corrected version - 24 Feb 2005

Fast Escape Analysis for Region-based Memory Management

G. Salagnac [1,3] S. Yovine [1,3] D. Garbervetsky [2,4]

Abstract

We present an algorithm for escape analysis inspired by, but more precise than, the one proposed by Gay and Steensgaard [11]. The primary purpose of our algorithm is to produce useful information to allocate memory using a region-based memory manager. The algorithm combines intraprocedural variable-based and interprocedural points-to analyses. This is a work in progress towards achieving an application-oriented trade-off between precision and scalability. We illustrate the algorithm on several typical programming patterns, and show experimental results of a first prototype on a few benchmarks.

Key words: Escape analysis. Dynamic memory management.

1 Introduction

Garbage collection (GC) [14] is not used in real-time embedded systems. The reason is that temporal behavior of dynamic memory reclaiming is extremely difficult to predict. Several GC algorithms have been proposed for real-time embedded applications (e.g., [12,17,16,13]). However, these approaches are not portable (as they impose restrictive conditions on the underlying execution platform), do require additional memory, and/or do not really ensure predictable execution times.

An appealing solution to overcome the drawbacks of GC algorithms, is to allocate objects in regions (e.g., [18]) which are associated with the lifetime of a computation unit (typically a thread or a method). Regions are freed when the corresponding unit finishes its execution. This approach is adopted, for

[1] *VERIMAG, Centre Equation, 2 Ave. Vignate, 38610 Gières, France.* E-mail: `firstname.lastname@imag.fr`.

[2] *School of Computer Science, Universidad de Buenos Aires, Argentina.* E-mail: `diegog@dc.uba.ar`.

[3] Partially supported by projects DYNAMO (Min. Research, France) and MADEJA (Rhône-Alpes, France).

[4] Partially supported by projects ANCyT grant PICT 11738 and IBM Eclipse Innovation Grants.

there are typographic errors in the AIOOL'05 version instance, by the Real-Time Specification for Java (RTSJ) [2], where regions can be associated to runnables, and by [10], which implements a library and a compiler for C. These region-based approaches define APIs which can be used to explicitly and manually handle allocation and deallocation of objects within a program. However, care must be taken when objects are mapped to regions in order to avoid dangling references. Thus, programming using such APIs is error-prone, mostly because determining objects' lifetime is difficult.

An alternative to programming memory management directly using an API consists in *automatically* transforming a program so as (a) to replace (whenever possible) "new" statements by calls to the region-based memory allocator, and (b) to place appropriate calls (i.e., guaranteeing absence of dangling references) to the deallocator. Such an approach requires to analyze the program behavior to determine the lifetime of dynamically allocated objects. In [8], analysis is based on profiling, while [9,4] rely on static (points-to and escape) analysis.

Escape analysis aims at conservatively determining if an object *escapes from* or is *captured by* a method. Intuitively, an object escapes a method when its lifetime is longer than the method's lifetime, so it can not be collected when the method finishes its execution. An object is captured by the method when it can be safely collected at the end of its execution.

Several approaches to escape analysis for Java have been proposed, most of which aim at allocating objects on the stack, and removing unnecessary synchronizations. [1] works on the bytecode, which brings in an additionnal complexity due to the stack-based model. [5,19] use points-to analysis to determine if an object escapes a method through a path in the points-to graph. [11] proposes a fast but very conservative escape analysis, based on solving a simple system of linear constraints obtained from a *Static Single Assignment* (SSA) form [7] of the program.

For region-based allocation in Java, we are aware of two works. [9] exploits method-call chains and escape analysis to dynamically map allocation sites to regions associated with methods. [4] defines a points-to analysis to determine regions of objects with similar lifetimes (with instruction-level resolution, as opposed to method-level).

In this paper, we present an algorithm for escape analysis inspired by, but more precise than, the one proposed in [11]. The primary purpose of our algorithm is to produce useful information to allocate memory using a region-based memory manager. The algorithm combines intraprocedural variable-based and interprocedural points-to analyses. This is a work in progress towards achieving an application-oriented trade-off between precision and scalability. We illustrate the algorithm on several typical programming patterns, and show experimental results of a first prototype on a few benchmarks.

