A Verifiable Lightweight Escape Analysis Supporting Creational Design Patterns

Gilles Grimaud — Yann Hodique — Isabelle Simplot-Ryl

N° 00081200

June 2006

Thème COM

Rapport de recherche

SN 0249-6399 ISRN INRIA/RR--00081200--FR+ENG

A Verifiable Lightweight Escape Analysis Supporting Creational Design Patterns

Gilles Grimaud*† , Yann Hodique† , Isabelle Simplot-Ryl†

Thème COM — Systèmes communicants
Projets Pops

Rapport de recherche n° 00081200 — June 2006 — 20 pages

Abstract: This paper presents a compositional escape analysis adapted for use in resource limited embedded systems. This analysis covers the full Java language, including dynamic class loading. Thanks to the use of an efficient verification algorithm, small embedded systems are able to check the escape analysis information of mobile code. The traditional escape analysis is also extended, taking further steps towards full Java programming support, by adding the support of common design patterns, namely aggregation and factory, in order to allow the programmer to use coding techniques that are usually somewhat inefficient on these constrained systems.

Key-words: embedded systems, static analysis, objects, design patterns, escape analysis

* INRIA Futurs, POPS
† Université Lille 1

Une analyse d'échappement légère, vérifiable, et supportant les motifs de conception de création d'objets

Résumé : Ce rapport présente une analyse d'échappement compositionnelle adaptée aux systèmes embarqués à ressources limitées. L'analyse présentée couvre l'intégralité du langage Java, notamment le chargement dynamique de classes. Grâce à l'utilisation d'un algorithme de vérification performant, les petits systèmes embarqués sont en mesure de vérifier les informations relatives à l'analyse d'échappement associées au code mobile. Nous présentons également une extension de l'analyse d'échappement dans l'optique de mieux supporter les habitudes de programmation en Java, en ajoutant un support étendu de certains motifs de conception, comme l'agrégation et les fabriques, afin de permettre au programmeur de faire usage de techniques réputées inefficaces en environnement contraint.

Mots-clés : systèmes embarqués, analyse statique, objets, motifs de conception, analyse d'échappement

Introduction

Escape analysis is a technique that has been originally developed for functional languages (see for example [8]) to help managing the allocation of structures. It has been transposed under various forms into object-oriented languages in many works [1, 2, 6, 10]. The principle behind escape analysis is to compute that an object is unreachable (that is, there is no possibility that it may be used later) at a certain point of the program, and thus can be safely destroyed. In other words, if a garbage collection were to be triggered at that point of the program, the object would be collected. It is obvious that using this property can help optimizing memory management, by for example saving a lot of work for the garbage collector since it reduces the amount of references to be checked, and also the number of calls to this garbage collector. Moreover, concurrent threads cannot access objects that do not escape and thus synchronization mechanisms can be eliminated. All these optimizations have been shown to significantly improve performance [3, 10].

In this paper, we present an application of escape analysis in the context of embedded systems and mobile code, that takes the Java language as a basis since it is heavily used in this context. One of the most usual problem with Java in constrained environments is its weight: traditionally, embedded systems rather use simplified/degraded versions of Java (like JavaCard), which imposes an adaptation of the programmer to the platform. Ideally the programmer would simply write the code the way he likes (or is used to), while the system would produce well-adapted code. Therefore, optimizations on critical parts of the system (like memory management) are highly relevant.

The presented analysis supports the main concepts of the Java language: from the interfaces and abstract classes to the exceptions, and of the Java runtime: openness, dynamically loading classes, reflection, etc. Some of these aspects imply security concerns, and we want to be able to load mobile code that benefits from escape analysis in a secure way. Thus, the system must not delegate trust on any other party: it must compute or verify the analysis itself. Since these systems are resource limited, we have to develop efficient computation techniques. We propose an analysis that combines several aspects: it is compositional like the one of [10], verifiable like the one of [1], and adapted for object-oriented issues.

This paper is organized as follow. Section 1 introduces the problems encountered with classic escape analysis when applied to constraint environments. Section 2 presents an alias analysis well suited for mobile code and that can be verified by small embedded systems. Finally, section 3 shows how to build the escape analysis on top of this alias analysis, with little additional cost. In particular, the needed computing to fill the gap between the alias analysis and the extended escape analysis can be done entirely in the embedded system. We also present in this section some extensions of the escape analysis principle that supports object aggregation, and factory design pattern.

1 Limits of the existing escape analyses

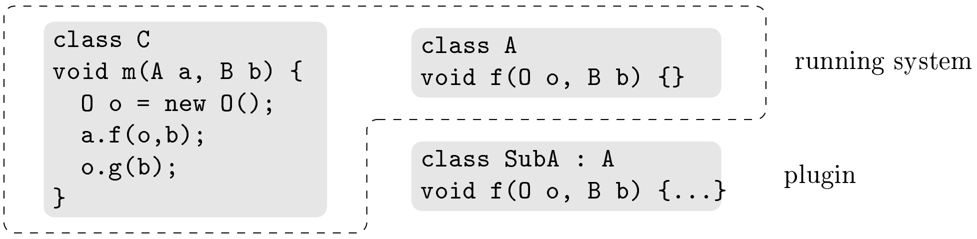
Existing works on escape analyses are applicable under different conditions and have different goals. For example, the statically analyzed code may be high level source code (mostly Java in our context, even if most escape analyses operate on functional languages) or a low-level representation (usually bytecode). A more important difference is the support for dynamic class loading: most analyses consider a closed-world model, so that the complete call graph of a program can be statically computed, which makes it easier to perform a very precise analysis. On the contrary, some works like [1] consider a fully open world, at the cost of precision. Also, the analysis scheme may be local or distributed: either the execution site has the resources (time and power) to perform the analysis on demand, or it needs precomputed results (using for example proof-carrying code[7]) to keep its power for the real execution.

