensitive analysis and therefore benefit from having the control flow graph given (see 7.2).

Global optimizations include live variable analysis and a range of optimizations done on IR in SSA form. Among them are global common subexpression elimination, redundant load elemination, redundant branch elimination, global value numbering and simple escape analysis to avoid synchronization penalties if a variable does not escape the current thread.

Inlining of methods and method specialization are done as part of the inter-procedural analysis.

The Jikes RVM divides these optimization into three different levels of optimizations labelled `O0`, `O1` and `O2`. If the optimizing compiler is used outside of the AOS the user can specify which level of optimization to use, otherwise the AOS will decide which of the optimization levels to apply.

6.2.2 The Baseline Compiler

The baseline compiler is a very fast compiler, that produces unoptimized code. In contrast to the optimizing compiler the baseline compiler does not generate an intermediate representation of the Java code. Instead it translates the Java code directly into native code, by simulating the operand stack in Java[LY99]. The result is a speedy compilation, which generates poorly performing code. The compiler was primarily used during the development of Jikes RVM[AAB+00], but is still an active part in the Jikes RVM.

6.2.3 The Adaptive Optimization System

The AOS is comprised of the following three components.

- The runtime measurement system
- The controller
- The recompilation system

In addition to these components the system contains a database, which holds information from all the components. The database creates static objects, that hold information on the sampled data.

Figure 6.1 shows an overview of the AOS.

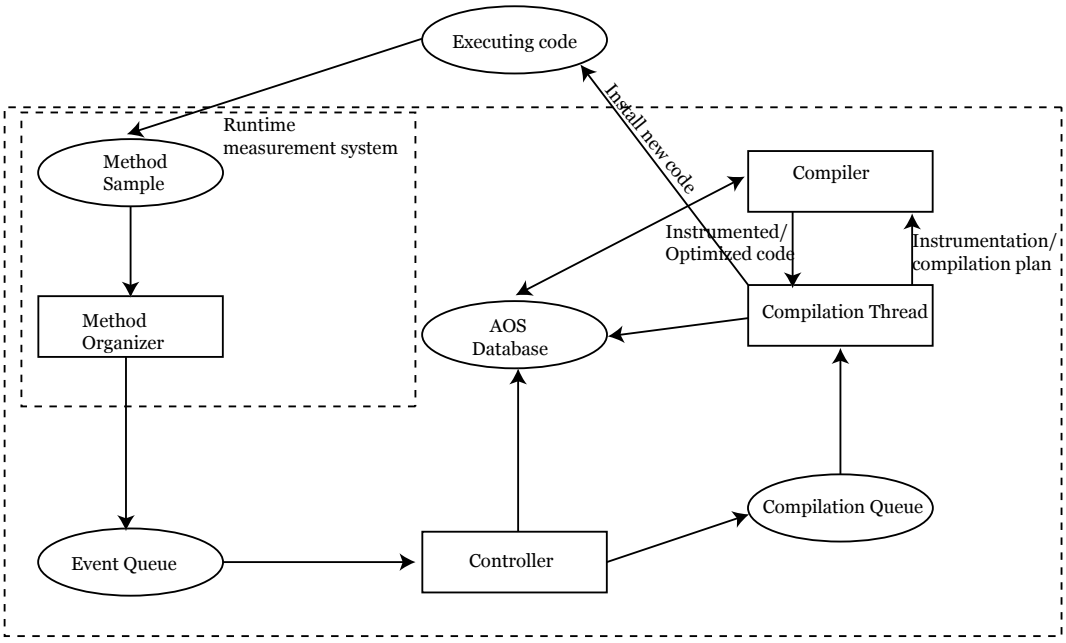


Figure 6.1: Crude overview of the adaptive optimization system

When the Jikes RVM is booted a call is made to the *boot* method in `VM_Controller`. This method takes care of initializing the AOS. The *boot* method starts by checking for command parameters and then moves on to initializing the rest of the components and the database.

The Runtime Measurement System

As mentioned the AOS optimizes the code while it is running. In order to determine whether a piece of code should be optimized or not , information about the running code is needed. This is provided by the Runtime Measurement System (RMS). This system monitors the code as it executes and gathers information from hardware and software monitors. Information such as method invocation counters and call-graph edge counters are fed into *organizer*-threads. The organizer threads analyze the data from the monitors and pass the results on the controller. The method invocation counter keeps track of the number of times a method has been invoked, while the call-graph edge counter keeps track of the number of times a method has been called from a given method. The former information can be used by the controller in deciding if a method should be recompiled with a higher optimization level. The latter aids in deciding whether or not a method should be inlined.

The Controller

The controller retrieves information from the RMS and/or the database. Based on this information the controller decides how the system should react. If it decides no action is to be taken, it tells the RMS to keep analysing data from the running program. Alternatively it can choose to recompile certain methods to improve the performance of the application. If the controller decides to recompile a method, it is added to a queue of methods waiting to be completed.

The controller bases it decisions to recompile on a cost/benefit analysis. It calculates the future running time (T_f) of the program, simply by looking at the current running time of the program. So if the program has been running for 5 minutes, then T_f will be equal to 5[AFG$^+$00]. It also calculates the percentage of time the program will spend in a method (P_m), by looking at the sample data. From these two numbers, the time spent in a method in the future is calculated (T_i).

$$T_i = T_f * P_m$$

The controller then has to predict if the method will perform better if recompiled. This can be done by using the following equation.

$$T_j = T_i * S_i/S_j$$

The above equation yields the future expected time spent in a method, where S_i/S_j represents the speedup ratio between the current level “i” and the new optimization level “j”. These constants are based on offline reference measurements, taken by the Jikes RVM development team.

In order to finish the analysis, the compilation time cost (C_j) is added to T_j. C_j is a constant based on the size of the method. If $T_j + C_j < T_i$ the method will be placed on the compilation queue.

Along with every method the controller outputs a compilation plan containing an optimization plan, profiling data and an instrumentation plan. The optimization plan tells the compiler which optimizations to apply to the method. The profiling data is the data collected by the RMS. This data is used when the compiler performs adaptive inlining. The instrumentation plan is used to insert further monitoring code into the method.

The Recompilation System

When elements are placed in the compilation queue the recompilation system will be invoked. The requested method will be recompiled in a compilation thread, which means that the application and recompilation can run simultaneously.

The new code generated from the compilation process is then inserted into the program. This monitoring and recompilation is an ongoing process that continues throughout the lifespan of the program.[AFG+00].

Chapter 7

Intermediate Representation of Code

This chapter will introduce the three kinds of intermediate code levels that the Jikes RVM uses. The focus will be on on the HIR, since the escape analysis is applied to this level of code. The HIR resembles bytecode but several optimization can be applied to it, among others the SSA form.

How they are represented, constructed and what optimizations are done on the code as it progresses through the three levels.

7.1 Intermediate Representation

The Jikes RVM's optimizing compiler operates with three levels of intermediate representation of code (IR): HIR (High level Intermediate Representation), LIR (Low level Intermediate Representation) and MIR (Machine specific Intermediate Representation).

The IR is register based as opposed to Java bytecode which is stack based. All three levels of IR has an unlimited number of symbolic registers available. IR code is register based because a register based form provides greater flexibility in code transformation and is closer to the target architecture and therefore enables optimizations for these architectures[AAB+00].

7.1.1 Instruction Format

IR instructions consist of an *operator* and zero or more *operands*. The operator represents the instruction to execute and the operands can represent registers (both symbolic and physical), memory locations, types, labels, guards etc. This is true for all three levels of IR.

Listing 7.1: A single statement in HIR

```
1 2  EG new                t9si(Ljava/lang/Object;,p) = java.
     lang.Object
```

The first field on each line of IR code is the index in the class file that corresponds to this line. The statement in listing 7.1 corresponds to index 2 in the class file. It is followed by a field that can be either empty, a label or an indication if this line can produce an exception (E) or yield to the garbage collector (G). An object allocation via the new instruction may both cause an exception and yield to the garbage collector.

If the current line is a label, the next field contains a frequency indicator used to predict control flow. Otherwise the field contains the operands.

References in IR are always preceded by a type identifier. The following list is all indentifiers used and their meaning.

B - byte
C - char
F - float
I - int
L - long
L<classname>; - reference to classname
S - short
Z - boolean
[- reference to array dimension

The Jikes RVM uses several kinds of registers. These are indentified by a sequence of letters and numbers. `l0i` denotes the local register `0` of type `i`(integer). The first letter can be `l` or `t` and indicates if the register is local or temporary. The last letter indicates the type of the register, it can be either:

i - integer
c - condition
d - double
f - float
l - long
v - validation

The number in between denotes the register number. Additionally the number can be followed by both a `p` and an `s`. `p` denotes that the register spans more than one basic block and `s` that the register is in SSA form.

7.1.2 Step by Step example

As an example consider a small Java program that simply prints “Hello Jikes!”. The main method is shown in listing 7.2 as bytecode.

Listing 7.2: Simple class’ main method as bytecode

```
1 Method void main(java.lang.String[])
2   0 getstatic #12 <Field java.io.PrintStream out>
3   3 ldc #15 <String "Hello Jikes!">
4   5 invokevirtual #20 <Method void println(java.lang.
      String)>
5   8 return
```

Line 2 gets the static `out` field from the `System` class and puts it on the stack. Then get the string constant "Hello Jikes!" and put it on the stack, this will be the argument for out method call. Finally invoke the method `println` on the `out` object with the string as argument, then return.

The first column in every line is the index of the current opcode in the class file, next is the opcode followed by the operands and an optional comment surrounded by < and >. The comment usually indicates what class or method the operation is working with. In order for `ldc #15` to be of any meaning one would need to know that index 15 in the constant pool contains the string "Hello Jikes!".

Listing 7.3: Simple class as HIR

```
1 -13 LABEL0  Frequency: 0.0
2 -2  EG      ir_prologue l0i([Ljava.lang.String;,d) =
3 0           getstatic  t1i(java.io.PrintStream) = <mem loc
    :java.lang.System.out>
4 5   EG      null_check t2v(GUARD) = t1i(java.io.
    PrintStream)
5 5   EG      call       AF CF OF PF SF ZF = <unused>,
    virtual"java.io.PrintStream.println (Ljava/lang/String
    ;)V", t2v(GUARD), t1i(java.io.PrintStream), "Hello
    Jikes!"
6 -3  return  <unused>
7 -1  bbend   BB0 (ENTRY)
```

The HIR code in listing 7.3 is slightly more complicated than the bytecode. Line 1 defines the label, the first field of line 1 indicates by a negative number that this is not part of the original bytecode. The exact number is determined by which part of Jikes inserted the code.

Line 2 is still not part of the original bytecode. The `ir_prologue` instruction sets up the call stack prior to executing a method including loading references to the method parameters. In this case loading a reference to parameter into the local register 0. The parameters are of type String array, as can be deduced from the string; `([Ljava.lang.String;,d)`. The `[` indicates a one dimensional array of the type indicated by `L`, `String`. The `,d` determines if the variable is either:

x - extant
d - declared type
p - precise type
+ - positive int

The statement in line 3, `getstatic`, corresponds directly to line 2 in the bytecode. Register `t1i` is a reference of type `java.io.PrintStream`. It is set to the content of the the memory location resolved by `<mem loc:java.lang.System.out>`.