2 The algorithm

In this section we describe our escape analysis algorithm in detail. We assume the program is in static single assignment form (SSA) [7], that is, every variable is assigned only once in the program. The transformation of the program into SSA comes at a cost, but gives to a flow-insensitive analysis the power of a flow-sensitive one. Our algorithm is mainly based on local variables, instead of on a complex points-to graph, which would be much more expensive to build and to work with. The analysis is based on abstract interpretation [6] and computes several properties for local variables and methods.

2.1 Properties

2.1.1 escape

For each local variable `v` of a method, escape(`v`)$\in$*Escape*, where *Escape* is the lattice in figure 1(a), says whether `v` may *escape* from its method, that is, if an object pointed to by `v` is referenced in a way such that its lifetime may exceed the method.

A variable `v` escapes because it is returned (escape(`v`)=RETURNED) or it is copied into a global variable (escape(`v`)=STATIC). When a variable is stored into an object field (escape(`v`)=FIELD), `v` may escape through a chain of references. Determining whether `v` escapes in this case, requires further analysis that will be explained later. The $\top$ value stands for variables that escape by *several* ways, or when the analysis cannot compute a tighter information (e.g., when `v` is used as a parameter in a non-analyzed method). For example, in the program shown on figure 1(b) escape(`a`)=STATIC, escape(`b`)=$\perp$, escape(`c`)=RETURNED, escape(`d`)=$\top$.

Notice that escape(`v`) $= \perp$ is not sufficient to say that the object pointed to by `v` is local to the method. It only means that the method does not create any *new* reference path from the outside of the method to the object, but the object may *already* be reachable from outside. This is the case for variable `b` in figure 1(b) which is an alias of the static variable `s`.

$$
\begin{array}{ccc}
 & \top & \\
\nearrow & \uparrow & \nwarrow \\
\text{FIELD} & \text{RETURNED} & \text{STATIC} \\
\nwarrow & \uparrow & \nearrow \\
 & \perp &
\end{array}
$$

(a) The escape lattice

```
class Test01 {
  static Object s,t;
  void m0() {
    Object a=m1();
    s=a;
    Object b=m2();
  }
  Object m1() {
      Object c=new Object();
      return c;
  }
  Object m2() {
      Object d=new Object();
      s=d;
      return d;
  }
}
```

(b) the `Test01` program

Fig. 1. The *Escape* lattice and the `Test01` program

2.1.2 mfresh

Let *MFresh* be the lattice: $\perp \leq$ RETURNED $\leq \top$. For each method `m`, mfresh(`m`) $\in$ *MFresh* describes how objects returned by `m` escape: mfresh(`m`)$=\perp$ when `m` does not return any object (it may be `void`, or return some primitive type value); mfresh(`m`)$=\top$ when returned values are already known to escape from `m` in a different way; mfresh(`m`)=RETURNED when `m` returns an object (or several objects) which does (do) not escape otherwise. If there is no other path leading to this object (see section 2.2.2), the caller of `m` can *capture* it.

2.1.3 sites

Let *Sites* be $\mathcal{P}(AllocationSites \cup \{$UNKNOWN$\})$, where *AllocationSites* is the set of all allocation sites in the program. For each local variable `v`, sites(`v`)$\in$*Sites* contains all allocation sites that can create an object referenced by `v`. sites(`v`) can always be computed at the unique (thanks to SSA) statement where `v` is defined. To be conservative, if we cannot determine *all* the sites that `v` can point to (e.g., because of a not analyzed method call), a "fake" allocation site UNKNOWN is added to sites(`v`). In the program of figure 1(b), sites(`a`) = sites(`c`) = {`[m1:c=new Object]`}, and sites(`b`) = sites(`d`) = {`[m2:d=new Object]`}.

2.1.4 msites

For each method `m`, msites(`m`) is an element of *Sites*, saying where objects returned by `m` come from. In the program of figure 1(b), msites(`m0`) = $\emptyset$, msites(`m1`) = {`[m1:c=new Object]`}, and msites(`m2`) = {`[m2:d=new Object]`}. Notice that, if mfresh(`m`) = RETURNED, then objects from msites(`m`) are possibly *captured* by callers of `m`, but it is not certain. In some complex situations, there can still be a path of references leading to these objects. For example in the program shown on figure 6(a), the `e` variable is *not* captured by `m0`.

2.1.5 isdereferenced

isdereferenced(`v`) is true iff `v`, or one of its aliases, is *dereferenced* in `m`. That is, is `v.f` appears in the right-hand side of an assignment.

2.1.6 usedasparameter

usedasparameter(`v`) is true iff `v`, or one of its aliases, is used as a concrete parameter in a method call.