Statically computing objects properties introduces approximations due to abstract interpretation. Therefore it is mandatory to make sure that these approximations produce safe results. In the introduction, we presented escape analysis as a convenient optimization for memory management. It follows that it is always safe (though potentially suboptimal) to mark objects as escaping. Our analysis will focus on *captured* objects, and the results will be defined as *correct* if no object marked as *captured* is a false positive.

Definition 1 (Captured Object) *An object o is said to be captured in the scope of a method m if it is created in m and at the end of the execution of m there exist no more reference to o.*

In the context we address, it is unacceptable to consider a closed world since runtime extensibility is a key feature of object-oriented systems, which moreover is absolutely needed in ubiquitous computing. The reason for that is the need to load mobile code and to share code between applications (APIs for example, to avoid replication of code that would lead to excessive memory consumption), which breaks the inherent isolation between code from different producers. Therefore, the analysis cannot be performed entirely at compilation time to take into account any valid extension. Figure 1 shows a trivial example of the situation we want to be able to manage. Looking at the code, it is obvious that there is only one way for the object `o` to survive method `m()`: a reference to `o` has to be kept "outside" during the execution of `f()`. A system running with classes `C` and `A` should be able to dynamically load the class `SubA`, provided that it does not compromise the integrity of the system. An open-world approach (for example [1]) would make the object `o` escape in any case, because it is not possible to statically know what happens in the method `f()`, whereas a closed-world approach would detect that `o` is captured (since the body of the only one available method `f()` is empty), but would prevent from loading `SubA`.

The late part of the analysis is a problem since it implies an overhead when loading the code into the system. Many choices can be made in order to decrease the cost of the analysis, like having recourse to approximations. For example, the complete tracking of object attributes like in [2] would be too expensive, and it may be cheaper to unify an object with its fields during the analysis, to keep space consumption at an acceptable level. Some



```
class C
void m(A a, B b) {
  O o = new O();
  a.f(o,b);
  o.g(b);
}
```

```
class A
void f(O o, B b) {}
```

```
class SubA : A
void f(O o, B b) {...}
```

Figure 1: Dynamic loading issue.

other related works like [4] choose to perform a flow-insensitive analysis instead, for the same reason. But such approximations are not sufficient due to the inherent complexity of the abstract interpretation algorithms involved (they imply non-trivial fix point computation), and the major source of gain is to perform most of the analysis off-line, to decrease the overhead at loading time. This, combined with the need for security that was previously exposed, brings us naturally to use some kind of proof-carrying code[7] strategy.

In this case the proof-carrying code scheme does not appear as a way to distribute a real first order proof of the mobile code between the code producer and the code consumer. We just try to distribute the complexity of an escape analysis abstract interpretation. As mentioned before, escape analysis is often based on a non-trivial fix point computation. Fix point computation on a java method like the one involved in bytecode type checking can be efficiently performed in an embedded system when an already computed fix point is joined to the mobile code. In this case the embedded system just checks the code with the final fix-point. The type checker, like the one proposed by [9], enforces both the bytecode typing correctness and the validity of the proposed fix-point in a linear analysis (more precisely, each bytecode is checked exactly once). Therefore, we will design our algorithm in such a way that its results are verifiable by the same mechanism.

2 Alias analysis

2.1 Preliminary and definitions

Our goal is to present an escape analysis that needs an underlying general purpose alias analysis. In this section we present a lightweight and verifiable alias analysis for Java bytecode. This analysis is split in two parts (off-line and on-line) that enable the analysis to be performed by the code producer, while the code consumer only has to verify the integrity of the results.

First, we expose the theoretical basis of our analysis, and then the algorithm itself. Finally, we discuss some aspects of the analysis and its verification.

In the following, O denotes the set of all the objects that can be used in a Java program. When dealing with aliases, our goal is to compute as precisely as possible the image of O by the following relation.

Definition 2 ("Pointed by" relation) *Let $\hookrightarrow : O \rightarrow O$ be the "pointed by" relation. The relation $o \hookrightarrow f$ reads "f is pointed by o" and means that f is a reference contained in a field of o.*

Note: to simplify the model, an array is considered as an object with a potentially infinite number of fields, therefore an array also "points to" its elements.

Definition 3 (Reachability) *We denote by $\hookrightarrow^*$ the transitive closure of $\hookrightarrow$ and $o \hookrightarrow^* f$ reads "f is reachable from o".*

2.2 Abstraction

To perform a static analysis of a program we need an abstraction of the relation $\hookrightarrow$ that deals with abstract objects instead of objects directly, because one cannot decide statically which real object is used at runtime. In the rest of the paper, we will also use the term "references" to designate those abstract objects, since we never consider real objects in the static analysis. The set of used references is composed of various subsets, described as follows.