The instruction at line 4 checks if the contents of the `t1i` register prior to the call instruction in line 5 is `null`. The result of the check is stored in `t2v` which is of type is `GUARD`. If `t1i` is `null` a null pointer exception is thrown.

Line 5 calls the virtual method `println` on the type `PrintStream` with argument of type `String` and return type `V`, for void. The call is made only if allowed by the guard `t2v`. If allowed `println` is invoked on the object referred to by register `t1i` with the parameter "Hello Jikes!". This is similar to invokevirtual in bytecode,

except when the null check is explicit. The letter combinations in the beginning, AF, CF etc, describe which EFLAGS can be set on the IA32 architecture[Bre03].

Line 6 contains a return instruction with no operands, thus returning void. And at last an indicator showing that this is the end of the basic block.

For further information on the IR instruction set please refer to the source code in rvm/src/vm/compilers/optimizing/ir dir of the Jikes RVM source distribution[dev03c].

7.2 Construction of the Intermediate Representation

7.2.1 Control Flow Graph and Basic Blocks

A program in IR form consists of a number of basic blocks, each representing a "straight" part of the program with no jumps in or out in the middle of the block (e.g. a basic block can be a part of a method from start until a condition, method call or throw operation).

Instructions that can end a basic block are jumps (conditional or not) or instructions that might throw an exception. Instructions that may throw an exception— are called PEI(Potential Exception-throwing Instruction) in Jikes RVM. In IR this is denoted by an E before the operator.

Figure 7.1 show a simple program in a simple IR divided into basic blocks. The `call_method` instruction can cause an exception to be thrown and can be seen as a conditional branch.

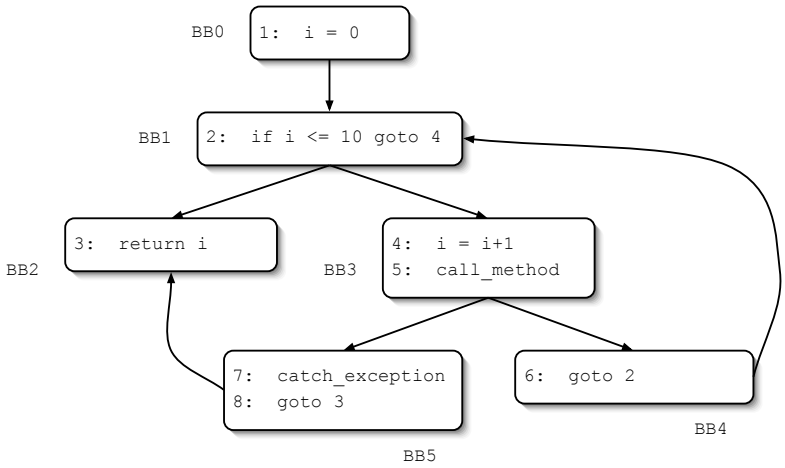


Figure 7.1: Basic blocks

The basic blocks form a control flow graph with the basic blocks as nodes and edges representing the control flow. Basic blocks can be found by determining *leaders*[Kri03]. A leader is the first instruction of a method, or an instruction `I` if in the program exists a statement `goto I`. The instruction immediatly after a `goto` is also a leader. Following this definition a basic block is a leader and all the instructions following it until the next leader.

Finding leaders

The simple IR from the previous example is shown in listing 7.4 on page 38. A leader is[Kri03]:

1. The first line of a method is a leader
2. Line L is a leader if there is a tuple that jumps to L
3. Line L is a leader if the previous line contains a jump

Thus it is determined that line 1 is a leader according to 1. Lines 2, 3, 4 and 7 are leaders according to 2. And line 6 is a leader according to 3 and since any instruction that might cause an exception to be thrown is also a conditional jump to the catch block.

Listing 7.4: Simple IR for a method

```
1 int i = 0
2 if i <= 10 goto 4
3 return i
4 i = i+1
5 call_method
6 goto 2
7 catch exception
8 goto 3
```

7.2.2 Factored Control Flow Graph and Extended Basic Blocks

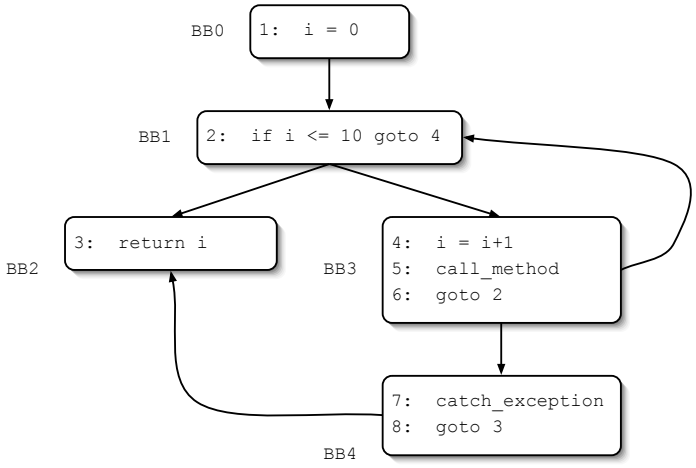


Figure 7.2: Extended basic blocks

Though the control flow graph seems nice, the basic blocks tend to become rather small, thus generating a large amount of nodes and edges.

Jikes RVM operates with the notion of *extended basic blocks* in which method calls and throw calls do not end a block. An extended basic block has a single entry point, one normal exit point, and can have several exits from throw operations. Extended basic blocks form a *factored control flow graph*[CGHS99].

The reason for using factored control flow graph is that for most analysis the possible exit points due to exceptions do not complicate the analysis and will enable some optimizations normally not feasible on basic blocks. Furthermore it decreases the number of nodes and edges in the control flow graph significantly thus making analysis faster.

The factored control flow graph for the example in listing 7.4 would look like in figure 7.2.

Notice that `call_method` no longer triggers the contruction of a new basic block.

For a more thourough discussion of factored control flow graphs see [CGHS99].

7.2.3 From Bytecode to Intermediate Representation

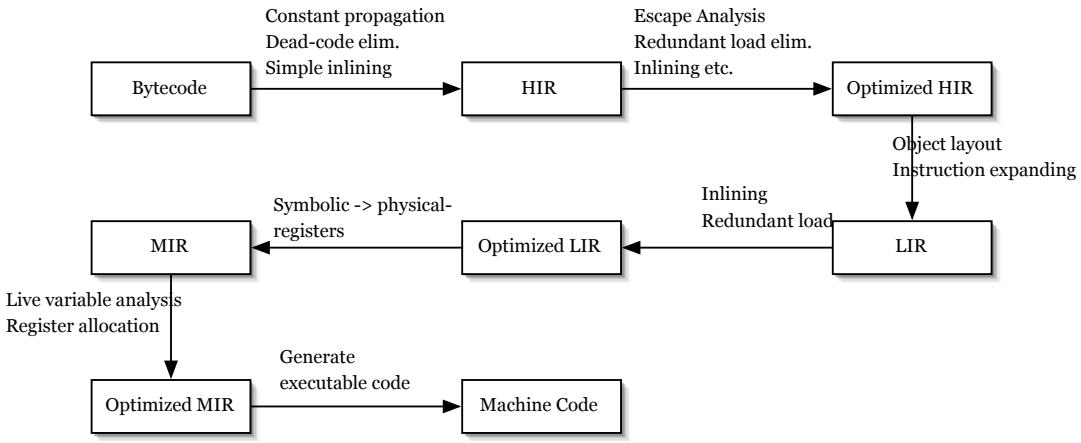


Figure 7.3: BC to HIR to LIR to MIR to MC

The HIR is constructed by abstract interpretation of the Java bytecode, with a few optimizations done at the same time; among them constant propagation and dead-code elimination.

When the bytecode has been translated to HIR all optimizations suitable for HIR is performed. Then HIR is converted to LIR, then optimization for LIR is applied and finally the code is translated to MIR, and after applying optimizations the code is compiled to machine code. A graphical overview is given in figure 7.3

LIR can be up to three times larger than HIR as instruction such as `new` is expanded with explicit calls to allocaters and specific instructions to get information in the object model.

MIR is closely tied to the target architecture and is thus different from IA32 and PPC. MIR code is even larger than LIR and is not suitable for reading.

7.3 Noteworthy Instructions

HIR has a special instruction for dealing with SSA form. The `phi` instruction, that based on which basic block (execution path) was used to reach the instruction can determine which register of two to move to the target register. `phi`

`<target>,<register from path1>,<register from path2>`. See chapter 4.

Chapter 8

Runtime data-organisation in Jikes

This chapter will give an overview of how the Jikes RVM organizes its objects, and how it manages its heap and stack. Allocation and deallocation of objects will be described, and a short introduction to implicit deallocation (garbage collection) will be given.

This knowledge will later be used to describe how method-local objects can be allocated on the stack.

8.1 The Object Model

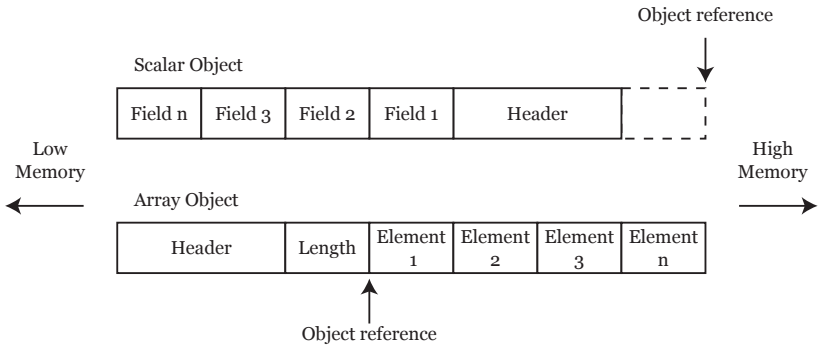


Figure 8.1: A scalar and an array object

An object in the Jikes RVM can either be a scalar-object containing fields, or an array-object containing elements[RVM03]. Scalar objects are laid out from high memory to low, and array-objects are laid out from low memory to high (see figure 8.1). Array elements are accessed with a negative offset from the object-reference. The reason this is done is that this layout gives free null-pointer checks via hardware-traps on the AIX-platform [AAB$^+$00]. The object header is placed at a fixed negative offset off the object reference, This guarantees that an object can be identified without knowing if it is an array or a scalar object.

Each object in the Jikes RVM includes a number of headers, containing information concerning different part of the Jikes RVM, such as locking information, reference

counts and different status bits. One of the key elements in the header is a reference to the Type Information Block (TIB) associated with the objects class.

The TIB is a set of object references that defines the type of a class [AAB+00]. The TIB contains references to the superclass (if any) of the class, fields of the class, and references to compiled versions of the class's methods. A compiled method is stored as an array of machine instructions.