2.1.7 def

For each variable `v`, def(`v`) says how `v` was defined.

2.1.8 fielduse

fielduse shows reference relations between local variables. For each `v` in `m`, fielduse(`v`) is the set of variables `u` in `m` such that `v` may be an alias of `u.f` (for some field `f`). fielduse is mainly useful when a variable `v` escapes by a FIELD:

for example, if escape(v)=FIELD, but all variables of fielduse(v) are captured by m, then so is v.

2.1.9 the mrefs graph

When objects are passed through several methods, knowledge about local variables is often not sufficient to determine objects' lifetimes, that's why a reference graph is needed. Our reference graph is very simple, in order to minimize the algorithmic cost of the analysis. mrefs is a subset of $AllocationSites \times Fields \times AllocationSites$, where $(\alpha, f, \beta) \in$ mrefs means: "an object created in α, may point, with its f field, to an object created in β".

2.1.10 side

The main goal of our analysis is to determine in which regions to allocate objects. Each method m has an associated *region*, containing objects which do not escape m. To determine the region, we compute for each variable v of m, where objects pointed to by v live, namely, side(v):

- side(v)=INSIDE, when objects pointed to by v are *captured* by m. If they are created by m, they can be allocated in m's region. If they are created by callees, m can *ask* for them to be allocated in its region, as is described in [9];
- side(v)=OUTSIDE, when objects pointed to by v live *longer* than m. If they are created by m, they must be allocated outside its stack frame. But such an object may be captured by a caller n of m, in this case m can allocate the object in n's region.

An example is presented on figure 6(a): the RefObject allocated by m2 is captured by m1. Our analysis detects this situation by computing side(a)=OUTSIDE and side(c)=INSIDE.

2.2 The rules

The algorithm works in two phases. First, it determines for each variable the values of escape, sites, isdereferenced, usedasparameter, fielduse, def, it builds the mrefsgraph and computes msites and mfresh values. To compute these values, the algorithm solves the least fixpoint in Figures 2 and 3.

In a second phase, the algorithm uses these values to compute, for each variable, its side value, as presented on figure 4. It is the combination of side and sites that will enable us to instrument the bytecode in order to use a region memory allocator for captured sites.

2.2.1 First phase

Most of these rules are simple. They are only intraprocedural information propagation. The only complicated rule is the one on figure 3, which handles method calls. This is not trivial, because we do not want to perform a full

α: `v := new`
$\alpha \in \mathsf{sites}(v)$

`v:=` φ`(`$\mathtt{v}_1$`..`$\mathtt{v}_n$`)`
$\mathsf{def}(\mathtt{v}) = \text{PHI}$
$\forall i = 1..n$
$\mathsf{sites}(\mathtt{v}) \supseteq \mathsf{sites}(\mathtt{v}_i)$
$\mathsf{escape}(\mathtt{v}) \sqsupseteq \mathsf{escape}(\mathtt{v}_i)$
$\mathsf{escape}(\mathtt{v}_i) \sqsupseteq \mathsf{escape}(\mathtt{v})$
$\mathsf{isdereferenced}(\mathtt{v}) \geq \mathsf{isdereferenced}(\mathtt{v}_i)$
$\mathsf{isdereferenced}(\mathtt{v}_i) \geq \mathsf{isdereferenced}(\mathtt{v})$
$\mathsf{usedasparameter}(\mathtt{v}) \geq \mathsf{usedasparameter}(\mathtt{v}_i)$
$\mathsf{usedasparameter}(\mathtt{v}_i) \geq \mathsf{usedasparameter}(\mathtt{v})$
$\mathsf{fielduse}(\mathtt{v}) \supseteq \mathsf{fielduse}(\mathtt{v}_i)$
$\mathsf{fielduse}(\mathtt{v}_i) \supseteq \mathsf{fielduse}(\mathtt{v})$

`v :=` $\mathtt{v}_1$
$\mathsf{def}(\mathtt{v}) = \text{COPY}$
other properties: similar to φ-expression

$\mathtt{v}_1$`.f := v`
$\mathsf{escape}(\mathtt{v}) \sqsupseteq \text{FIELD}$
$\mathsf{fielduse}(\mathtt{v}) \ni \mathtt{v}_1$
$\mathsf{mrefs} \supseteq \{\mathtt{s}_1 \xrightarrow{\mathtt{f}} \mathtt{s}_2,$
$\mathtt{s}_1 \in \mathsf{sites}(\mathtt{v}_1),\ \mathtt{s}_2 \in \mathsf{sites}(\mathtt{v}_2)\}$