Definition 4 (References for a method) *Let m be a method, then the set of references for m is $Ref_m = Para_m \cup \{Ret_m\} \cup Except_m \cup Alloc_m \cup Func_m \cup \{Static\}$ where:*

- *$Para_m$ represents the set of parameters of m,*
- *Ret_m the return value of the method,*
- *$Except_m$ the exceptions to be thrown (they are unified in our abstraction),*
- *$Alloc_m$ the set of allocation sites in the method,*
- *$Func_m$ the set of function calls returning values in the method,*
- *and $Static$ an abstraction of the static objects.*

An important subset is $Alloc_m$, that represents all the objects that may be created during the method execution. We consider one reference for each `new` bytecode (the same goes for other `new`-like bytecodes). Therefore we will call r_N the reference that represents all the objects that are created executing the N^{th} bytecode.

Definition 5 (Object abstraction) *Let $o \in O$ and $\breve{o} \in Ref_m$. We note $o \propto \breve{o}$ if $\breve{o}$ is an abstraction of o.*

We also note $\breve{O}$ the set of abstract objects.

`putfield` : `o.f = x`

$$\frac{L \qquad o.f = x}{L' = closure(L\{r_1 \overset{\alpha}{\hookrightarrow} r_2 : r_1 \in \bar{o} \cup \{y \mid o \overset{\alpha}{\hookrightarrow}^* y\}, r_2 \in \bar{x}\})}$$

`invokeVirtual` : `x = o.f(p1,...pn)`

$$\frac{L \qquad o.f(\overrightarrow{p}) \qquad S = exactType(o)?exactSign(f) : approxSign(f)}{L' = closure(L\{\pi_{\overrightarrow{p}}(L) \star S\})}$$

Table 1: Bytecode semantics excerpt.

Obviously the relations described in Definitions 2 and 3 cannot be statically computed, so that we focus on an abstraction of those relations.

Definition 6 ("Pointed by" relation abstraction) *The abstraction of the "pointed by" relation is its modulo on the domain of abstract objects. It is defined as follows:* $\breve{o} \overset{\alpha}{\hookrightarrow} \breve{f} \iff \exists o, f \; : \; o \propto \breve{o} \wedge f \propto \breve{f} \wedge o \hookrightarrow f$.

Definition 7 (Abstract reachability) *As previously,* $\overset{\alpha}{\hookrightarrow}^*$ *denotes the transitive closure of* $\overset{\alpha}{\hookrightarrow}$.

Our algorithm aims at computing alias relations that provide a safe basis for more complex analyses (like escape analysis), so that the information we focus on expresses the "links" between references. But still, the $\overset{\alpha}{\hookrightarrow}^*$ relation is not statically computable (due to control flow), so that we have to approximate it. A central structure in our algorithm the link array. It is meant to describe (at a given time, or step in the algorithm) the relation between references (in the sense of the $\overset{\alpha}{\hookrightarrow}^*$ relation).

Definition 8 (Link array) *A link array is a part of* $Ref_m \times Ref_m$.

We note $Link_m$ *the set of link arrays for a method* m *(*$Link_m$ *is the set of parts of* $Ref_m \times Ref_m$*).*

Definition 9 (Link array correctness) *A link array* A *is said to be correct regarding its context if* $\forall f, t \in \breve{O} \; : \; f \overset{\alpha}{\hookrightarrow}^* t \Rightarrow (f, t) \in A$.

Note that there is only a one-way implication between the description stored in the link array and the relation between references. This is due to necessary approximations in our algorithm.

2.3 Algorithm

Our algorithm is divided into two major aspects: the inter-method analysis and the intra-method one.

2.3.1 Inter-method analysis

Since we consider an open world (and will not re-analyze all the dependent classes when loading a new one), we cannot rely on a complete call-graph for our analysis. Instead, some methods may be loaded *a posteriori*, which forces us to analyze a piece of code independently of its calling context. We use persistent informations to describe the behavior of a method, in order not to be forced to re-analyze its body when it is invoked.

A signature is associated to each method, which describes as precisely as possible the links that are created by calling it. The signature is a transformation to be applied over a concrete vector of arguments to obtain an overestimation of the created links without the need to re-analyze the method in a new context.

Definition 10 (Visible links and Signature) *Vis_m is the set of visible references for a method m, that is the references that have an existence outside the method (they correspond mostly to the formal parameters of m), thus excludes working objects.*

$VLink_m$ is the result of the projection of $Link_m$ over $Vis_m \times Vis_m$.

A signature for a method m is a particular $VLink_m$ that represents the visible links created by the complete execution of the body of m.

2.3.2 Signatures and typing

By analyzing the bytecode, we compute a signature associated to each method: this is what we call the *exact signature*. Obviously, this signature can be applied when we know statically which method is called: this is the case for example with the `invokeStatic` bytecode. When dealing with `invokeVirtual`, it is not that trivial: one has to know the *exact type* of a reference to be able to apply the correct signature. For example, after instruction `Object o = new String();` the object `o` is known to be a `String`, therefore at this point of the program any call to an `Object` method could be resolved statically, even if an `invokeVirtual` bytecode is generated.

Definition 11 (Exact type) *An object (hence a reference) is of an exact type if one can statically decide what constructor was used to create the instance. This property is described by the following predicate: $exactType : Ref_m \to bool$.*

Exact types have been used in [1] to compensate the lack of precision introduced by method calls, due to inheritance. It states that under some conditions, one is able to compute statically the exact code that is invoked, so that inheritance is not a problem. On the other hand, when those conditions are not fulfilled, a less precise signature needs to be applied.