Listing 8.1: A couple of static declarations

```
1 class A { static int i = 1337; }
2 class B { static String s = "asdf"; }
3 class C { static void m(){} }
```

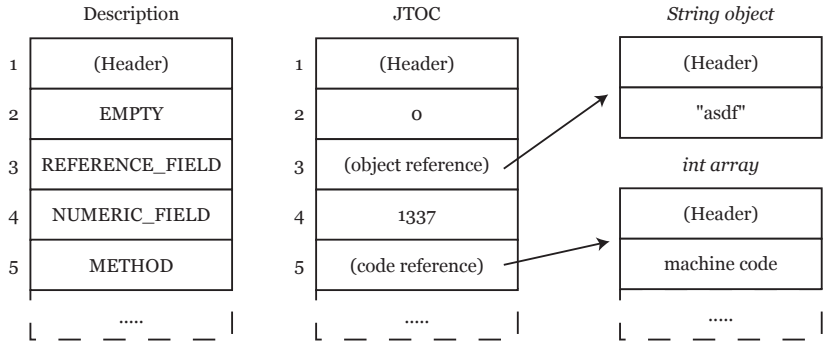


Figure 8.2: Resulting entries in JTOC

All global accessible data are available through the Jikes RVM Table Of Content (JTOC). The JTOC is declared as an array of ints, but is really an array of mixed types. Each value in the JTOC is either a reference or the int-value of a primitive type. To keep track of the different types in the JTOC, Jikes maintains another array co-indexed with the JTOC. This array describes the type of each value in the JTOC.

JTOC contains references to all static fields, all static methods and all TIBs. Primitive fields are stored directly into the JTOC. The JTOC is in other words for Jikes what the constant pool is for Java. An example of how the JTOC will look after a couple of static declarations is depicted on figure 8.2 and the sample code is listed in listing 8.1

8.2 The Memory Model

The memory is divided into the heap and the stack. The stack grows from low memory to high, and the heap from high to low. All objects are allocated on the heap.

Conceptually the memory can be seen as two different areas, the program-area and the Data-area [GBJL00]. The program-area contains the executable part of the data, and the data-area contains the remaining data. There is a long way from this abstraction to how the memory is organized in reality. The heap contains most of the program data (JTOC and compiled methods), since these are kept in an array stored on the heap.

8.2.1 The Stack

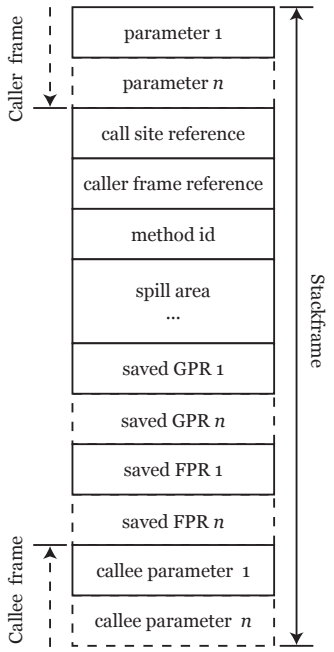


Figure 8.3: Layout for a stackframe (Intel version)

The stack is a stack of frames, one frame for each method that has been invoked but not yet completed. When a method invocation occurs, a new stackframe is pushed on the stack. A stackframe is generated for each method invocation; the instruction that causes the invocation is called the *call site*. During execution of a method, the state of the registers will change. Some registers are dedicated, while others can be used as the compiler wishes. These are the General Purpose Registers (GPR) and Floating Point Registers (FPR). This means that when a method invocation occurs, the state of the GPRs and FPRs at the call-site must to be saved, in order to guarantee that they return to their "original" state when the invoked method returns to the call-site.

When a method returns, its stackframe is popped from the stack. The stackframe-reference is moved back to the caller frame, and the instruction reference is moved back to the call-site. The method's return-values (if any), are stored in a dedicated register, which can be read by the caller frame.

Before a method is executed, its *prologue* is executed. The prologue is responsible for updating frame-references, call-site references, and saving the needed registers. After a method is executed, its *epilogue* is executed. The epilogue restores the saved registers, and updates the frame- and call-site- references.

The frame contains:

- References to method-arguments (shared with the calling frame.)
- Reference to the call site (an instruction reference)
- Reference to caller's frame
- ID of the method
- A spill area that can be to store local data than cannot be fit in registers.
- Value of registers from caller (General Purpose Registers and Floating Point Registers)

– Parametres to callee frame

The layout of a stackframe is depicted on figure 8.3.

The stack is managed by a stack manager. The stack manager controls all manipulation with the stack. It also pushes, and poppes stackframes when requested to. The stack is formally declared as an array of ints (but like the JTOC, it is really an array of mixed types). Each entry in the stack-array, is called a slot. Each slot can either contain a reference to an object (such as an parameter to the method), or the integer value of a primitive. As such any value can be stored on the stack, the stack manager contains methods to allocate a given size of data. So one could for instance allocate enough space to store an entire object on the stack.

8.2.2 The Heap

The Jikes RVM supports a number of memory managers for management of the heap. A memory manager provides an object allocator, and a garbage collector. The choice of allocator and collector is represented in a *plan*. Each time an object needs to be allocated, or an garbage collection is trigged, the plan is consulted. The correct collector/allocater is found, and the requested operation is preformed.

Different memory managers use different methods to allocate and deallocate objects from the heap, but they all provide a mechanism to put (allocate) objects onto the heap, and remove (deallocate) them again. They all use a small-object heap, and large-object heap. This enables the manager to handle allocation and deallocation of large and small objects differently[AAB+00]. The performance penalty for copying small objects is smaller than for large objects.

Threads in the Jikes RVM are multiplexed onto a number of *virtual processors*[AAB+00]. Each virtual processor has a garbage collector and a local memory-space associated with it. The processors share a memory resource from which they request memory-block, but all access to this shared resource must be synchronized, and is therefore expensive. To reduce this overhead, each processor keeps a processor "local" memory-space. The local memory-spaces are not restricted, processors can freely address each others memory-space, but only the owner of the local memory space can allocate and deallocate from it.

8.2.3 Heap Allocation

An allocator's job is to allocate space for an object on the heap. When fed with information about the type of object (scalar or array), and size, it will find a free block of memory, and allocate it to the object.

The memory manager divides the heap up into a large and small-object space. Allocation from memory that is shared between the virtual processors require synchronization. To reduce the synchronization overhead, each virtual processor keeps a processor-local space used for allocation of small objects. This space does not require synchronization, and all small-object allocations are done here. When a processor local-space runs out, it requests a new from the shared pool.

When an allocator runs out of free space, it triggers a garbage collection, that hopefully will result in enough free space to handle the request. If an allocator

fails to allocate the requested amount of memory after a garbage collection, the system runs out of memory.

8.2.4 Object Creation

The creation of an object is divided into two phases: *allocation* and *initialization*. In the allocation phase, the object's class is examined and the amount of memory needed to store the object is determined. An allocator is queried for the needed memory, and if the allocator succeeds it returns a reference to the memory-block. An simple illustration of this is depicted on figure 8.4.

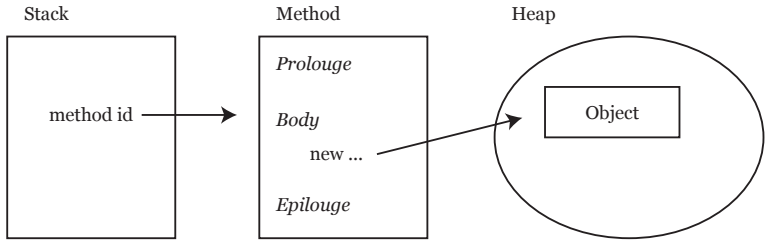


Figure 8.4: References to objects point into the heap

In the initialization phase, the object's header is created, and a reference to the TIB is created. If the object is an array object, the length is stored in the header as well. When the object has been prepared, the object's constructor is invoked.

Objects in Java are allocated with the `new` keyword. On this level no distinction is made between object allocation and initialization.

Listing 8.2: Allocation of a String object in Java

```
1 String s = new String();
```

After this line `s` will be a reference to the newly created `String`-object.

On the bytecode level the new operator allocates memory for a new object. Object initialization must be done explicit by invoking the object-constructor.

Listing 8.3: Allocation of a String object in bytecode

```
1   0 new #2 <Class java.lang.String>
2   3 dup
3   4 invokespecial #3 <Method java.lang.String()>
```

Each instruction explained:

1. Allocate space for an new `String` object, and put the reference to the new object on the operand stack.
2. Duplicate the reference to the object
3. Invoke String objects constructor (on the reference)

The HIR-level also distinguishes between object allocation and initialization. The `new` operator allocates memory for a new object. The object are initialized by calling the objects constructor.

Listing 8.4: Allocation of a String object in HIR

```
5  EG new    t4si(Ljava/lang/String;,p) = java.lang.String
9  EG call   AF CF OF PF SF ZF = 696, special_exact"java.lang.
   String.<init> ()V", <TRUEGUARD>,
t4si(Ljava/lang/String;,p)
```

These instructions boil down to

1. Allocate space for an new String object and store the reference to it in register t4si
2. Invoke the String objects constructor (java.lang.string.<init>)

The main difference is that the reference now is stored in a register, and not on a stack, as HIR is written for a register-based machine.

8.2.5 Heap Deallocation (Garbage Collection)

Java uses implicit-deallocation. This relieves the programmer from the tedious task of de-allocating data when it is no longer used. This is instead done by the garbage collector.

A garbage collector's job is to remove garbage from the data-area. Garbage is data that are no longer referred from the program-area. Garbage is found by locating all data in the heap that is directly or indirectly referred to from the program area. The remaining data can safely be regarded as garbage. The referred data can be located, by scanning the program-area for references. Each reference is followed, and the referred block of data is marked as being alive. Each found block of data, is then again scanned for references, which again are followed. This process is repeated, until all referred data have been found.[GBJL00]

The garbage collector works from a *root set*. The root set is the set of references that, when followed resolves to all live objects in the data-area. The root set at a given point in Jikes is the union of all references in the stack and JTOC.[GBJL00] [AAB+00].

At every point in the code that can trigger a garbage collection, the compiler generates a garbage collection map (GC-map)[RVM03]. This map describes all locations in the stack that currently contain references. The GC-map combined with the references in the JTOC are the root set for the garbage collector.[AAB+00]

When a garbage collection is triggered, thread switching is disabled on the virtual processors, and each processor's garbage collector is started. Each garbage collector is responsible for marking live objects in its virtual processor's local memory-area. At certain points during the collection, the garbage collectors must synchronize to perform tasks that cannot be performed in parallel. Marking live objects can for instance be performed in parallel, while deallocation of the individual object can only be done by a single garbage collector. When the garbage collectors have completed the collection, thread switching is re-enabled, and life goes on.

Part III

Implementation

Chapter 9

Implementation of the escape analysis

This chapter will describe how the escape analysis is implemented and applied to the HIR code. The description will not cover the code in detail, but will unveil the general principals of the escape analysis.

9.1 Overview of the Implementation

The escape analysis algorithm from chapter 5 has been implemented in the Jikes RVM's optimizing compiler, as part of the compiler pipeline just after the code has been converted to SSA form. It is implemented as a standard compilation phase and as such can be enabled and disabled just like any other Jikes RVM optimization.