`s := v`
$\mathsf{escape}(\mathtt{v}) \sqsupseteq \text{STATIC}$
$\mathsf{mrefs} \supseteq \{\text{UNKNOWN} \longrightarrow \mathtt{s},\ \mathtt{s} \in \mathsf{sites}(\mathtt{v}_i)\}$

`v := s`
$\mathsf{def}(\mathtt{v}) = \text{STATIC}$
$\mathsf{sites}(\mathtt{v}) \ni \text{UNKNOWN}$

`v := p`
$\mathsf{def}(\mathtt{v}) = \text{PARAM}$
$\mathsf{sites}(\mathtt{v}) \ni \text{UNKNOWN}$
other properties: similar to φ-expression

`v := constant`
$\mathsf{def}(\mathtt{v}) = \text{CONSTANT}$
$\mathsf{sites}(\mathtt{v}) \ni \text{UNKNOWN}$

`v :=` $\mathtt{v}_1$`.f`
$\mathsf{def}(\mathtt{v}) = \text{FIELD}$
$\mathsf{isdereferenced}(\mathtt{v}_1) \geq \mathit{true}$
$\mathsf{sites}(\mathtt{v}) \supseteq \{\mathtt{s} \mid \exists\, \mathtt{s}' \in \mathsf{sites}(\mathtt{v}_1),\ \mathtt{s}' \xrightarrow{\mathtt{f}} \mathtt{s}\ \}$
If $\text{UNKNOWN} \in \mathsf{sites}(\mathtt{v}_1)$
$\mathsf{sites}(\mathtt{v}) \ni \text{UNKNOWN}$

$\mathtt{return}_m$ `v`
$\mathsf{escape}(\mathtt{v}) \sqsupseteq \text{RETURNED}$
$\mathsf{mfresh}(\mathtt{m}) \sqsupseteq \mathsf{escape}(\mathtt{v})$
$\mathsf{msites}(\mathtt{m}) \supseteq \mathsf{sites}(\mathtt{v})$

Fig. 2. Escape analysis rules

`v :=` $\mathtt{v}_0$`.m(`$\mathtt{v}_1$`..`$\mathtt{v}_n$`)`
$\forall$ `m` that may be invoked here
If $\mathsf{istobeprocessed}(\mathtt{m})$
$\mathsf{sites}(\mathtt{v}) \supseteq \mathsf{msites}(\mathtt{m})$
If $\mathsf{mfresh}(\mathtt{m}) \neq \text{RETURNED}$
$\mathsf{escape}(\mathtt{v}) \sqsupseteq \mathsf{mfresh}(\mathtt{m})$
$\forall i = 0..n$
$\mathsf{usedasparameter}(\mathtt{v}_i) \geq \mathit{true}$
Let $\mathtt{p}_i$ the i-th formal parameter of m
$\mathsf{isdereferenced}(\mathtt{v}_i) \geq \mathsf{isdereferenced}(\mathtt{p}_i)$
If $\neg\mathsf{escape}(\mathtt{p}_i) \in \{\text{RETURNED}, \bot\}$
$\mathsf{escape}(\mathtt{v}_i) \sqsupseteq \top$
$\mathsf{mrefs} \supseteq \{\text{UNKNOWN} \longrightarrow \mathtt{s},\ \mathtt{s} \in \mathsf{sites}(\mathtt{v}_i)\}$
If $\mathsf{isdereferenced}(\mathtt{p}_i) = \mathit{true}$
$\mathsf{mrefs} \supseteq \{\text{UNKNOWN} \longrightarrow \mathtt{s} \mid \exists\, \mathtt{s}' \in \mathsf{sites}(\mathtt{v}_i),\ \mathtt{s}' \longrightarrow \mathtt{s})\}$
else
$\mathsf{sites}(\mathtt{v}) \ni \text{UNKNOWN}$
$\forall i = 0..n$
$\mathsf{usedasparameter}(\mathtt{v}_i) \geq \mathit{true}$
$\mathsf{isdereferenced}(\mathtt{v}_i) \geq \mathit{true}$
$\mathsf{escape}(\mathtt{v}_i) \sqsupseteq \top$
$\mathsf{mrefs} \supseteq \{\text{UNKNOWN} \longrightarrow \mathtt{s},\ \mathtt{s} \in \mathsf{sites}(\mathtt{v}_i)\}$

Fig. 3. Escape analysis rules (cont)

points-to analysis, neither to be too conservative about method calls.