Definition 12 (Signature order) *Let $s_1, s_2 \in Sign$ two signatures for the same method.*

$s_1 \leq s_2 \iff \forall\ v_1, v_2 \in Vis_m, \forall\ l : VLink_m, (v_1, v_2) \in s_1(l) \Rightarrow (v_1, v_2) \in s_2(l)$.

$\top_m$ and $\bot_m$ are two special signatures defined by: $\forall\ v_1, v_2 \in Vis_m, \forall\ l\ :\ \begin{cases} (v_1, v_2) \in \top_m(l) \\ (v_1, v_2) \notin \bot_m(l) \end{cases}$

The signatures for single method m form a lattice. The most pessimistic signature for this method is $\top_m$, and the most optimistic signature is $\bot_m$. With those definitions, we introduce the notion of *approximate signature*, which is a *supremum* of the exact signatures computed for an inheritance sub-tree. This signature is used when no exact type can be computed statically (for example references obtained as return values).

The structure that fills the gap between the inter-method analysis and the intra-method one is called a *dictionary*. Its goal is to hold the signatures associated to the callable methods.

Definition 13 (Dictionary) *Let $Func$ be the set of Java methods in a program, and $Sign$ be the set of possible signatures. A dictionary is a mapping $d : Func \rightarrow (Sign \times Sign)$.*

In a dictionary, the exact and approximate signatures are accessed *via* the following projections:

$$exactSign : Func \rightarrow Sign \tag{1}$$

$$approxSign : Func \rightarrow Sign \tag{2}$$

2.3.3 Intra-method analysis

The algorithm is iterative: starting from the most optimistic signature $\bot_m$ we analyze the bytecode by following the control flow and applying the previously presented signatures when an invocation is performed. As stated in Definition 10, the signature of the method corresponds to the visible links that are created during the method body.

Each standard Java bytecode is given a semantic to express the transformation applied on the state of links when in is encountered in a program. Most bytecodes are given a trivial (neutral) semantic, since they do not play a role in the links creation process. Table 1 gives the semantic for the main actions. Several operators and notations are used in the semantic rules:

- $closure(L)$ ensures that the link array L remains transitively closed,
- $\pi_{\vec{p}}(L)$ denotes the projection of L on the vector of effective arguments,
- $I \star S$ is the composition operator, that propagates the links of S from the initial situation I,
- $\bar{v}$ represents the abstract content of the variable v.

The algorithm stops when reaching a fix point for the state of the link array at return points (we note $Link_m^r$ this link array for a method m), and its projection on the visible links of m gives us the exact signature for the method.

2.3.4 Verification

Once computed, the signatures must be made available for the execution engine, so that it can take advantage of them. Actually, two informations must be transmitted: the signature itself and, more important, enough data for enabling the system to verify its correctness. In order to achieve this, we rely on a lightweight bytecode verification[9]. The principle is to run a similar analysis in the embedded system, while avoiding the complexity of the algorithm by keeping the final states of the algorithm at label points (so that there is no need for unification).

2.4 Implementation details and choices

2.4.1 Approximation

An approximation was hidden in the semantic of the `putfield` bytecode used in Table 1. One key point in references aliasing is how precise we wish to be regarding fields: one can imagine being as precise as "this reference is contained in field `x` of field `y` of field `z` of that object", following works like [2]. But there are at least two drawbacks to this method. First, we want to provide informations that are verifiable by small embedded systems, and that kind of description can't be handled without heavy treatments. Moreover, the dictionary must be kept during the lifetime of the system, so it must be as small as possible. In addition, keeping such precise informations would constrain the inherited methods in the bounds of those links (an inherited method could not create other links than the ones imposed by the base method), which would closely bind a semantic to implementation details. Instead of that, we decided to unify all the fields of an object: whenever a field is accessed, we pretend the object itself is accessed. It is a big approximation, but a very convenient one, both in terms of efficiency and simplicity. Similarly, when accessing an object, every field of this object is considered as accessed, for the very same reason. By relaxing constraints on implementations, we gain flexibility. Some other related works like [4] make different choices, remaining very precise with the object fields but losing inter-method precision by performing a flow-insensitive analysis.

Therefore, when updating the link array, some additional (and maybe counter-intuitive) links are created to reflect the approximation. Considering those additional elements in the link arrays is the simplest way to obtain a correct result while keeping the quantity of information as low as possible.

2.4.2 Extra structures

Other structures are needed during the analysis, such as an abstraction of the execution stack. Those structures are needed only for type computation and also to determine the exact moment when the analysis has reached its fix point. They do not appear in the semantics briefly described in Table 1 in order to keep things as clear as possible.

```
public static void chain(c head, int n) {
  c p = head;
  while(n-- > 0) {
    c x = new c();
    p.link = x;
    p = x;
  }
}
```

Figure 2: Alias example.