The analysis is done on HIR code, and does not support neither LIR or MIR. HIR was chosen as the target as it still contains object allocation as a single instruction whereas LIR and MIR expand these to several instructions.

The implementation includes a switch to enable inter-procedural analysis(IPA). To enable IPA just give the option `escape_ipa=true` to the optimizing compiler.

The implementation also supports printing the HIR code prior to analysis.

9.2 Algorithm Implementation Details

Central to the analysis is a method looping through all of the statements in a method in HIR form. The code is assumed to be in SSA form, though no check to ensure this is made.

Inside the loop a switch statement delegates control to each of the effect statement's handler methods. That is, there is a method for handling `new` statements and one for method calls.

9.2.1 Effect Statements

Each register encountered in an effect statement is stored in a summary object pr method, where each register has values for each property: fresh, escaped, returned, and loop. These summary objects are themselves stores in a central summary database for the entire VM.

Depending on the effect statement properties may be modified and contraints may be implied on these. For instance a register assigned a reference to an new object allocation should be set to fresh.

Most effect statements in HIR correspond directly to the Java counterpart described in section 5. This is true for return, throw, assignment to a field, assignment to a static field, new and method calls.

Phi functions are present in HIR as actual instructions; only later converted to regular ref_move instructions at the end each control flow branch. This makes it easy to determine the registers involved in loops or complex conditional branches in a program.

An assignment from on local variable to another local variable in Java is equivalent of a ref_move in HIR. In such case, if the target's properties change, the changes should be reflected in the trigger as they are essentially pointing to the same object, and naturally there is only one object to allocate on the stack so the target can never be fresh.

9.2.2 Modelling Constraints

To reflect changes to the target register in the source register, a dependency is added between the properties of the target register and the source register. A seperat list of dependencies is maintained for each register along with the properties. The dependency list associated is then evaluated when the register changes any of it's properties. A dependency is removed if it's target reaches the top element of it's lattice ($\leq \top$). The dependency is more or less an expression of the upper bound between two properties.

Contrary to the theoretical walkthrough of the algorithm, the implementation models the freshness lattice τ as a binary lattice because HIR does not declare variables as in Java source: A register is only introduced if it is assigned a value. So a register can never be uninitialized. This simplifies the upper bound operation to a simple logical OR.

9.2.3 Inter-Procedural Analysis

IPA is achieved by keeping track of a methods parameters in the same way as any other register in the analysis. This information is stored in the method summary as well. This is used to determine the effect of method calls. If a method has not yet been analyzed, the analysis can it self expand the analysis to include this method. If a method tries to analyze itself it is conservatively assumed that all arguments escape and that the method is not fresh.

If IPA is not enabled the arguments to a method is conservatively assumed to escape.

9.3 Fitting it All Into Jikes

The Jikes RVM provides classes for almost any kind of manipulation of the IR code one could imagine. All analyses are done using these classes, though it already is quite easy to do analyses on a representation like HIR.

The only part of the Jikes RVM modified to include our escape analysis is the optimization planner which now includes a line for initializing the escape analysis.

To facilitate easy testing a small test suite was written to run a specified program a certain number of times, dumping the contents of the method summary database after each iteration. This can also be used for timing the process.

Chapter 10

Explicit Deallocation of Method-local Objects

This section presents two suggestions on how to deallocate method-local objects without the help of the garbage collection system. None of these suggestions have been implemented.

As Objects in the Jikes RVM are deallocated implicitly by garbage collection, an object will remain on the heap until it is identified as being unused by a garbage collector, which will then deallocate it.

Garbage collection is a costly process, as all other processes must halt while the garbage collector identifies and deallocates objects. A garbage collection can be triggered by a number of events, but usually the key factor is the amount of space left on the heap. If method-local objects could be deallocated explicitly when a method returns, fewer objects would be allocated on the heap, and in the best case several garbage collection-runs could be avoided.

When a method is invoked, a new stackframe is pushed to the top of the stack. When the method returns, its stackframe is popped from the stack. The stackframe's lifetime is thus, ignoring the prologue and epilogue, the same as the method's.

10.1 Stack Allocation of Method Local Objects

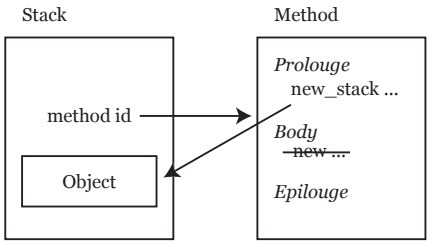


Figure 10.1: Objects could be allocated in the methods stackframe

If objects where stored inside the stackframe, they would be removed from the stack as the method returns. Method local objects can therefore safely be stored in

the method stackframe instead of the heap. When the method returns, the objects will be removed along with the stackframe.

This approach requires following steps

- The `new` instructions that would have allocated the objects on the heap must be removed from the method.
- A new `new_stack` instruction that allocates space on the stack must be inserted in the method prologue for each method-local object.
- The object must be allocated on the stack, and references to the allocated space must be stored in the registers that would have been set by the `new` instructions.

Figure 10.1 depicts how the stack and method would look after these steps. Note that we are operating on HIR level, and the code is in SSA-form. So if we store the reference to the stack-allocated method-local object in the same register as the objects `new` instruction would have done, we don't have to worry about the register being overwritten by some other instruction between the allocation and the point where the object's init-method needs the reference.

When a method's prologue is executed, the stackframe is already written, and it would be unfortunate to have to resize it. It is therefore necessary to do the object allocation in the stackframe when the stackframe itself is created. The `new_stack` instruction which is inserted in the method prologue will store the reference to the allocated object in the same register as the `new` would have.

No changes to the GC-map are needed, as the reference to the object on the stack is stored the same place as the `new` instruction would have stored its reference.

Objects are method-local, so when the method ends, no references to the object will exists other than the one originally created in the method. A block of memory on the stack is just as accessible as a block on the heap. A reference to an object on the stack is therefore as valid as a reference to an object on the heap.

Stack-allocation of the objects would probably be the most efficient way to allocate and deallocate method-local objects, but it could also turn out to be a bit difficult to implement as changes to the stackframe are needed. If the objects is to be stored on a stack-frame, the layout of the stackframe must be changed to accommodate it and the size of the stack would grow considerably. None of these problems would be impossible to overcome though.

10.1.1 Allocation of Object in a Object Stack

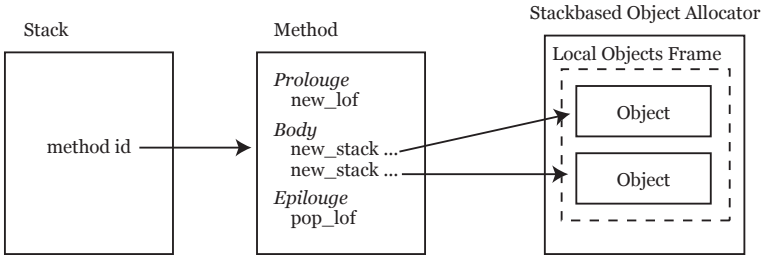


Figure 10.2: Method-local objects could be allocated in a object-stack

Another way of implementing the deallocaton of method-local objects is to introduce a new Stackbased Object Allocator (SOA). The SOA will maintain a stack of Local Object Frames (LOF), each frame will contain one or more objects. When a method invocation occurs, a new LOF will be created. Method-locale objects are now allocated in the LOF instead of the heap. When the method returns, the LOF is popped.

This approach requires following steps

- A `new_lof` instruction that creates a new LOF must be inserted in the method prologue.
- The `new` instructions that would have allocated the objects on the heap must be replaced with a `new_stack` instruction that allocates the object using the SOA.
- A `pop_lof` instruction that pops the methods LOF must be inserted in the method's epilogue.

This would result in the SOA pushing and popping LOFs in tandem with the stack. One possibility is to let the SOA's alloc-method accept a list of object types, and return a list of references to the newly allocated objects. These references would then be stored in the correct registers. The allocation of the objects would no longer need to be done in processes of creating stack-frames, but could be inserted in the method's prologue. When the method returns, the SOA simply pops the Local Object Frame. This model is depicted on figure 10.2

10.2 Implementation

None of the above described systems have been implemented. We are aware of the fact that we could have overlooked a lot of implementation specific problems, but we feel confident if one of the above systems where implemented it would lead to performance gain.

Part IV

Results

Chapter 11

Analysis Results

This chapter will describe the tests of the escape analysis implementation in the Jikes RVM. The test where also preformed on an actual application. A full listing of the tests and the analysis-result can be found in chapter B on page B in the appendix.

11.1 Testing the Jikes RVM

As the Jikes RVM is made with the purpose of being a testbed for VM technologies, it provides several possibilities for testing and benchmarking.

A set of tests is provided with the distribution, unfortunately none which can be used to verify escape analysis. The tests can be run on different versions (images) of the VM. The `OptTestHarness` image is designed to test the optimizing compiler. It compiles everything with the baseline compiler, except the classes specified by the user which are compiled by the optimizing compiler.

Benchmarking should be performed on images built with optimizing and adaptive optimizations such as an `FullAdaptive` image which will compile the whole VM with maximum optimizations and run the program in a adaptive context. This is currently the best performing Jikes RMV image. But benchmarks would only matter for measuring the overhead involved in performing the escape analysis, which has not been a goal for this paper.

The `Opt` images, images which compile all methods with the optimizing compiler, have terrible startup performance as all methods have to be compiled with the resource intensive optimizing compiler[RVM03].

11.2 Examining the Escape Analysis

To test the implementation of escape analysis we chose to use the `OptTestHarness` program. This gives us full control of which methods to compile and later checking against the source.

For each variable in a method following boolean properties are examined:

returned is the object returned from an method?

fresh is the register fresh?
escaped does the object escape the method?
loop is the object involved in more than one program-path?

Further more each method is flagged as fresh if it returns a fresh object.

Following tests will be preformed:

- Assignments to a static field
- Return of a fresh object
- Assignment to a non-static field
- Objects passed as argument to a method
- Recursive calls
- Multiple program paths

For a full listing of the test-results, see chapter B on page B in the appendix.

11.2.1 Assignment to a Static Field

This test will assign a fresh object to a static field, this should result in the object escaping.

Expected Result: the object is fresh, but escapes.
Result: the object is identified as fresh, and escaped. The test is successful.

As expected the fresh object is identified as having escaped. The object is fresh as it is created with the `new` instruction on line 4. The test is successful.

The Java- and HIR-code for this test can be seen:

Javacode:

```
1 static Object o;
2 public void assignStatic()
3 {
4    Object t = new Object();
5    o = t;
6 }
```

HIR-code:

```
1 -13   LABEL0 Frequency: 1.0
2 -2 EG ir_prologue       l0si(LTest3;,x,d) =
3 0   G yieldpoint_prologue
4 0  EG new               t2si(Ljava/lang/Object;,p) = java.lang.
    Object
5 4  EG call               AF CF OF PF SF ZF = 696, special_exact"
    java.lang.Object.<init> ()V", <TRUEGUARD>, t2si(Ljava/lang/
    Object;,p)
6 10    putfield          t2si(Ljava/lang/Object;,p), l0si(LTest3;,
    x,d), -16, <mem loc: LTest3;.i>, <TRUEGUARD>
7 -3    return            <unused>
8 -1    bbend             BB0 (ENTRY)
```

11.2.2 Returning a fresh object

This test will invoke a method that should return a fresh object. When a method returns a fresh object, the method itself should be marked as being fresh.