Our analysis is designed to process arbitrary portions of an application. That is why we have an $\mathsf{istobeprocessed}$ predicate, that tells if a method must be analyzed or not. If not, for example because the method is native, or unavailable, we must be conservative about it.

For a not analyzed method, we assume that all parameters escape, and are referenced by the UNKNOWN site.

On the other hand, if the method *is* analyzed, then we can be more precise. Obviously, we have $\mathsf{sites}(\mathtt{v}) \supseteq \mathsf{msites}(\mathtt{m})$, that is, $\mathtt{v}$ will point to any object returned by $\mathtt{m}$. If these objects have escaped ($\mathsf{mfresh}(\mathtt{m}) \neq$ RETURNED), then the return value is not capturable either. ($\mathsf{escape}(\mathtt{v}) \sqsupseteq \mathsf{mfresh}(\mathtt{m})$)

To process the parameters of $\mathtt{m}$, we match the formal parameters ($\mathtt{p}_i$) with the concrete ones ($\mathtt{v}_i$): if $\mathtt{p}_i$ escapes from $\mathtt{m}$, $\mathtt{v}_i$ is considered as escaping from the current method, and we put an edge from UNKNOWN to all sites pointed to by $\mathtt{v}_i$. If $\mathtt{p}_i$ does not escape but $\mathsf{isdereferenced}$ in $\mathtt{m}$, then we cannot be precise about those references without performing a points-to analysis. In this case, we conservatively consider that all children of $\mathtt{v}_i$ escape.

2.2.2 Second phase

escape(v) \ def(v)	NEW	RETVAL	PARAM STATIC	COPY PHI	FIELD	CONSTANT
$\perp$	(3)	(3)	OUTSIDE	(1)	(2)	OUTSIDE
FIELD	(2)	(2)	OUTSIDE	(1)	(2)	OUTSIDE
RETURNED	OUTSIDE	OUTSIDE	OUTSIDE	OUTSIDE	OUTSIDE	OUTSIDE
STATIC	OUTSIDE	OUTSIDE	OUTSIDE	OUTSIDE	OUTSIDE	OUTSIDE
$\top$	OUTSIDE	OUTSIDE	OUTSIDE	OUTSIDE	OUTSIDE	OUTSIDE

(1)
$\mathtt{v} := \varphi(\mathtt{v}_1 .. \mathtt{v}_n)$ or $\mathtt{v} := \mathtt{v}_1$
$\quad \mathsf{side}(\mathtt{v}) \sqsupseteq \mathsf{side}(\mathtt{v}_i)$
$\forall i \ \mathsf{side}(\mathtt{v}_i) \sqsupseteq \mathsf{side}(\mathtt{v})$

(3)
If $\exists \mathtt{s} \in \mathsf{sites}(\mathtt{v})$ s.t. UNKNOWN $\rightsquigarrow \mathtt{s}$
$\quad \mathsf{side}(\mathtt{v}) =$ OUTSIDE
else
$\quad \mathsf{side}(\mathtt{v}) =$ INSIDE

(2)
If $\exists \mathtt{u} \in \mathsf{fielduse}(v)$ s.t. $\mathsf{side}(\mathtt{u}) =$ OUTSIDE
$\quad$ or s.t.$\mathsf{isdereferenced}(\mathtt{u}) \wedge \mathsf{usedasparameter}(\mathtt{u})$
$\quad \mathsf{side}(\mathtt{v}) =$ OUTSIDE
else
$\quad$ (3)

Fig. 4. Computation of $\mathsf{side}(\mathtt{v})$

Once the fixed point is reached, the algorithm computes $\mathsf{side}(\mathtt{v})$ for each variable using rules shown in figure 4. This is not a one-pass computation, but a second least fixpoint, because of the (1) and (2) rules:

- The (1) rule says that, if a variable may alias another, then those two variables cannot have different side values.
- Similarly, the (2) rules says that if a variable $\mathtt{v}$ is referenced by another

variable's field (e.g. by a `u.f=v`), `v` cannot be captured unless `u` is.