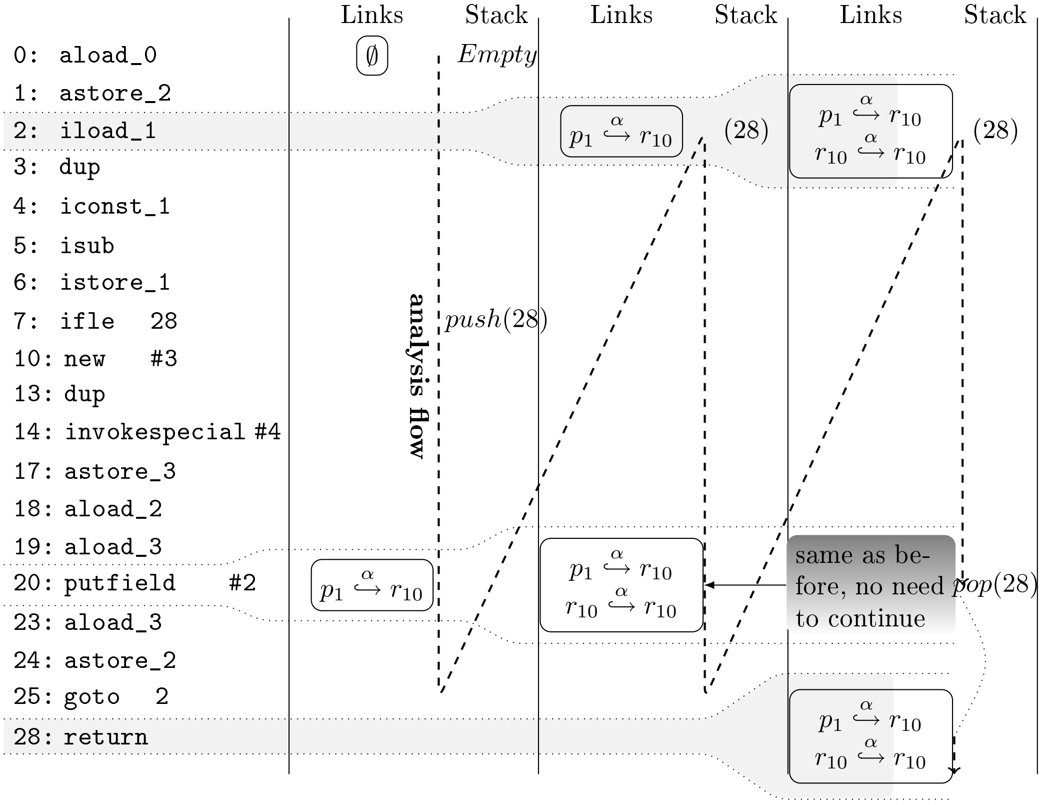

Links
Stack
Links
Stack
Links
Stack
0: aload_0
1: astore_2
2: iload_1
3: dup
4: iconst_1
5: isub
6: istore_1
7: ifle 28
10: new #3
13: dup
14: invokespecial #4
17: astore_3
18: aload_2
19: aload_3
20: putfield #2
23: aload_3
24: astore_2
25: goto 2
28: return
$\emptyset$
Empty
analysis flow
push(28)
$p_1 \overset{\alpha}{\hookrightarrow} r_{10}$
(28)
$p_1 \overset{\alpha}{\hookrightarrow} r_{10}$ $r_{10} \overset{\alpha}{\hookrightarrow} r_{10}$
(28)
$p_1 \overset{\alpha}{\hookrightarrow} r_{10}$
$p_1 \overset{\alpha}{\hookrightarrow} r_{10}$ $r_{10} \overset{\alpha}{\hookrightarrow} r_{10}$
same as before, no need to continue
pop(28)
$p_1 \overset{\alpha}{\hookrightarrow} r_{10}$ $r_{10} \overset{\alpha}{\hookrightarrow} r_{10}$

Figure 3: Algorithm run.

2.4.3 Signatures

Please also note that the computation of the signature of a method depends from the signatures of all the methods that are invoked in this method. When a method `f` is still unknown, we give it the signature $\bot_f$ when calling it, which is *not* correct. But when the method `f` is finally analyzed, the result that depend from its signature are invalidated, so that they require a new analysis. It is important to note that the algorithm may associate an *incorrect* signature to each method until it stops (when a fix-point is reached). On the other hand, when dealing with a method f that are unknown and will not be analyzed (for example a new native function, that has no bytecode), the only safe signature to be applied is $\top_f$.

2.5 Example

We present a detailed example of our analysis. The source code given in Figure 2 shows a very simple linked-list construction. The bytecode and the complete execution of our alias analysis is given in Figure 3. This array illustrates the fact that when following the control flow, it appears statistically better to go first to the lowest labels, so as to minimize the number of analysis steps applied to each bytecode. In our simple example, the only real link creation occurs with the `putfield` bytecode, which corresponds to the linkage of a cell in the linked-list (`p.link = x`).

As for the verification part, the informations highlighted in Figure 3 are sufficient since the bytecodes at offsets 2 and 28 are the only targets of jumps. Therefore, the verification informations will be the link arrays of the third column for those bytecodes. It is trivial to see that with these informations, the system can check the correctness of the signature by analyzing each bytecode only once.

3 Escape Analysis and extension

We propose in this section an escape analysis based on the previously described alias analysis. This escape analysis is verifiable by an embedded system and benefits the openness features of the alias analysis. Moreover, we present an extension of traditional escape analysis that supports creational design patterns like containers and factories. For each pattern, the result can easily be infered from the alias analysis. Thus, an embedded system is able to compute the result as soon as it has verified the alias analysis without any help from another party.

3.1 Escape analysis

3.1.1 Solution

The goal of the escape analysis is to detect "captured" objects in the sense of Definition 1.