Expected Result: the method is marked as being fresh, and the object should be identified as being fresh and returned.

Result: the object is identified as fresh, and escaped. The method is fresh.

The algorithm successfully identifies the return object as fresh, and the method is also tagged as fresh. The test is successful.

The Java- and HIR-code for this test can be found in section B.2 on page 82 in the appendix.

11.2.3 Assignment of an Object to a Non-static Field

This test will assign a fresh object to a non-static field, the object should escape.

Expected Result: The object should escape.

Result: The object escapes, the test is successful.

In the current implementation of the escape analysis, an object escapes if it is assigned to a field. The object do not necessarily escape if it is assigned to non-static field of `this`, but the current implementation will not differentiate between `this` and any other object reference. The algorithm correctly identifies the object as escaped, the test is successful.

The Java- and HIR-code for this test can be found in section B.3 on page 83 in the appendix

11.2.4 Objects Passed as Argument

This test will pass two objects as arguments to a method. The callee will assign one of the objects to a static field, and the other will be returned. The parameter that is assigned to at static field should escape as it where the case in section 11.2.3. A parameter that is returned does not necessary escape.

Expected Result: The object that is assigned to a static field will escape, while the other will not.

Result: The first parameter (that is assigned to a static field) escapes, the second does not.

The test is successful.

The Java- and HIR-code for this test can be found in section B.4 and B.5 on page 84 and 85 in the appendix.

11.2.5 Recursive Calls

This test will invoke a method recursiveCall, that returns an objected created by a method `recursive` that calls `recursiveCall`. Despite the fact that this recursion is infinite, all objects should still be identified as not escaped.

Expected Result No objects should escape
Result No objects escape.

Even though the recursion is infinite, none of the objects survives longer than the method that originally invoked `recursiveCall`. Therefore non of them escapes. The test is successful.

The Java- and HIR-code for this test can be found in section B.7 and B.6 in the appendix.

11.2.6 Multiple program paths

This test contains a if-then-else block. This means that there are multiple paths in the program. The purpose of the test is to examine how the algorithm handles this.

Expected Result The object should escape.
Result The object escapes.

As the value of a variable cannot be determined if it is assigned in one or more program-paths that converge, if an object participates in multiple data-paths, it should simply be regarded as escaped. The algorithm correctly identifies the object as escaped. The test is successful.

The Java- and HIR-code for this test can be found in section B.8 on page

11.3 Examining Real Programs

We have examined real programs with a BaseOptSemiSpace image. The examined programs were:

VolanoMark - a benchmark simulating a Java servlet based chat server
Scimark - another benchmark simulating some scientific calculations

11.3.1 VolanoMark

VolanoMark consists of a server and a client program. The test program starts the server and connects 100 clients to the server.

The escape analysis identified 280 allocation sites in the VolanoMark server code. 139 of these could have been allocated on the stack.

The VolanoMark client exhibited similar behavior with 116 allocation sites of which 59 could have been stack allocated.

11.3.2 Scimark

Scimark is a different kind of beast than VolanoMark. It mainly uses primitives in calculating it's benchmarks. This is reflected in the results from the escape analysis which shows 33 allocations and 31 of these stack allocatable. This large percentage is caused by the way Scimark is constructed.

Part V

Discussion

Chapter 12

Discussion

In this paper we have presented and implemented an algorithm for escape analysis in the Jikes RVM, with the purpose of examining the feasibility of allocating objects on the stack compared to Java's default heap allocation.

The algorithm from [GS00] was chosen because of its simplicity and basis in abstract interpretation. As mentioned in chapter 5, others variations of escape analysis exists. These algorithms are more precise but also a lot more complex.

The Jikes RVM was chosen for many reasons. Most importantly, because it is implemented in Java, and thus provided all the benefits of the Java language. Also, the compile only approach easily enables static analysis of small programs. The escape analysis could just as well have been done using JavaCC or Soot or any similar framework, on Java source code or bytecode. If escape analysis had been the sole goal of this paper, a functional language like SML would have been more suitable for implementing the actual escape analysis.

Our tests shows that our implementation works as expected, and that approximately 50% of all objects in a real application are suitable for allocation on the stack. The original [GS00] report identifies approximately of 20% of all objects as allocable on the stack. We are confident that a broader testsuite would result in similar test. It should also be noted that the high percentage of stack allocatable objects, might be due to the fact the we count allocation statements in the code and not the number of allocated objects at runtime.

Not all local objects are suitable for allocation on the stack. It is only desirable to allocate an object on the stack if the lifetime of its scope is relatively short compared to the program. This is due to the fact that, unlike heap-allocated objects, a stack-allocated object cannot be deallocated until its scope terminates, tying up precious memory. For a discussion of escaping objects and their lifetime and connectivity see [HHDH02].

In listing 12.1, the object referenced by `a` isn't used past line 4. It clearly does not escape `main`. Normally, if `a` is allocated on the heap, liveliness analysis [GBJL00] would determine that `a` isn't referenced past line 4, and therefore can be triggered for garbage collection. If `a` is allocated on the stack, it will not be allocated until `main` terminates on line 8 after the call to `someLargeMethod`.

Listing 12.1: Stack-allocating an object in main()

```
1 public static void main( String args[] )
2 {
3    C a = new C();
4    System.out.println( a.f );
5
6    C b = new C();
7    b.someLargeMethod();
8 }
```

Because of this the algorithm is not dependent on analysis of the whole program. Outlook for combining the escape analysis and eventually stack allocation with the Jikes RVM's adaptive optimization system looks bright as the adaptive system will.

So would stack allocation in the Jikes RVM be feasible? Absolutely. But it would require a some non-trivial changes to the Jikes RVM and optimization both for speed and memory utilization of the escape analysis implementation.

Chapter 13

Conclusion

We have successfully implemented escape analysis in the Jikes RVM. The escape analysis implementation correctly determines objects that escape the creating method's scope in complex programs. Furthermore, we have suggested two seperat ways to achieve stack allocation of objects in the Jikes RVM, based on the information collected in the escape analysis. We are confident that both would prove useful if implemented.

Chapter 14

Related and Future Work

Recent work in escape analysis either takes a more theoretical approach ([Bla98], [HS], [WR99]) or focus on the speed improvements the analysis results can facilitate ([CGS+99], [GS00]).

Future work on the escape analysis we have implemented in the Jikes RVM could include expanding the algorithm to handle synchronization elimination and as suggested stack allocation of objects.

To support future optimizations more benchmarks should be performed e.g. with regards to how many of the actual allocated objects could be allocated on stack and studies of these lifetimes.

Furthermore a similar algorithm might be interesting to see implemented in the open source static Java compiler GCJ.

Part VI

Lists and Citations

Chapter 15

Credits

The following figures where redrawn from ASCII-illustrations found in the Jikes source-code.

Figure 8.1 on page 41 where found in /rvm/src/vm/objectModel/VM_ObjectModel.java in the 2.2.0 distribution.
Authors:

- Bowen Alpern
- David Bacon
- Stephen Fink
- Dave Grove
- Derek Lieber

Figure 8.2 on page 42 where found in /rvm/src/vm/runtime/VM_Statics.java in the 2.2.0 distribution.
Authors:

- Bowen Alpern
- Derek Lieber

Figure 8.3 on page 43 where found in /rvm/src/vm/arch/intel/VM_StackframeLayoutConstants.java in the 2.2.0 distribution.
Authors:

- David Grove
- Bowen Alpern

Figure 6.1 on page 31 is based on the figure found page on page 6 in [AFG+00].
Authors:

- Matthew Arnold
- Stephen Fink
- David Grove
- Michael Hind
- Peter F. Sweeney

Remaining figures are all done by Ulf Holm Nielsen, Thomas Riisbjerg, Trols Krogh, or Mads Danquah. They may be freely used as long as the authors are credited.

List of Figures

Listings

Chapter 16

Citations

[AAB+00] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. *The Jalapeño Virtual Machine*. IBM System Journal, vol 39, no 1, february edition, 2000.

http://www.research.ibm.com/journal/sj/391/alpern.pdf

[AFG+00] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. *Adaptive Optimization in the Jalapeño JVM*. 2000.

http://www.research.ibm.com/jalapeno/papers/oopsla00_aos.pdf

[BCF+99] Michael Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio Serrano, V.C. Sreedhar, and Harini Srinivasan. *The Jalapeño Dynamic Optimizing Compiler for Java*. 1999.

http://www.research.ibm.com/jalapeno/papers/grande99.ps

[Bla98] Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Symposium on Principles of Programming Languages*, pages 25–37, 1998.

[Bre03] Shane A. Brewer. *Jikes Intermediate Code Representation*. University of Alberta, 2003.

http://www.cs.ualberta.ca/~brewer/presentations/jikesIR.ppt

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CGHS99] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. *Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs*. IBM Research, 1999.

http://www.research.ibm.com/jalapeno/pub/paste99.ps

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape analysis for java. In *Proceed-*

ings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), pages 1–19, 1999.

[Cla03] GNU Classpath. *The GNU Classpath Project*. 2003.

http://www.gnu.org/software/classpath/

[dev03a] IBM developerWorks. *Jikes ™*. IBM developerWorks, 2003.

http://oss.software.ibm.com/developerworks/opensource/jikes/

[dev03b] IBM developerWorks. *Jikes ™RVM*. IBM developerWorks, 2003.

http://www-124.ibm.com/developerworks/oss/jikesrvm/

[dev03c] IBM developerWorks. *Jikes ™RVM source distribution*. IBM developerWorks, 2003.

http://www-124.ibm.com/developerworks/oss/jikesrvm/download/index.shtml

[FKR+00] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. *Marmot: an optimizing compiler for Java*, volume 30. 2000.

http://citeseer.nj.nec.com/fitzgerald99marmot.html

[GBJL00] Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs, and Koben G. Lagendoen. *Modern Compiler Design*. WILEY, 2000.

[GS00] David Gay and Bjarne 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.

[HHDH02] Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. *Understanding the Connectivity of Heap Objects*. jun 2002.

http://citeseer.nj.nec.com/hirzel02understanding.html

[HS] Patricia M. Hill and Fausto Spoto. *A Foundation of Escape Analysis*.

[Kri03] Chandra Krintz. *Lecture notes on Jalapeño*. University of California, Santa Barbara, 2003.

http://www.cs.ucsb.edu/~ckrintz/

[LY99] Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification*. 1999.

http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

[Rin] Martin Rinard. *Pointer and Escape Analysis*.

http://www.cag.lcs.mit.edu/~rinard/pointer_and_escape_analysis/

[Ros95] Mads Rosendahl. *Introduction to Abstract Interpretation*. DIKU, Computer Science University of Copenhagen, 1995.

http://www.dat.ruc.dk/~madsr/webpub/absint.pdf

[RVM03] Jikes RVM. *The Jikes™ Research Virtual Machine User's Guide post 2.2.1*. 2003.

http://www-124.ibm.com/developerworks/oss/jikesrvm/userguide/HTML/index.html

[SM02] Inc Sun Microsystems. *The Java™ HotSpot Virtual Machine*. Sun Microsystems, Inc, 2002.

http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. *ACM SIGPLAN Notices*, 34(10):187–206, 1999.