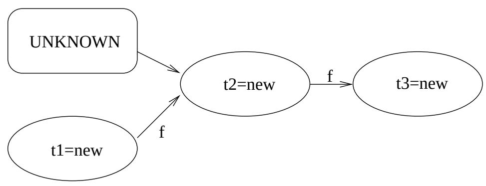
2.2.3 Examples

Let us consider the example presented in fig.5(a). First, `m0` builds a small chained structure, then it calls `m1` which makes the last element (`t3`) escape. As shown on fig.5(b), the analysis of `m0` understands the behavior of `m0`, but as we can only match `x` with `t1`, and not `a` with `t2`, we cannot keep track of `m1`. Nevertheless, to stay conservative, we put an edge from UNKNOWN to the site of `t2` because `x` is dereferenced in `m1`. Notice that, `t2` and `t3` are usedasparameter, because they are the `this` parameter of their constructor. That is why the only captured site is `[m0:t1 = new RefObject]`.

```
class RefObject {
    Object f;
}

class Test25 {
    void m0() {
        RefObject t1=new RefObject();
        RefObject t2=new RefObject();
        Object t3=new Object();
        t1.f=t2;
        t2.f=t3;
        m1(t1);
    }
    static Object s;
    void m1(RefObject x)
    {
        RefObject a=(RefObject)x.f;
        Object b=a.f;
        s=b;
    }
}
```

(a) The Test25 program



(b) mrefs graph

	escape mfresh	def	IsD	uP	fielduse	sites msites	side
m0	⊥					∅	
t1	⊥	NEW	true	true	[]	[m0:t1 = new RefObject]	INSIDE
t2	FIELD	NEW	false	true	[t1]	[m0:t2 = new RefObject]	OUTSIDE
t3	FIELD	NEW	false	true	[t2]	[m0:t3 = new java.lang.Object]	OUTSIDE
m1	⊥					∅	
x	⊥	PARAM	true	false	[]	[UNKNOWN]	OUTSIDE
a	⊥	FIELD	true	false	[]	[UNKNOWN]	OUTSIDE
b	STATIC	FIELD	false	false	[]	[UNKNOWN]	OUTSIDE

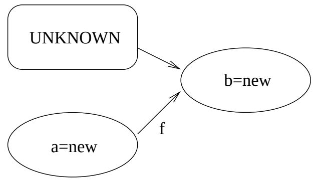
(c) analysis results

Fig. 5. The `Test25` program

The second example, shown in figure 6(a), illustrates the msites property. The `m2` method allocates two objects and makes one (`a`) point to the other (`b`), which escapes. Then it returns `a`, which is captured by `m1` (side(`c`)=INSIDE). `m1` dereferences c to get the `Object` and returns it, but `m0` cannot capture it because of the edge from UNKNOWN to `[m2:b = new Object]`.

```
class Test30 {
    void m0() {
        Object e=m1();
    }
    Object m1() {
        RefObject c=m2();
        Object d=c.f;
        return d;
    }
    static Object s;
    RefObject m2() {
        RefObject a=new RefObject();
        Object b=new Object();
        s=b;
        a.f=b;
        return a;
    }
}
```

(a) the `Test30` program



(b) mrefs graph

	escape mfresh	def	IsD	uP	fielduse	sites msites	side
m0	$\perp$					$\emptyset$	
e	$\perp$	RETVAL	false	false	$\emptyset$	`[m2:b = new Object]`	OUTSIDE
m1	RETURNED					`[m2:b = new Object]`	
c	$\perp$	RETVAL	true	false	$\emptyset$	`[m2:a = new RefObject]`	INSIDE
d	RETURNED	FIELD	false	false	$\emptyset$	`[m2:b = new Object]`	OUTSIDE
m2	RETURNED					`[m2:a = new RefObject]`	
a	RETURNED	NEW	false	true	$\emptyset$	`[m2:a = new RefObject]`	OUTSIDE
b	$\top$	NEW	false	true	`[a]`	`[m2:b = new Object]`	OUTSIDE

(c) analysis results

Fig. 6. the `Test30` program

Program	Lines	Allocation	Analysis time			INSIDE		G&S's analysis
		sites	escape	side	total	variables	sites	*stackable* variables
bh	1128	41	9.430	23.51	32.481	34	21	23
bisort	340	10	7.876	11.509	19.385	7	7	7
em3d	462	26	8.551	15.706	24.257	13	11	11
health	562	28	8.454	19.414	27.868	18	13	10
mst	473	16	8.106	14.260	22.366	8	8	7
perimeter	745	13	11.357	23.944	35.301	7	7	7
power	765	21	3.628	1.159	4.787	9	9	5
treeadd	195	11	10.876	27.539	38.415	6	6	6
tsp	545	12	11.19	30.201	41.220	7	7	7
voronoi	1000	35	12.778	66.566	79.344	34	20	31