Definition 14 *Let m be a method. The set $\mathcal{C}_m$ of* captured *objects of m is defined by:* $\mathcal{C}_m = \{o \mid o \in Alloc_m, \forall (o', o) \in Link^r_m, o' \in Alloc_m\}$. [1]

Note: due to the approximations of our alias analysis exposed in Section 2.4.1, if one of its fields may escape, then the entire reference does escape.

[1] $o' \in Alloc_m$ is sufficient, since if o' escapes, then $\exists o'' \notin Alloc_m$ such that $(o'', o') \in Link^r_m$, hence $(o'', o) \in Link^r_m$, which would be a contradiction.

3.1.2 Implementation

We compare in Table 3.1.2 our algorithm with the one presented in [1], which defines a verifiable escape analysis suitable for open-world environment. We obtain better percentages for every benchmarks, which can be explained by two main facts. First, by working on bytecode, some "hidden" allocations of Java are taken into account. But the most important point is that our approximation regarding non-exact types is far less pessimistic. The two set of columns "Dynamic Benchmarks" and "Dynamic API" separate the allocation sites that are present in the source code of the benchmarks from the ones that correspond to objects instantiated in the API (which is mainly based on GNU Classpath).

We propose an implementation of our escape analysis on the Java Virtual Machine `JITS` which is specifically designed to be usable in a range of environments that includes very small and resource-constrained devices. We made some implementation choices to deal with loop allocations and exceptions:

Some previous works [6, 10] make implementation choices that enables them to allocate as many objects as possible in the pool of local variables for performance reasons. Nevertheless the constraints of the Java runtime environment imposes that the size of one frame in the execution stack has to be statically computable at any point of the program. Obviously that would be false if we were to allocate new objects in the stack within a loop. To avoid such problems, the choice has been made to separate stack allocations from the main execution stack, in order to be able to allocate in the stack in any situation. This allows us to witness great runtime gains most benchmarks.

Exceptions in their generality are a big problem for escape analysis. Any object that is somehow attached to an exception should be marked as escaped, since there is no possibility to ensure statically whether an exception is caught in the method or not. We chose to prevent exceptions from making attached objects escape, and to compensate by moving at runtime stack allocated objects into heap, arguing that exceptions should be exceptional enough for the overhead to be acceptable in regard of the benefit that arises in regular cases.

3.1.3 Limits of the traditional analysis

The escape analysis marks objects that do not escape and allows the virtual machine to allocate them in a stack. This technique gives good runtime results and has been studied for a long time. Still, there are some limitations and even common coding style can defeat the escape analysis in frustrating ways.

First, we consider the aggregation mechanism, which is heavily used in the Java standard API, and often leads to situations where some fields of an object are also objects that are allocated in its constructor. More generally, the Java API provides many "container" classes, that is, classes whose role is to aggregate collections of other objects. Those containers include for example the `Collection` hierarchy, but also the streams and files abstraction, that keep an internal representation that is closer to the system. Typically these internal structures have no reason to be exported outside the container, thus they could be allocated in stack whenever the main object itself is allocated in stack. Anyway, these fields are never

Benchmarks		S.A. = Stack Allocations, T.A. = Total number of Allocation									
Source	Name	Static				Dynamic Benchmarks			Dynamic API		
		[1]	S.A.	T.A.	%	S.A.	T.A.	%	S.A.	T.A.	%
	Dhrystone		9	13	69%	20004	20011	100%	247	3902	6%
Java Grande	euler	26%	20	47	43%	6427102	6531342	98%	2590	34027	8%
	fft	63%	12	14	86%	4	6	67%	184	3115	6%
	heapsort	40%	4	6	67%	2	4	50%	140	1664	8%
	lufact	38%	6	11	55%	2	9	22%	171	4067	4%
	raytracer	11%	33	56	59%	426746	427224	100%	863	11408	8%
	series	80%	10	12	83%	5	9	56%	144	1951	7%
	crypt	27%	9	16	56%	30	37	81%	162	4277	4%
	moldyn	29%	3	8	38%	3	10	30%	2689	44213	6%
	search	42%	15	28	54%	7321078	7321091	100%	306	7694	4%
	sor	40%	4	6	67%	3	5	60%	129	1752	7%
	sparsematmult	25%	4	12	33%	3	11	27%	137	1954	7%
spec	check		103	122	84%	21	275	8%	1098	16538	7%
	compress		13	28	46%	20	73	27%	557	8624	6%
	raytrace		58	134	43%	5533239	6277696	88%	29092	133816	22%

Table 2: Figures from usual benchmarks extracted with our analyzer and executed on the JITS runtime environment.

stack allocated using traditional escape analysis since they are created in the constructor of the container and are used in other methods. Another popular design pattern makes objects escape: the factories.