Chapter 17

Additional Litterature

[CC77] P. Cousot and R. Cousot. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. ACM Press, New York, NY, Los Angeles, California, 1977.

Part VII

Appendix

Appendix A

Source-code for Escape Analysis

```
/**
 * -----------------------------------------------------------------------------
 * "THE BEER-WARE LICENSE" (Revision 42):
 * <doktor@dyregod.dk> <mads@danquah.dk> <tkrogh@ruc.dk> <tnjr@ruc.dk> wrote
 * this file. As long as you retain this notice you can do whatever you want
 * with this stuff. If we meet some day, and you think this stuff is worth it,
 * you can buy us a beer in return.
 * -----------------------------------------------------------------------------
 */
package com.ibm.JikesRVM.opt;

import com.ibm.JikesRVM.*;
import com.ibm.JikesRVM.classloader.*;

import java.util.*;

import com.ibm.JikesRVM.opt.ir.*;
/**
 * Does the actual escape analysis, based on paper from 2000 by David Gay and Bj
arne Steensgaard
 *
 * Last change by: $Author: dyregod $
 *
 * $Header: /var/cvs/Jikes\040RVM/com/ibm/JikesRVM/opt/OPT_EscapeAnalysis.java,v
 1.6 2003/05/25 12:27:52 dyregod Exp $
 *
 * @version $Revision: 1.6 $
 */
public class OPT_EscapeAnalysis extends OPT_CompilerPhase implements OPT_Operato
rs
{
    public final boolean shouldPerform(OPT_Options options)
    {
        if (options.getOptLevel() >= 2 && (options.ESCAPE_IPA || options.ESCAPE_
NONIPA))
            return true;
        else
            return false;
    }
    public void perform(OPT_IR ir)
    {
        OPT_EscapeSummary summary = OPT_EscapeSummaryDatabase.getEscapeSummary(i
r.getMethod());
        if (summary == null)
            summary = OPT_EscapeSummaryDatabase.createEscapeSummary(ir.getMethod
());
        summary.working = true;

        for (OPT_BasicBlockEnumeration oe = ir.getBasicBlocks(); oe.hasMoreEleme
nts();)
        {
            OPT_BasicBlock bb = oe.next();
            int ints = 0;
            for (OPT_InstructionEnumeration ie = bb.forwardInstrEnumerator(); ie
.hasMoreElements();)
            {
                OPT_Instruction i = ie.next();
                char opcode = i.getOpcode();
                switch (opcode)
                {
                    case IR_PROLOGUE_opcode :
                        prologue(i, summary);
                        break;
```



```
                    case CALL_opcode :
                        call(i, summary, ir.options);
                        break;
                    case REF_MOVE_opcode :
                        move(i, summary);
                        break;
                    case PUTFIELD_opcode :
                        putfield(i, summary);
                        break;
                    case PUTSTATIC_opcode :
                        putstatic(i, summary);
                        break;
                    case ATHROW_opcode :
                        throwed(i, summary);
                        break;
                    case RETURN_opcode :
                        returns(i, summary);
                        break;
                        // only new and new unresolved, as we don't yet allocate
 arrays on stack
                    case NEW_UNRESOLVED_opcode :
                        news(i, summary);
                        break;
                    case NEW_opcode :
                        news(i, summary);
                        break;
                        // phi join
                    case PHI_opcode :
                        phi(i, summary);
                        break;
                    default :
                        break;
                }
            }
        }

        //
        if ()

        int s = 0;
        OPT_MethodRegisterProperties[] mregs =
            (OPT_MethodRegisterProperties[]) summary.store.values().toArray(new
OPT_MethodRegisterProperties[] {
        });
        System.out.println("***** BEGIN HIR for " + ir.getMethod() + " *****");
        ir.printInstructions();
        System.out.println("***** END HIR for " + ir.getMethod() + " *****");

        System.out.println("var:");
        OPT_Register[] regs = (OPT_Register[]) summary.store.keySet().toArray(ne
w OPT_Register[] {
        });
        for (int k = 0; k < mregs.length; k++)
        {
            if (!mregs[k].props[OPT_MethodRegisterProperties.ESCAPED]
                && !mregs[k].props[OPT_MethodRegisterProperties.LOOP]
                && !mregs[k].props[OPT_MethodRegisterProperties.RETURNED]
                && mregs[k].props[OPT_MethodRegisterProperties.FRESH])
                s++;
            System.out.print(
                regs[k]
                    + ": returned = "
                    + mregs[k].props[2]
```

```
                    + ", fresh = "
                    + mregs[k].props[0]
                    + ", escaped = "
                    + mregs[k].props[1]
                    + ", loop = "
                    + mregs[k].props[3]
                    + "\n");
        }
        System.out.println("parameters:");
        for (int i = 0; i < summary.parmprop.length; i++)
        {
            System.out.print(
                summary.parameters
                    + ": returned = "
                    + summary.parmprop[i].props[2]
                    + ", fresh = "
                    + summary.parmprop[i].props[0]
                    + ", escaped = "
                    + summary.parmprop[i].props[1]
                    + ", loop = "
                    + summary.parmprop[i].props[3]
                    + "\n");

        }

        System.out.println("Method is fresh? " + summary.fresh);
        System.out.println("Total new: " + summary.news);
        System.out.println("Total stackable: " + s);
    }
    private void prologue(OPT_Instruction i, OPT_EscapeSummary summary)
    {
        summary.setParameters(Prologue.getNumberOfFormals(i));
        summary.parameters = new OPT_Operand[Prologue.getNumberOfFormals(i)];
        for (int j = 0; j < summary.parameters.length; j++)
        {
            summary.parameters[j] = Prologue.getFormal(i, j);
        }
    }
    private void putstatic(OPT_Instruction i, OPT_EscapeSummary summary)
    {
        OPT_Operand oper = PutStatic.getValue(i);
        if (oper.isRegister())
        {
            summary.setEscaped(oper.asRegister().register, true);
        }
    }
    private void throwed(OPT_Instruction i, OPT_EscapeSummary summary)
    {
        OPT_Operand oper = Athrow.getValue(i);
        if (oper.isRegister())
        {
            summary.setEscaped(oper.asRegister().register, true);
        }
    }
    private void returns(OPT_Instruction i, OPT_EscapeSummary summary)
    {
        OPT_Operand operand = Return.getVal(i);

        if (operand != null && operand.isRegister())
        {
            OPT_Register register = operand.asRegister().register;
            summary.setReturned(register, true);
            summary.fresh = summary.getFresh(register);
```



```
        }
    }
    private void putfield(OPT_Instruction i, OPT_EscapeSummary summary)
    {
        OPT_Operand operand = PutField.getValue(i);
        if (operand.isRegister())
        {
            OPT_Register source = operand.asRegister().register;
            summary.setEscaped(source, true);
        }
    }
    private void phi(OPT_Instruction i, OPT_EscapeSummary summary)
    {
        OPT_Register des = Phi.getResult(i).asRegister().register;
        OPT_Operand source = Phi.getValue(i, 1);
        if (source != null && source.isRegister())
        {
            summary.setDependence(des, source.asRegister().register, OPT_MethodR
egisterProperties.ESCAPED);
            summary.setDependence(des, source.asRegister().register, OPT_MethodR
egisterProperties.RETURNED);
            summary.setLoop(source.asRegister().register, true);
        }
        source = Phi.getValue(i, 0);
        if (source != null && source.isRegister())
        {
            summary.setDependence(des, source.asRegister().register, OPT_MethodR
egisterProperties.ESCAPED);
            summary.setDependence(des, source.asRegister().register, OPT_MethodR
egisterProperties.RETURNED);
            summary.setLoop(source.asRegister().register, true);
        }
        summary.setFresh(des, false);
    }
    private void news(OPT_Instruction i, OPT_EscapeSummary summary)
    {
        OPT_Operand oper = New.getResult(i);
        if (oper.isRegister())
        {
            summary.setFresh(oper.asRegister().register, true);
        }
        summary.news++;
    }
    private void move(OPT_Instruction i, OPT_EscapeSummary summary)
    {
        OPT_Register dest = Move.getResult(i).asRegister().register;
        if (!Move.getVal(i).isRegister())
        {
            summary.setFresh(dest, false);
        }
        else
        {
            OPT_Register source = i.getOperand(1).asRegister().register;
            summary.setDependence(dest, source, OPT_MethodRegisterProperties.ESC
APED);
            summary.setDependence(dest, source, OPT_MethodRegisterProperties.LOO
P);
            summary.setDependence(dest, source, OPT_MethodRegisterProperties.RET
URNED);
        }
    }
    private void call(OPT_Instruction i, OPT_EscapeSummary summary, OPT_Options
options)
```

```
    {
        OPT_Register retval = null;
        OPT_Register source = null;
        OPT_Operand oper = Call.getResult(i);
        if (oper != null && oper.isRegister())
            retval = oper.asRegister().register;

        OPT_MethodOperand method = Call.getMethod(i);
        OPT_Operand[] parameters = new OPT_Operand[Call.getNumberOfParams(i)];
        for (int j = 0; j < parameters.length; j++)
        {
            /*if (Call.getParam(i, j).isRegister())
            {*/
            //summary.setEscaped(Call.getParam(i, j).asRegister().register, true
);
            parameters[j] = Call.getParam(i, j);
            //  }
        }
        if (method == null)
        {
            for (int j = 0; j < parameters.length; j++)
            {
                if (parameters[j].isRegister())
                    summary.setEscaped(parameters[j].asRegister().register, true
);
            }
            return;
        }
        // we automagically detect if we are trying to evaluate recursively. It
will just say that its not fresh as fresh is init to false
        OPT_EscapeSummary es = OPT_EscapeSummaryDatabase.getEscapeSummary(method
.getTarget());
        if (es != null)
        {
            if (es.fresh && retval != null)
                summary.setFresh(retval, true);

            checkParameters(summary, es, parameters, retval);
        }
        else if (options.ESCAPE_IPA)
        {
            OPT_OptimizationPlanCompositeElement eplan =
                OPT_OptimizationPlanCompositeElement.compose(
                    "Escape Analysis",
                    new Object[] { new OPT_ConvertBCtoHIR(), new OPT_Simple(true
, true), new OPT_EscapeAnalysis()});
            if (method.getTarget() == null || method.getTarget().isAbstract())
                return;
            if (!(method.getTarget() instanceof VM_NormalMethod))
                return;
            es = OPT_EscapeSummaryDatabase.createEscapeSummary(method.getTarget(
));
            //es.setParameters(Call.getNumberOfParams(i));

            OPT_CompilationPlan plan = new OPT_CompilationPlan((VM_NormalMethod)
 method.getTarget(), eplan, null, options);
            plan.analyzeOnly = true;
            try
            {
                OPT_Compiler.compile(plan);
            }
            catch (OPT_MagicNotImplementedException e)
            {
```



```
                //ignore
            }
            checkParameters(summary, es, parameters, retval);
            if (es.fresh && retval != null)
                summary.setFresh(retval, true);

        }
    }
    private void checkParameters(OPT_EscapeSummary summary, OPT_EscapeSummary es
, OPT_Operand[] parameters, OPT_Register retval)
    {
        if (es.parmprop != null)
        {
            for (int j = 0; j < es.parmprop.length; j++)
            {
                if (!parameters[j].isRegister())
                    continue;
                if (es.parmprop[j].props[OPT_MethodRegisterProperties.RETURNED]
== true && retval != null)
                {
                    summary.setDependence(retval, parameters[j].asRegister().reg
ister, OPT_MethodRegisterProperties.ESCAPED);
                    summary.setDependence(retval, parameters[j].asRegister().reg
ister, OPT_MethodRegisterProperties.RETURNED);
                    summary.setDependence(retval, parameters[j].asRegister().reg
ister, OPT_MethodRegisterProperties.LOOP);
                }
                if (es.parmprop[j].props[OPT_MethodRegisterProperties.ESCAPED] =
= true)
                {
                    // safe as the parameter can no longer change its value
                    summary.setEscaped(parameters[j].asRegister().register, true
);
                }
            }
        }
        else
        {
            for (int j = 0; j < parameters.length; j++)
            {
                if (parameters[j].isRegister())
                    summary.setEscaped(parameters[j].asRegister().register, true
);
            }
        }

    }
    public String getName()
    {
        return "Escape Analysis";
    }
}
```

```
/**
 * -----------------------------------------------------------------------------
 * "THE BEER-WARE LICENSE" (Revision 42):
 * <doktor@dyregod.dk> <mads@danquah.dk> <tkrogh@ruc.dk> <tnjr@ruc.dk> wrote
 * this file. As long as you retain this notice you can do whatever you want
 * with this stuff. If we meet some day, and you think this stuff is worth it,
 * you can buy us a beer in return.
 * -----------------------------------------------------------------------------
 */
package com.ibm.JikesRVM.opt;

import java.util.HashMap;
import java.util.Iterator;

import com.ibm.JikesRVM.opt.ir.*;
/**
 * Contains info on all registers in a method
 *
 * Last change by: $Author: dyregod $
 *
 * $Header: /var/cvs/Jikes\040RVM/com/ibm/JikesRVM/opt/OPT_EscapeSummary.java,v
1.4 2003/05/25 12:27:52 dyregod Exp $
 *
 * @version $Revision: 1.4 $
 */
public class OPT_EscapeSummary
{
    public HashMap store = new HashMap();
    public boolean fresh = false;
    boolean working = false;
    public int news = 0;
    public OPT_Operand[] parameters;
    public OPT_MethodRegisterProperties[] parmprop;

    public void setParameters(int i)
    {
        parmprop = new OPT_MethodRegisterProperties[i];
        for (int j = 0; j < parmprop.length; j++)
        {
            parmprop[j] = new OPT_MethodRegisterProperties();
        }
    }
    private int isParameter(OPT_Register register)
    {
        for (int i = 0; i < parameters.length; i++)
        {
            if (parameters[i].isRegister())
            {
                if (parameters[i].asRegister().register.equals(register))
                {
                    return i;
                }
            }
        }
        return -1;
    }
    public void setProperty(OPT_Register register, int prop, boolean value)
    {
        int i = isParameter(register);
        if (i != -1)
        {
            parmprop[i].props[prop] = value;
        }
```



```
        else
        {

            OPT_MethodRegisterProperties mreg;
            Object o = store.get(register);
            if (o == null)
            {
                mreg = new OPT_MethodRegisterProperties();
                store.put(register, mreg);
            }
            else
            {
                mreg = (OPT_MethodRegisterProperties) o;
            }
            mreg.props[prop] = value;
        }
        resolveDependencies(register);
    }
    public void setReturned(OPT_Register register, boolean returned)
    {
        setProperty(register, OPT_MethodRegisterProperties.RETURNED, returned);
    }
    public void setFresh(OPT_Register register, boolean fresh)
    {
        setProperty(register, OPT_MethodRegisterProperties.FRESH, fresh);
    }
    public void setEscaped(OPT_Register register, boolean escaped)
    {
        setProperty(register, OPT_MethodRegisterProperties.ESCAPED, escaped);
    }
    public void setLoop(OPT_Register register, boolean loop)
    {
        setProperty(register, OPT_MethodRegisterProperties.LOOP, loop);
    }
    public void setDependence(OPT_Register trig, OPT_Register tar, int prop)
    {
        setDependence(trig, tar, prop, this);
    }
    public void setDependence(OPT_Register trigger, OPT_Register target, int pro
p, OPT_EscapeSummary summary)
    {
        OPT_MethodRegisterProperties mreg;
        Object o = store.get(trigger);
        if (o == null)
        {
            mreg = new OPT_MethodRegisterProperties();
            //mreg.number = trigger.number;
            store.put(trigger, mreg);
        }
        o = store.get(target);
        if (o == null)
        {
            mreg = new OPT_MethodRegisterProperties();
            store.put(target, mreg);
        }
        Dependence dep = new Dependence(target, prop);
        ((OPT_MethodRegisterProperties) store.get(trigger)).deps.add(dep);
    }
    public void resolveDependencies(OPT_Register register)
    {
        OPT_MethodRegisterProperties smreg;
        int i = isParameter(register);
        if (i != -1)
```

```
            smreg = parmprop[i];
        else
            smreg
             = (OPT_MethodRegisterProperties) store.get(register);
        Iterator iter = smreg.deps.iterator();
        while (iter.hasNext())
        {
            Dependence element = (Dependence) iter.next();
            OPT_MethodRegisterProperties tmreg = (OPT_MethodRegisterProperties)
store.get(element.register);

            if (smreg.props[element.prop])
                tmreg.props[element.prop] = true;
        }
    }
    public boolean getFresh(OPT_Register register)
    {
        OPT_MethodRegisterProperties mreg = new OPT_MethodRegisterProperties();
        Object o = store.get(register);
        if (o == null)
        {
            return false;
        }
        else
        {
            return ((OPT_MethodRegisterProperties) o).props[OPT_MethodRegisterPr
operties.FRESH];
        }
    }
    public boolean isWorking()
    {
        return working;
    }
    public void setWorking(boolean w)
    {
        working = w;
    }
}
class Dependence
{
    OPT_Register register;
    int prop;
    public Dependence(OPT_Register register, int prop)
    {
        this.register = register;
        this.prop = prop;
    }
}
```



```
/**
 * ------------------------------------------------------------------------
 * "THE BEER-WARE LICENSE" (Revision 42):
 * <doktor@dyregod.dk> <mads@danquah.dk> <tkrogh@ruc.dk> <tnjr@ruc.dk> wrote
 * this file. As long as you retain this notice you can do whatever you want
 * with this stuff. If we meet some day, and you think this stuff is worth it,
 * you can buy us a beer in return.
 * ------------------------------------------------------------------------
 */
package com.ibm.JikesRVM.opt;

import java.util.HashMap;

import com.ibm.JikesRVM.classloader.VM_Method;
/**
 * Contains all method summaries
 *
 * Last change by: $Author: dyregod $
 *
 * $Header: /var/cvs/Jikes\040RVM/com/ibm/JikesRVM/opt/OPT_EscapeSummaryDatabase
.java,v 1.2 2003/05/25 12:27:52 dyregod Exp $
 *
 * @version $Revision: 1.2 $
 */
public class OPT_EscapeSummaryDatabase
{
    static HashMap database = new HashMap();
    public static OPT_EscapeSummary[] getAll()
    {
        return (OPT_EscapeSummary[]) database.values().toArray(new OPT_EscapeSum
mary[] {});
    }
    public static OPT_EscapeSummary getEscapeSummary(VM_Method method)
    {
        return (OPT_EscapeSummary) database.get(method);
    }
    public static OPT_EscapeSummary createEscapeSummary(VM_Method method)
    {
        OPT_EscapeSummary summary = new OPT_EscapeSummary();
        database.put(method, summary);
        return summary;
    }

}
```