Table 1
Analysis results

3 Empirical results

We have implemented a prototype version of this algorithm using the Soot framework [15] v.2.2.1. Table 1 presents the results of our algorithm on the Jolden benchmarks [3]. The first two column are the size of the program in lines, and the number of allocation sites. The next three columns present the time spent by our escape analysis, in seconds, not including Soot's phases:

class loading, transformation from bytecode to Jimple (Soot's three-address stackless code), and transformation into SSA form.

The last three columns give the number of inside variables and allocation sites, as computed by our algorithm, and the number of *stackable* variables, as computed by our implementation of G&S's analysis [11]. Our analysis is more precise than [11] as it subsumes all its rules. That is, all *stackable* variables in the sense of [11] are inside variables, but the converse is not true. In our experiments, we did not use any inlining of analyzed code. It is interesting to remark that without inlining, [11] does not find any stackable variable in the programs of figures 5 and 6. As noted in [11], both analyses will benefit from method inlining.

We did not have enough time to use the computed information to actually instrument the benchmarks as described in [9]. We count on doing this soon. Anyway, a preliminary implementation on another test program revealed a gain of 20% of total utilized memory, when using GC together with region-based manager, w.r.t. GC only, even if the actual region-allocated memory is about 5%.

Besides, only a subgraph of the whole call graph has been analyzed for each test case. The subgraph contains all application methods and a subset of library methods transitively invoked by the program. This explains why there are only a few allocation sites. Nevertheless, these results are interesting, because an important fraction of analyzed allocation sites are indeed computed to be captured. Our algorithm is parameterized by the set of classes to be analyzed. This allows the user to fine-tune the analysis trading precision against performance according to specific application behaviors.

Ackgnoledgements

We thank Chaker Nakhli and the anonymous referees for their helpful remarks.

References

[1] B. Blanchet. Escape Analysis for Java(TM). Theory and Practice. *ACM TOPLAS*, 25(6):713-775, 2003.

[2] G. Bollella and J. Gosling. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[3] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in java controller. In *PACT'01*, p. 280–291, Sep. 2001.

[4] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *ISMM'04*, ACM SIGPLAN Notices, Oct. 2004. ACM Press.

[5] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Stack Allocation and Synchronization Optimizations for Java Using Escape Analysis. *ACM TOPLAS*, 25(6):876-910, 2003.

[6] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, 1991.

[8] M. Deters and R. Cytron. Automated discovery of scoped memory regions for real-time Java. In *ISMM'02*, ACM SIGPLAN Notices, June 2002.

[9] D. Garbervetsky, C. Nakhli, S. Yovine, and H. Zorgati. Program instrumentation and run-time analysis of scoped memory in java, 2004. RV'04. To appear in ENTCS.

[10] D. Gay and A. Aiken. Language support for regions. In *SIGPLAN PLDI'01*, pages 70–80, 2001.

[11] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *CC'00*. Springer-Verlag, 2000.

[12] R. Henriksson. Scheduling garbage collection in embedded systems. PhD. Thesis, Lund Institute of Technology, July 1998.

[13] T. Higuera, V. Issarny, M. Banatre, G. Cabillic, J-Ph. Lesot, and F. Parain. Memory management for real-time Java: an efficient solution using hardware support. *Real-Time Systems Journal*, 2002.

[14] R. Jones and R. Lins. *Garbage collection. Algorithms for automatic dynamic memory management.* John Wiley and Sons, 1996.

[15] V. Sundaresan, P. Lam, E. Gagnon, R. Vallee-Rai, L. Hendren and P. Co. Soot - A java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[16] T. Ritzau and P. Fritzon. Decreasing memory over-head in hard real-time garbage collection. In *EMSOFT'02, Grenoble, France. LNCS 2491*, 2002.

[17] F. Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. CASES'00, 2000.

[18] M. Tofte and J-P. Talpin. Region-based memory management. *Information and Computation*, 1997.

[19] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. in *OOPSLA'99*, volume 34(10) of *ACM SIGPLAN Notices.* ACM Press.