Let us consider the example of Figure 4. The StringBuffer used to compute name = "name: "+ s ; in the class Student can be captured by the escape analysis and can be stack allocated. One of the fields of the StringBuffer class is an array which is allocated in the constructor. Thus, maybe this field could also be stack allocated (if it does not escape anywhere else), which cannot be detected by traditional escape analysis. For the same reason, if one created a Student object which does not escape, the Vector created in the constructor of the class Student (and its aggregated array) could also be stack allocated. The last field of the class Student we have to consider is a, of class Address. This field is somehow created in the same conditions that the field book_list. The difference consists in the creation method: the field a is created by a factory.

```
class Student {
  private String name;
  private Address a;
  private Vector book_list;
  public Student (String s, int n){
    name = "name: "+ s ;
    a = FactAddress.getInst().createAddr(n);
    book_list = new Vector();
  }
}

class Address {
  private int number;
  public Address (int n){ number = n;}
}

class FactAddress{
  private  static FactAddress f;
  private FactAddress () {f = this ;}
  public static FactAddress getInst() {
    if (f == null) f = new FactAddress();
    return f;
  }
  public Address createAddr (int n){
    return new Address(n);
  }
}
```

Figure 4: Example of use of creational patterns.

To propose a complete analysis of objects that can be allocated in stack, we have to consider aggregation and factory creational patterns. We propose an extension of our escape analysis considering these patterns in the rest of the paper.

3.2 Escape analysis extension

3.2.1 Solution

We have to modify our analysis to consider creations in constructors and factories. Due to the naming specifications of Java, constructors are easy to detect: constructors are the `<init>` methods. Factories are much more difficult to detect. A simple way could be to define naming conventions but this would potentially reduce the number of detected factories, especially in an open world where loaded code comes from anywhere. Thus, we decide to choose a very simple definition: we treat as a factory any method that returns an object that is not attached to anything else. We call such objects *fresh* objects using the same terminology as[6]. From our definition, a call to a factory is quite similar as a `new` bytecode: it returns a *newly allocated* object. In the example of Figure 4, objects of class `Address` may be created using the factory `FactAddress`. The method `createAddress` returns a newly allocated object whose reference is not attached to anything and is just returned: this method will be detected as a factory.

We now propose an extension of the basic mechanism of our escape analysis in order to detect the allocations in the constructors of objects and factories, and to be able to allocate them in the stack if the object itself is allocated in the stack. Therefore, it is a *conditional* stack allocation, which occurs only under conditions computed at *runtime*. Anyway, we propose a solution which is still based on a static analysis. The main issues to take care about are the following:

1. We have to determine allocation site candidate for conditional stack allocation.
2. The order to perform the conditional allocations in stack in constructors or factories will be given by the calling method, so we need an information flow from caller to callee, which is a pure runtime information.
3. We have to consider nested constructors and factories to maximize the number of objects effectively allocated in stack at runtime.

3.2.2 Algorithm

First we have to detect allocation sites that are candidate for conditional allocation.

Definition 15 *Let m be a method. The set of conditional allocation sites of m is defined by: $\mathcal{A}_m = \{o \mid o \in (Alloc_m \setminus \mathcal{C}_m), \forall (o', o) \in Link^r_m : o' \in \{\alpha\} \cup Alloc_m \cup Func_m\}$, where α =* `this` *if m is an* `<init>` *method, and Ret_m otherwise.*

Then, we have to detect in each method the places where constructors or factories must be called at runtime with the order to perform conditional allocations effectively in stack.

For a method m, the escape analysis algorithm computes the set $\mathcal{C}_m$ of captured objects. The `invokespecial` bytecodes used to invoke the constructors of the captured objects must be called so as to perform conditional allocations in stack. This is true because of the approximation we presented earlier, in Section 2.4.1: whenever a field of an object escapes, then the whole object is considered as escaping. This implies that if an object is marked as captured, then all his fields are "captured" as well, which means that the calling context does not make them escape. The only remaining thing to do is to associate each allocation with the corresponding constructor call. This can be done during the analysis by marking for each `invokespecial` bytecode, on which object it is called. Moreover, the factories invoked in m to create objects that do not survive the execution of m must perform conditional allocations in stack as well.

Definition 16 *Let m be a method. The `invokespecial` bytecodes only used to construct objects that belongs to $\mathcal{C}_m$ must be marked as* invocation with stack allocation order*. The set of allocations by factories that do not escape is defined by: $\mathcal{CF}_m = \{f_N \in Func_m \mid \forall(o, f_N) \in Link^r_m, o \in Alloc_m \cup Func_m\}$, the corresponding invocation of the factory on Nth bytecode is marked as* invocation with stack allocation order*.*

The last thing we have to consider is the possibility to use nested constructors and factories. For example, in a constructor, one of the fields can be created using a factory: the call to the factory cannot be marked as *invocation with stack allocation order* since the way the allocations must be performed in the factory depends on the fact that the constructor itself is called with the stack allocation order or not. We have to introduce a last mechanism which allows us to transmit what is called *transmission of stack allocation order.*

Definition 17 *Let m be a method. For each $o \in \mathcal{A}_m$, the constructors of o (from the class of o to `Object`) must be called with the* transmission of stack allocation order*.*

For each object $f_N \in Func_m \setminus \mathcal{CF}_m$ such that $\forall(o, f_N) \in Link^r_m, o \in \{\alpha\} \cup \mathcal{A}_m \cup Alloc_m \cup Func_m$, the corresponding invocation of the factory on Nth bytecode is marked as invocation with transmission of stack allocation order*. As previously, $\alpha =$ `this` if m is an `<init>` method, and Ret_m otherwise.*

3.2.3 Implementation

The escape analysis extension is also implemented on top of our alias analysis. In fact, it refines the standard escape analysis by providing static informations to be used at runtime.