OPT_MethodRegisterProperties.java


```
/**
 * -----------------------------------------------------------------------
 * "THE BEER-WARE LICENSE" (Revision 42):
 * <doktor@dyregod.dk> <mads@danquah.dk> <tkrogh@ruc.dk> <tnjr@ruc.dk> wrote
 * this file. As long as you retain this notice you can do whatever you want
 * with this stuff. If we meet some day, and you think this stuff is worth it,
 * you can buy us a beer in return.
 * -----------------------------------------------------------------------
 */
package com.ibm.JikesRVM.opt;

import java.util.LinkedList;
/**
 * Contains properties for each method
 *
 * Last change by: $Author: dyregod $
 *
 * $Header: /var/cvs/Jikes\040RVM/com/ibm/JikesRVM/opt/OPT_MethodRegisterPropert
ies.java,v 1.2 2003/05/25 12:27:52 dyregod Exp $
 *
 * @version $Revision: 1.2 $
 */
public class OPT_MethodRegisterProperties
{
    // ought to be ok as code i supposed on ssa form
    public static final int FRESH = 0;
    public static final int ESCAPED = 1;
    public static final int RETURNED = 2;
    public static final int LOOP = 3;
    public boolean[] props = new boolean[4];
    public LinkedList deps = new LinkedList();
}
```

Appendix B

Test

The class used for testing contains a number of methods, each used for a specific test. The test itself is executed from the `test()` method, which again is executed from the `main()` method of the test-class.

Listing B.1: The class used for testing

```
1  public class Test3
2  {
3     Object i;
4     static Object o;
5
6     public static void main(String[] args)
7     {
8        Test3 t = new Test3();
9        t.test();
10    }
11    public void test()
12    {
13       assignStatic();
14       Object b = fresh();
15       assignField();
16       methodCall();
17       recursiveCall();
18       phi(4);
19    }
20
21 (Test methods here)
22 }
```

The following sections contains the source code for each test methods (including `main()` and `test()`. Lines of code from the above will be includes as needed, to increase readability.

B.1 assignStatic

Java-code

```
static Object o;

public void assignStatic()
{
   Object t = new Object();
   o = t;
}
```

HIR-code

```
-13   LABEL0 Frequency: 1.0
-2 EG ir_prologue       l0si(LTest3;,x,d) =
0   G yieldpoint_prologue
0  EG new               t2si(Ljava/lang/Object;,p) = java.lang.
   Object
4  EG call               AF CF OF PF SF ZF = 696, special_exact"
   java.lang.Object.<init> ()V", <TRUEGUARD>,
t2si(Ljava/lang/Object;,p)
9     putstatic         t2si(Ljava/lang/Object;,p), 64904, <mem
   loc: LTest3;.o>
-3    return            <unused>
-1    bbend             BB0 (ENTRY)
```

Analysis-result

```
var:
   t2si: returned = false, fresh = true, escaped = true, loop =
      false
parameters:
Method is fresh? false
Total new: 1
Total stackable: 0
```

B.2 fresh

Java-code

```
1 public Object fresh()
2 {
3    return new Object();
4 }
```

HIR-code

```
1 -13   LABEL0 Frequency: 1.0
2 -2 EG ir_prologue       l0si(LTest3;,x,d) =
3 0   G yieldpoint_prologue
4 0  EG new               t3si(Ljava/lang/Object;,p) = java.lang.
    Object
5 4  EG call               AF CF OF PF SF ZF = 696, special_exact"
    java.lang.Object.<init> ()V", <TRUEGUARD>, t3si(Ljava/lang/
    Object;,p)
6 -3    return            t3si(Ljava/lang/Object;)
7 -1    bbend             BB0 (ENTRY)
```

Analysis-result

```
1 var:
2    t3si: returned = true, fresh = true, escaped = false, loop =
        false
3 parameters:
4 Method is fresh? true
5 Total new: 1
6 Total stackable: 0
```

B.3 assignField

Java-code

```
Object i;

public void assignField()
{
   Object t = new Object();
   i = t;
}
```

HIR-code

```
-13   LABEL0 Frequency: 1.0
-2 EG ir_prologue       l0si(LTest3;,x,d) =
0   G yieldpoint_prologue
0  EG new               t2si(Ljava/lang/Object;,p) = java.lang.
   Object
4  EG call               AF CF OF PF SF ZF = 696, special_exact"
   java.lang.Object.<init> ()V", <TRUEGUARD>, t2si(Ljava/lang/
   Object;,p)
10    putfield          t2si(Ljava/lang/Object;,p), l0si(LTest3;,
   x,d), -16, <mem loc: LTest3;.i>, <TRUEGUARD>
-3    return            <unused>
-1    bbend             BB0 (ENTRY)
```

Analysis-result

```
var:
   t2si: returned = false, fresh = true, escaped = true, loop =
      false
parameters:
Method is fresh? false
Total new: 1
Total stackable: 0
```

B.4 method

Java-code

```
static Object o;

public Object method(Object a, Object b)
{
   o = a;
   return b;
}
```

HIR-code

```
-13   LABEL0 Frequency: 1.0
-2 EG ir_prologue       l0si(LTest3;,x,d), l2si(Ljava/lang/Object
   ;,x,d), l3si(Ljava/lang/Object;,x,d) =
0   G yieldpoint_prologue
1     putstatic         l2si(Ljava/lang/Object;,x,d), 64904, <mem
     loc: LTest3;.o>
-3    return            l3si(Ljava/lang/Object;)
-1    bbend             BB0 (ENTRY)
```

Analysis-result

```
var:
parameters:
[Lcom.ibm.JikesRVM.opt.ir.OPT_Operand;@c8: returned = false,
   fresh = false, escaped = true, loop = false
[Lcom.ibm.JikesRVM.opt.ir.OPT_Operand;@c8: returned = true, fresh
    = false, escaped = false, loop = false
Method is fresh? false
Total new: 0
Total stackable: 0
```

B.5 methodCall

Java-code

```
public void methodCall()
{
   Object a = new Object();
   Object b = new Object();
   method(a, b);
}
```

HIR-code

```
-13   LABEL0 Frequency: 1.0
-2 EG ir_prologue       l0si(LTest3;,x,d) =
0   G yieldpoint_prologue
0  EG new               t2si(Ljava/lang/Object;,p) = java.lang.
   Object
4  EG call               AF CF OF PF SF ZF = 696, special_exact"
   java.lang.Object.<init> ()V", <TRUEGUARD>, t2si(Ljava/lang/
   Object;,p)
8  EG new               t6si(Ljava/lang/Object;,p) = java.lang.
   Object
12 EG call               AF CF OF PF SF ZF = 696, special_exact"
   java.lang.Object.<init> ()V", <TRUEGUARD>, t6si(Ljava/lang/
   Object;,p)
19 EG call              t10si(Ljava/lang/Object;) AF CF OF PF SF
   ZF = 80, virtual"Test3.method (Ljava/lang/Object;Ljava/lang/
   Object;)Ljava/lang/Object;",
<TRUEGUARD>, l0si(LTest3;,x,d), t2si(Ljava/lang/Object;,p), t6si(
   Ljava/lang/Object;,p)
-3    return            <unused>
-1    bbend             BB0 (ENTRY)
```

Analysis-result

```
var:
t6si: returned = false, fresh = true, escaped = false, loop =
   false
t2si: returned = false, fresh = true, escaped = false, loop =
   false
l0si: returned = false, fresh = false, escaped = true, loop =
   false
t10si: returned = false, fresh = false, escaped = false, loop =
   false
parameters:
Method is fresh? false
Total new: 2
Total stackable: 2
```

B.6 recursive

Java-code

```
public Object recursive(Object a)
{
   return recursiveCall();
}
```

HIR-code

```
-13   LABEL0 Frequency: 1.0
-2 EG ir_prologue       l0si(LTest3;,x,d), l2si(Ljava/lang/Object
   ;,x,d) =
0   G yieldpoint_prologue
1  EG call              t4si(Ljava/lang/Object;) AF CF OF PF SF
   ZF = 84, virtual"Test3.recursiveCall ()Ljava/lang/Object;", <
   TRUEGUARD>, l0si(LTest3;,x,d)
-3    return            t4si(Ljava/lang/Object;)
-1    bbend             BB0 (ENTRY)
```

Analysis-result

```
var:
   t4si: returned = true, fresh = false, escaped = false, loop =
      false
parameters:
   [Lcom.ibm.JikesRVM.opt.ir.OPT_Operand;@ef: returned = false,
      fresh = false, escaped = false, loop = false
Method is fresh? false
Total new: 0
Total stackable: 0
```

B.7 recursiveCall

Java-code

```
1 public Object recursiveCall()
2 {
3    Object a = new Object();
4    return recursive(a);
5 }
```

HIR-code

```
1 -13   LABEL0 Frequency: 1.0
2 -2 EG ir_prologue       l0si(LTest3;,x,d) =
3 0   G yieldpoint_prologue
4 0  EG new               t3si(Ljava/lang/Object;,p) = java.lang.
    Object
5 4  EG call               AF CF OF PF SF ZF = 696, special_exact"
    java.lang.Object.<init> ()V", <TRUEGUARD>,
6 t3si(Ljava/lang/Object;,p)
7 10 EG call              t7si(Ljava/lang/Object;) AF CF OF PF SF
    ZF = 88, virtual"Test3.recursive (Ljava/lang/Object;)Ljava/
    lang/Object;", <TRUEGUARD>, l0si(LTest3;,x,d), t3si(Ljava/lang
    /Object;,p)
8 -3    return            t7si(Ljava/lang/Object;)
9 -1    bbend             BB0 (ENTRY)
```

Analysis-result

```
1 var:
2    t3si: returned = false, fresh = true, escaped = false, loop =
        false
3    t7si: returned = true, fresh = false, escaped = false, loop =
        false
4 parameters:
5 Method is fresh? false
6 Total new: 1
7 Total stackable: 1
```

B.8 phi

Java-code

```
public int phi(int i)
{
   int x = 0;
   if (i < 4)
      i = i *5;
   else
      i = i / 4;
   x = i;
   return x;
}
```

HIR-code

```
-13   LABEL0 Frequency: 1.0
-2 EG ir_prologue       l0si(LTest3;,x,d), t6psi(I,d) =
0   G yieldpoint_prologue
4     int_ifcmp         t5sv(GUARD) = t6psi(I,d), 4, >=, LABEL2,
   Probability: 0.5
-1    bbend             BB0 (ENTRY)
7     LABEL1 Frequency: 0.5
9     int_mul           t8psi(I) = t6psi(I,d), 5
11    goto              LABEL3
-1    bbend             BB1
14    LABEL2 Frequency: 0.5
16    int_div           t9psi(I) = t6psi(I,d), 4, <TRUEGUARD>
-1    bbend             BB2
18    LABEL3 Frequency: 1.0
-9    phi               t10psi(I) = t8psi(I), BB1, t9psi(I), BB2
-3    return            t10psi(I)
-1    bbend             BB3
```

Analysis-result

```
var:
   t10psi: returned = true, fresh = false, escaped = false, loop
       = false
   t9psi: returned = true, fresh = false, escaped = false, loop
       = true
   t8psi: returned = true, fresh = false, escaped = false, loop
       = true
parameters:
   [Lcom.ibm.JikesRVM.opt.ir.OPT_Operand;@122: returned = false,
      fresh = false, escaped = false, loop = false
Method is fresh? false
Total new: 0
Total stackable: 0
```

B.9 test

Java-code

```
public void test()
{
   assignStatic();
   Object b = fresh();
   assignField();
   methodCall();
   recursiveCall();
   phi(4);
}
```

HIR-code

```
-13   LABEL0 Frequency: 1.0
-2 EG ir_prologue       l0si(LTest3;,x,d) =
0   G yieldpoint_prologue
1  EG call               AF CF OF PF SF ZF = 64, virtual"Test3.
   assignStatic ()V", <TRUEGUARD>,
l0si(LTest3;,x,d)
5  EG call              l3si(Ljava/lang/Object;) AF CF OF PF SF
   ZF = 68, virtual"Test3.fresh ()Ljava/lang/Object;",
<TRUEGUARD>, l0si(LTest3;,x,d)
10 EG call               AF CF OF PF SF ZF = 72, virtual"Test3.
   assignField ()V", <TRUEGUARD>,
l0si(LTest3;,x,d)
14 EG call               AF CF OF PF SF ZF = 76, virtual"Test3.
   methodCall ()V", <TRUEGUARD>,
l0si(LTest3;,x,d)
18 EG call              t4si(Ljava/lang/Object;) AF CF OF PF SF
   ZF = 84, virtual"Test3.recursiveCall ()Ljava/lang/Object;",
<TRUEGUARD>, l0si(LTest3;,x,d)
24 EG call              t5si(I) AF CF OF PF SF ZF = 92, virtual"
   Test3.phi (I)I",
<TRUEGUARD>, l0si(LTest3;,x,d), 4
-3    return            <unused>
-1    bbend             BB0 (ENTRY)
```

Analysis-result

```
var:
   l3si: returned = false, fresh = true, escaped = false, loop =
      false
parameters:
Method is fresh? false
Total new: 0
Total stackable: 1
```

B.10 main

Java-code

```
public static void main(String[] args)
{
   Test3 t = new Test3();
   t.test();
}
```

HIR-code

```
-13   LABEL0 Frequency: 1.0
-2 EG ir_prologue       l0si([Ljava/lang/String;,d) =
0   G yieldpoint_prologue
0  EG new               t1si(LTest3;,p) = Test3
4  EG call               AF CF OF PF SF ZF = 64912, special_exact
   "Test3.<init> ()V", <TRUEGUARD>,
t1si(LTest3;,p)
9  EG call               AF CF OF PF SF ZF = 60, virtual"Test3.
   test ()V", <TRUEGUARD>,
t1si(LTest3;,p)
-3    return            <unused>
-1    bbend             BB0 (ENTRY)
```

Analysis-result

```
var:
   t1si: returned = false, fresh = true, escaped = false, loop =
      false
parameters:
Method is fresh? false
Total new: 1
Total stackable: 1
```