As already said, we implement our analysis on a Java Virtual Machine which has two stacks: the traditional stack and one containing stack allocated objects. Each time a method is invoked a new frame is created in the object stack, to hold the objects statically or dynamically marked as captured. When the method returns, the statically marked objects can be destroyed, while the dynamically marked ones have to be integrated into the calling frame (since they survive the method) to be destroyed with their "container".

For this purpose, we have modified the descriptor of the frame (or context) that is created for each method invocation, by adding a flag to it. When the flag is `true`, then the conditional stack allocations that take place during the execution of the method have to be performed in stack, when it is `false`, they are performed in the heap.

Mostly for clarity and semantic simplicity, we have decided to represent the exploitation of the analysis with new bytecodes. Since those bytecodes only exist from the loading of the code, they have no real existence, and can be as well compiled as native code in the system. The informations given by our extended escape analysis are exploited by our JVM in the following way. When it loads a class file, it replaces some of the bytecodes by new ones that have a new semantics regarding to stack allocation:

- The bytecodes `newif`, `newarrayif`, `anewarrayif`, and `multianewarrayif` respectively replace `new`, `newarray`, `anewarray`, and `multianewarray` when they are marked as *conditional stack allocation*. They perform the allocation in stack when the flag of the current frame is `true`, in heap otherwise,
- The bytecodes `invokespecialTrue`, `invokevirtualTrue`, `invokeinterfaceTrue`, and `invokestaticTrue` repectively replace `invokespecial`, `invokevirtual`, `invokeinterface`, and `invokestatic` when they are marked as *invocation with stack allocation order*. They have the same semantics except that they set the flag to `true` in the new frame when the original bytecodes set the flag to `false`,
- The bytecodes `invokespecialCopy`, `invokevirtualFlag`, `invokeinterfaceFlag`, and `invokestaticFlag` respectively replace `invokespecial`, `invokevirtual`, `invokeinterface`, and `invokestatic` when they are marked as *invocation with propagation of the stack allocation order*. They have the same semantics that the original bytecodes except that they copy the current frame flag value in the new frame.

3.2.4 Experimental results

To illustrate our contribution we chose to have a look at something that is more representative of this pattern. The book "Design Patterns"[5] presents the factory pattern with a simple example for C++ that we translated to Java. This example creates mazes from various objects (rooms, walls, doors,...) by using factories. In particular, the `CreateMaze` of the main class (`MazeGame`, see Figure 5) is the starting point for the creation of a complex `Maze` and performs many calls to factories. The escape analysis would mark all the objects created in the factory methods as simply escaping. Our analysis, however, marks them as stack allocatable under some conditions. For example, with the `Main` class described in Figure 5, the entire `Maze` is allocated in stack, including all its components. In the general case, we have no guaranty that any object marked as conditionally stack allocatable will actually fill the conditions, but this information can only be a benefit, since it comes in addition to standard escape analysis (it cannot lead to a decreasing number of stack allocations).

Conclusion

In this paper we have presented a modular escape analysis, based on an alias analysis, and well suited for use in resource limited embedded systems. This escape analysis is sound and takes into account the entire Java language, including dynamic class loading, since it does not require the call graph of the program.

In addition, we have presented two extensions of the escape analysis that intend to treat fairly common design patterns (aggregation and factories), that traditionally defeat escape

```
public class MazeGame {
  public Maze CreateMaze () {
    Maze aMaze = MakeMaze();
    Room r1 = MakeRoom(1);
    Room r2 = MakeRoom(2);
    Door theDoor = MakeDoor(r1, r2);
    aMaze.AddRoom(r1);
    aMaze.AddRoom(r2);
    r1.SetSide(Room.East, theDoor);
    r2.SetSide(Room.West, theDoor);
    return aMaze;
  }
  public Maze MakeMaze() {
    return new Maze();
  }
  public Room MakeRoom(int n) {
    return new Room(n);
  }
  public Door MakeDoor(Room r1, Room r2) {
    return new Door(r1, r2);
  }
  public static void main(String[] args) {
    MazeGame mg = new MazeGame();
    mg.CreateMaze();
  }
}
```

Figure 5: Maze Game example.

analysis. The goal of those extensions is to enable the system to use more efficiently code that is not written specifically for embedded systems, and thus makes use of those patterns, which are usually considered bad practice in constrained environments, due to a high cost.

Our analysis is fully verifiable by verifying the underlying alias analysis, since all the required additional computation can be performed at a low cost in the embedded system.

References

[1] Matthew Q. Beers, Christian H. Stork, and Michael Franz. Efficiently verifiable escape analysis. In M. Odersky, editor, *ECOOP 2004, LNCS 3086*, pages 75–95, Oslo, 2004. Springer-Verlag.

[2] Bruno Blanchet. Escape Analysis for Object Oriented Languages. Application to Java(TM). In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 20–34, Denver, Colorado, November 1999.

[3] Jeff Bogda and Urs Holzle. Removing unnecessary synchronization in java. Technical report, Santa Barbara, CA, USA, 1999.

[4] Antoine Galland. *Contrôle des ressources dans les cartes à microprocesseur*. PhD thesis, Pierre and Marie Curie University, 2005.

[5] R. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

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

[7] G. C. Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, January 1997.

[8] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, volume 27, pages 116–127, New York, NY, 1992. ACM Press.

[9] E. Rose and K. H. Rose. Lightweight bytecode verification. In *Workshop "Formal Underpinnings of the Java Paradigm", OOPSLA'98*, 1998.

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

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399